Exact Computation of the Adjacency Graph of an Arrangement of Quadrics

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz,

vorgelegt von
Michael Hemmer
geboren in Püttlingen.

Mainz, den 18. September 2007

Abstract

We present a complete, exact and efficient algorithm to compute the adjacency graph of an arrangement of quadrics, *i.e.* surfaces of algebraic degree 2. This is a major step towards the computation of the full 3D arrangement. We enhanced an implementation [58] for an exact parameterization of the intersection curves of two quadrics [23, 24, 25], such that we can compute the exact parameter value for intersection points and from that the adjacency graph of the arrangement. Our implementation is *complete* in the sense that it can handle all kinds of inputs including all degenerate ones, *i.e.* singularities or tangential intersection points. It is *exact* in that it always computes the mathematically correct result. It is *efficient* measured in running times, *i.e.* it compares favorably to the only previously implemented approach.

Our approach has been implemented within the EXACUS [6] project. The central goal of EXACUS is the development of a demonstrator of a reliable and efficient CAD geometry kernel. Although we call our library design prototypical, we spent nonetheless a great effort on completeness, exactness, efficiency, documentation and reusability. Beside its primary contribution, the algorithm presented in this work had an essential impact on fundamental parts of EXACUS due to its specific requirements. This work has in particular contributed to the generic number type support and the modular methods used within EXACUS. In the context of our ongoing integration of EXACUS into CGAL [33, 54] these parts have been successfully advanced into mature CGAL packages.

Zusammenfassung

Präsentiert wird ein vollständiger, exakter und effizienter Algorithmus zur Berechnung des Nachbarschaftsgraphen eines Arrangements von Quadriken (Algebraische Flächen vom Grad 2). Dies ist ein wichtiger Schritt auf dem Weg zur Berechnung des vollen 3D Arrangements. Dabei greifen wir auf eine bereits existierende Implementierung [58] zur Berechnung der exakten Parametrisierung der Schnittkurve von zwei Quadriken [23, 24, 25] zurück. Somit ist es möglich, die exakten Parameterwerte der Schnittpunkte zu bestimmen, diese entlang der Kurven zu sortieren und den Nachbarschaftsgraphen zu berechnen. Wir bezeichnen unsere Implementierung als vollständig, da sie auch die Behandlung aller Sonderfälle wie singulärer oder tangentialer Schnittpunkte einschließt. Sie ist exakt, da immer das mathematisch korrekte Ergebnis berechnet wird. Und schließlich bezeichnen wir unsere Implementierung als effizient, da sie im Vergleich mit dem einzigen bisher implementierten Ansatz gut abschneidet.

Implementiert wurde unser Ansatz im Rahmen des Projektes Exacus [6]. Das zentrale Ziel von Exacus ist es, einen Prototypen eines zuverlässigen und leistungsfähigen CAD Geometriekerns zu entwickeln. Obwohl wir das Design unserer Bibliothek als prototypisch bezeichnen, legen wir dennoch größten Wert auf Vollständigkeit, Exaktheit, Effizienz, Dokumentation und Wiederverwendbarkeit. Über den eigentlich Beitrag zu Exacus hinaus, hatte der hier vorgestellte Ansatz durch seine besonderen Anforderungen auch wesentlichen Einfluss auf grundlegende Teile von Exacus. Im Besonderen hat diese Arbeit zur generischen Unterstützung der Zahlentypen und der Verwendung modularer Methoden innerhalb von Exacus beigetragen. Im Rahmen der derzeitigen Integration von Exacus in Cgal [33, 54] wurden diese Teile bereits erfolgreich in ausgereifte Cgal Pakete weiterentwickelt.

Acknowledgement

I would like to thank all participants of the EU-funded projects ECG and ACS, with whom I enjoyed working. I am in particular thankful to all members from the CGAL Editorial Board for their advice and many useful remarks.

This work was partially supported by the IST Program of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473
(ECG - Effective Computational Geometry for Curves and Surfaces).

This work was partially supported by the IST Program of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413
(ACS - Algorithms for Complex Shapes with certified numerics and topology).

Preamble

In 2002 six European research groups at INRIA Sophia-Antipolis, University Groningen, ETH Zürich, FU-Berlin, Tel-Aviv and MPII Saarbrücken conceived the ECG-project (*Effective Computational Geometry for Curves and Surfaces*). Enriched by a research group from Athens and GeometryFactory this project was succeeded by another EU-funded project, namely the ACS-project (*Algorithms for Complex Shapes with certified numerics and topology*) which started in 2005.

The platform for the majority of software joint ventures within both projects was and is CGAL, the Computational Geometry Algorithms Library. CGAL is the state-of-the-art in implementing geometric algorithms completely, exactly, and efficiently. However, when CGAL was started in 1996 it was designed having linear geometry in mind and in particular the number type support of CGAL was not able to cope with the new algebraic challenges. Moreover, CGAL was already a huge library with many users and therefore it seemed quite intricate to change such fundamental issues. Therefore, the group at the MPI decided to initially outsource its investigations for curved geometry from CGAL and conceived the EXACUS-project (*Efficient and Exact Algorithms for Curves and Surfaces*) in April 2002. In the past years EXACUS proved to be a good laboratory for our ideas. In order to profit from these experiences and from synergy effects, the ACS partners finally decided to integrate core parts of EXACUS into CGAL and in particular the algebraic layer of EXACUS.

The work presented in this thesis was started within the context of the ECG project. Its major result namely an exact, complete and efficient algorithm for computing the adjacency graph of quadrics has been implemented within EXACUS. However, due to its specific requirements it had an essential impact on fundamental parts of EXACUS. In particular, it has contributed to the generic number type support and the modular methods used within EXACUS. In the context of our ongoing integration of EXACUS into CGAL these parts have been successfully advanced into mature CGAL packages.

According to this apportionment this thesis consists of two parts. Chapter 1 is dedicated to the algorithm computing the adjacency graph, whereas the new algebraic foundations for CGAL and further implementation issues are discussed in Chapter 2.

Contents

Chapter 1

Towards the exact 3D Arrangement of Quadrics

1.1 Introduction

Computer Aided Design (CAD) systems dealing with curved objects have been available since the 60's. However, all these systems still suffer from approximation and rounding errors due to the use of fast but inexact floating point arithmetic. This is due to the fact that without an exact representation of the resulting curves and without exact arithmetic it is difficult, or just impossible, to detect degenerate configurations, which are frequent in the design of geometric objects. As a consequence, non of these systems is either exact or complete. On the other hand, computer algebra introduced very general methods, such as cylindrical algebraic decomposition presented by Collins [17]. These methods are exact and complete in principle but cause unacceptable runtimes. Our intention is to close this gap, and join the three goals *exactness*, *completeness* and *efficiency*.

We decided to initially aim for an *exact*, *complete* and *efficient* implementation computing the 3D arrangement of quadric surfaces. This decision was mainly driven by the following reasons.

- A recurring and important task in solid modeling is to perform boolean operations on curved surfaces. However, the cardinal problem behind this task is the computation of the underlying 3D arrangement of the involved

surfaces. Once the arrangement is computed, performing the boolean operation is just a combinatorial problem. See Halperin *et al.* [41] for a good and brief overview on arrangements.

- Quadrics, surfaces of algebraic degree 2, are the simplest curved surfaces. They play an important role in solid modeling and in the design of mechanical parts, *e.g.* patches of natural quadrics (planes, cones, spheres and cylinders) and tori make up to 95% of all mechanical pieces according to Requicha and Voelcker [68].

So far we have not solved the complete problem of computing the full 3D arrangement of quadric surfaces. However, we achieved a major milestone, namely the computation of the adjacency graph connecting the vertices of the 3D arrangement. Our prototype is implemented within the framework of the EXACUS project. It is *complete* in the sense that it can handle all kind of inputs including all degenerate ones, where intersection curves have singularities or pairs of curves intersect with high multiplicity. It is *exact* in the sense that it always computes the mathematical correct result. It is *efficient* measured in running times, *i.e.* it compares favorably to the only previous implementation by Berberich *et al.* [9].

A short version of this work has been published in cooperation with Sylvain Petitjean, Laurent Dupont and Elmar Schömer.

1.1.1 Previous Work

Extending algorithms well known in Computational Geometry for linear primitives to exact, complete and efficient algorithms for curved objects has received a lot of attention during the last years. For instance the exact computation of planar arrangements of curved objects arouse active research in computational geometry, since this is often an important subproblem when dealing with curved primitives. Wein [82] implemented an exact traits class for CGAL's planar arrangement package [34] to support planar maps of conics and conic arcs. Berberich et al. [7] implemented a similar technique for conic arcs based on the improved LEDA [62] implementation of the Bentley-Ottmann sweep-line algorithm [4]. Eigenwillig et al. [29] extended this framework to cubic curves. Both algorithms have been implemented within the context of the EXACUS project and are available through the libraries CONIX and CUBIX, respectively. Very

recently, Kerber *et al.* [27, 26] presented an algorithm to compute the planar arrangement induced by segments of arbitrary algebraic curves with the Bentley-Ottmann sweep-line algorithm. The method makes essential use of the Bitstream Descartes method presented by Eigenwillig *et al.* [28] and reduces the geometric primitives to the cylindrical algebraic decompositions of the plane for one or two curves. It produces the mathematically true arrangement, undistorted by rounding error, for any set of input segments. The algorithm has been developed within the context of the EXACUS project and is provided in the C++ library ALCIX.

With respect to algorithms and systems dealing with curved objects in three dimensions one firstly should mention ESOLID by J. Keyser *et al.* [19]. This geometric solid modeler provides accurate CSG-tree to B-rep conversion for low-degree curved solids, in particular for quadric surfaces. The approach is based on the study of the intersection curves within the parameter space of each surface. But it follows a very general approach with the drawback that the polynomial degree needed to represent the appearing intersection curves is not optimal. Moreover, it is not able to handle all degenerate situations and has to assume that all solids are in general position, *i.e.* the system may fail or crash on degenerate inputs. Hence, the system can not be considered as *complete*, not even for quadric surfaces.

To the best of our knowledge, there are only three approaches targeting towards the computation of the 3D arrangement of quadric surfaces.

- The first approach, by Mourrain *et al.* [64], is based on a spatial sweep over the arrangement of quadric surfaces. It defines a pseudo trapezoidal decomposition in the sweep plane and studies the evolution of this decomposition during the sweep. The approach is based on Rational Univariate Representation [69] of the roots of the appearing multivariate systems. However, this has to be considered as disadvantage since the algebraic degree of the needed predicates can be very high. Hence, it is unclear whether this exact and complete approach can be considered as efficient, *i.e.* it has not been implemented so far.

- The second approach by Berberich *et al.* [9] is based on Wolpert [71, 84] and Eigenwillig *et al.* [29]. The approach has been developed in the context of the EXACUS project and is implemented in the C++ library QUADRIX

in a very mature way. However, it still does not solve the complete problem, namely the computation of the full 3D arrangement of quadrics. It computes, for a given set $Q = \{q_1, \ldots, q_n\}$ of quadric surfaces, the planar map induced by all intersection curves $q_1 \cap q_i$, $2 \leq i \leq n$, running on the surface of q_1. This is done by first computing the planar arrangement of the projections of all intersection curves and the silhouette curve of q_1 and then lifting this arrangement back onto the surface of q_1. Since we will compare our own approach to this *projection approach*, it is again discussed in section 1.9.3, illuminating more details of the implementation inasmuch as they are relevant for a better understanding of the benchmarks.

- The last approach by Dupont *et al.* [23, 24, 25] computes an exact parametric representation of the intersection curve of two quadrics in a very efficient way. The output functions parameterizing the intersection are rational functions whenever it is possible, which is the case when the intersection is not a smooth quartic (for example, a singular quartic, a cubic and a line, or two conics). The coefficient field of the parameterization is either minimal or involves one possibly unneeded square root. Unlike existing implementations, it correctly identifies and parameterizes all the connected components of the intersection in all the possible cases. The algorithm is available via the library QI implemented by Lazard *et al.* [58]. It is the basis for the approach presented in this chapter and is discussed in more detail in section 1.5.

Up to now none of the three approaches has finally lead to a complete algorithm nor to an implementation for computing the 3-dimensional arrangement. Only the 3-dimensional event-points can be computed so far, but the final step of connecting this information to a complete picture of the 3D arrangement is still missing.

1.1.2 Outline

Given a finite set $\mathcal{S}$ of geometric objects the *arrangement* $\mathcal{A}(\mathcal{S})$ is the *subdivision* of the space into cells as induced by the objects in $\mathcal{S}$, that is, each cell is defined as a maximal connected area $\mathcal{C}$ in space such that all points in $\mathcal{C}$ have identical relation to all objects in $\mathcal{S}$. Cells are classified by their dimension, for

instance a 3-dimensional arrangement is comprised of vertices, edges, faces and (3-dimensional) cells.

We are interested in the computation of the 3-dimensional arrangement of quadric surfaces. Hence, it is first of all important to comprehend the nature of quadrics and their possible intersection types. Though the algebraic degree of quadrics is just 2, they cover a couple of common surfaces such as spheres, ellipsoids, cones, cylinders, hyperboloids but also planes and double planes. In general the intersection of two quadrics is a quartic, a curve of algebraic degree 4. However, in some cases the intersection may decompose into *algebraic components* of lower degree, *i.e.* cubics, conics or lines or even isolated points. An exact definition and full classification of quadrics is given in Section 1.2. The possible intersection types of quadric surfaces are discussed in Section 1.3.

Our algorithm is based on an exact parameterization of the appearing intersection curves by Dupont *et al.* [23, 24, 25]. The algorithm by Dupont identifies, separates and parameterizes all algebraic components in an exact manner. It gives all the information on the incidence between the components, *i.e.* it reports where and how (e.g., tangentially or not) two components intersect. The algorithm is complete, that is, it places no restriction of any kind on the input quadrics and the type of their intersection. Since this is based on a prior algorithm by Levin [59, 60], this algorithm is briefly discussed in Section 1.4. Thereafter, Dupont's algorithm is summarized in Section 1.5.

The fundamental idea of our approach is to compute and represent the points in the arrangement with respect to this parameterization, that is, we represent the points by their exact parameter values with respect to the parameterizations of the algebraic components they lie on. This has the advantage that we can easily determine the adjacency of points by sorting them along the curves. However, there are some stumbling blocks as well.

- In all but the generic case the intersection of an algebraic component with a third quadric is straightforward, since the parameterization is given in terms of rational functions. However, in the case of a smooth quartic, the parameterization is not rational as it involves a square root of a polynomial. Hence, this case requires a separate and more sophisticated consideration. The intersection of algebraic components with another quadric is discussed in Section 1.6.

- For each intersection point we obtain several representations, one representation for each algebraic component the point lies on. Since each intersection point should be represented by one object only, it is essential to identify and join representations representing the same point. Unfortunately, it is very expensive to compare just two representations on different algebraic components due to the degree of the involved algebraic expressions. However, given all representations of the intersection points of three quadrics, we found a very efficient way to match these representations all at once. Our solution is presented in Section 1.7.

- It is not possible to avoid the construction of another parameterization of the same algebraic component in all cases. Moreover, the algorithm by Dupont *et al.* can not guarantee a unique parameterization of algebraic components. This entails two problems. First, we have to identify equal algebraic components before we start to intersect them with the other quadrics. Second, before we delete a redundant parameterization of an algebraic component, we have to 'rescue' the intersection points which are already defined with respect to this algebraic component. Our solution to this problem is discussed in Section 1.8.

- Though the polynomial degree of the parameterization is minimal it considerably increases the complexity on the level of coefficients. This refers to both, the bit size as well as the algebraic complexity of the coefficients. Therefore, the approach demanded an elaborated application of filtering techniques throughout the algorithm. Now, the approach compares favorably to other existing methods, as it is documented in Section 1.9.

Draft: Overall Algorithm

We are now ready to state a brief version of the overall algorithm, a more detailed version is given in Section 1.1.2.

Given a set $\mathcal{S}$ of quadric surfaces, defined by rational coefficients of any size. Our algorithm computes the adjacency graph $\mathcal{G}(\mathcal{S})$ of the arrangement $\mathcal{A}(\mathcal{S})$, that is, it computes all vertices and their connectivity along the edges of $\mathcal{A}(\mathcal{S})$. Note that since the class of quadric surfaces covers double planes, *i.e.* rational planes, our algorithm is capable to handle rational planes as well.

For a given set $\mathcal{S}$ of rational quadrics do:

0. Remove duplicates from $\mathcal{S}$ and ensure that all quadrics are coprime. If quadrics are not coprime, replace them by their common factor and the according remainders, *i.e.* rational planes
1. Construct all lower dimensional features induced by one quadric, *e.g.* the singular point of a cone.
2. For each pair of quadrics, compute all algebraic components of their intersection using the approach by Dupont *et al.*. Ensure for each component that it is new, otherwise unify it with the existing one.
3. For each triple of quadrics take the components of each pair and intersect these components with the third quadric in the triple. This results in at least three representations for each intersection point, one for each component it lies on. Match all representations representing the same intersection point and join them into one vertex. Note that each vertex stores several representations, one for each component it lies on.
4. Sort the vertices along common algebraic components. Output the adjacency graph.

Since the vertices are sorted along their algebraic components and since every vertex knows all its algebraic components it lies on, it is easy to compute the full adjacency graph connecting all vertices of the arrangement with their neighbors in $\mathbb{P}^3$. The next step is to explore the local neighborhood of each vertex. For instance, this could be done by computing the arrangement of conics on a sufficiently small cube around each vertex. Therewith, the arrangement itself could be stored in a variant of a structure used to represent the so called *Selective Nef Complex* as presented in [35]. This structure is a vertex oriented structure, that stores the local neighborhood around each vertex in a so called *Sphere Map*, see Section 1.10 on further work for more details.

1.2 Quadric Surfaces

Though we work in projective space $\mathbb{P}^3 = \mathbb{P}(\mathbb{R})^3$, several propositions are given for any dimension. In what follows, all the matrices considered are real square matrices.

Given a real symmetric matrix S of size $n+1$, the upper left submatrix of size n, denoted S_u, is called the *principal submatrix* of S and the determinant of S_u the *principal subdeterminant* of S.

Two real symmetric matrices S and S' of size n are said to be *similar* if and only if there exists a real nonsingular matrix P such that

$$S' = P^{-1}SP. \tag{1.1}$$

Note that two similar matrices have the same characteristic polynomial, and thus the same eigenvalues.

Two matrices are said to be *congruent* or *projectively equivalent* if and only if there exists a nonsingular matrix P with real coefficients such that

$$S' = P^{T}SP. \tag{1.2}$$

Note that the determinant of S is invariant by a congruence transformation, up to a square factor, *i.e.* the square of the determinant of the transformation matrix.

1.2.1 Definition

A *quadric surface*, or quadric for short, is defined in $\mathbb{P}^3$ by an implicit equation of degree 2:

$$\sum_{0\leq i\leq j\leq 3} \alpha_{ij}x_ix_j = 0, \text{ with } \alpha_{ij} \in \mathbb{R}. \tag{1.3}$$

This can be written as $\mathbf{x}^T S\mathbf{x} = 0$, where S is a real symmetric matrix of size 4. Hence, we denote the quadric associated to S by

$$Q_S = \{\mathbf{x} \in \mathbb{P}^3 \mid \mathbf{x}^T S\mathbf{x} = 0\}. \tag{1.4}$$

Note that for every $\alpha \in \mathbb{R} \setminus \{0\}$ the quadrics associated to αS are equal to the quadric Q_S. When the ambient space is $\mathbb{R}^3$ instead of $\mathbb{P}^3$, the quadric is simply Q_S minus its points at infinity.

1.2.2 Classification by Inertia

Since S is symmetric, it is clear that all of its eigenvalues are real. Let σ^+ and σ^- be the numbers of positive and negative eigenvalues of S, respectively. The *rank* of S is the sum of σ^+ and σ^-. We define the *inertia* of S as the pair

$$(max(\sigma^+, \sigma^-), min(\sigma^+, \sigma^-)). \tag{1.5}$$

This definition slightly differs from the usual definition of the inertia in the literature, which in general denotes the inertia of S as the pair (σ^+, σ^-). However, the definition chosen here effectively reflects the fact that the associated quadrics of S and $-S$ are one and the same quadric.

For convenience we will indicate, in some case, the inertia by the triple

$$(max(\sigma^+, \sigma^-), min(\sigma^+, \sigma^-), n - rank), \tag{1.6}$$

where n is the size of S.

Theorem 1 (Sylvester's Inertia Law). *For each diagonal form received by a real nonsingular transformation of a real quadric form A, beside the rank r, the number p_A of positive diagonal elements (inertia index), and consequently also the number $q_A = r - p_A$ of negative diagonal elements is invariant.*

This theorem is named for the English mathematician J. J. Sylvester[1]. A proof of this fundamental result of matrix theory can be found, *e.g.*, in [11], [36] or [56]. It essentially states that the inertia is invariant under change of basis. Thus, we identify the projective type of a quadric Q_S by the inertia of S.

We discuss the classification along the rank of the matrix S for quadrics in $\mathbb{P}^3$:

- **rank 4:** A quadric of inertia $(4, 0)$ is an empty quadric, *i.e.* empty of real points. The only quadrics with a negative determinant are those of inertia $(3, 1)$. All quadrics of inertia $(2, 2)$ are ruled quadrics, *i.e.* they can be swept out by moving a line in space. This is a very important property, since this can be used to provide a very convenient parameterization, see also Section 1.5. Note that all quadrics with positive determinant are either ruled or empty.

[1] James Joseph Sylvester (* September 3, 1814 London; † March 15, 1897 Oxford)

- **rank 3:** A quadric of rank 3 is called a cone. The cone is said to be real if its inertia is $(2, 1)$. If the inertia is $(3, 0)$ it is an imaginary cone, with the singular point being its only real solution.
- **rank 2:** A quadric of rank 2 is a pair of planes. The pair of planes is real if the inertia is $(1, 1)$. If the inertia is $(2, 0)$ the quadric consists of two imaginary planes intersecting in a real rational line.
- **rank 1:** A quadric of inertia $(1, 0)$ is called a double plane and is necessarily real.

For a classification of Q_S in $\mathbb{R}^3$ it is also necessary to examine the inertia of S_u, the *principal submatrix* of S. Table 1.1 recalls the correspondence of inertias in $\mathbb{P}^3$ and the classical Euclidean types of quadrics in $\mathbb{R}^3$.

Table 1.1: Correspondence of quadric inertias and Euclidean types.

inertia of $\mathbf{S}$	inertia of $\mathbf{S_u}$	Euclidean canonical equation	Euclidean type of Q_S
(4,0,0)	(3,0,0)	$x^2+y^2+z^2+1$	$\emptyset$(imaginary ellipsoid)
(3,1,0)	(3,0,0)	$x^2+y^2+z^2-1$	ellipsoid
	(2,1,0)	$x^2+y^2-z^2+1$	hyperboloid of two sheets
	(2,0,1)	x^2+y^2+z	elliptic paraboloid
(2,2,0)	(2,1,0)	$x^2+y^2+z^2$	hyperboloid of one sheet
	(1,1,1)	x^2+y^2+1	hyperbolic paraboloid
(3,0,1)	(3,0,0)	$x^2+y^2+z^2$	singular point
	(2,0,1)	x^2+y^2+1	$\emptyset$(imaginary elliptic cylinder)
(2,1,1)	(2,1,0)	$x^2+y^2-z^2$	cone
	(2,0,1)	x^2+y^2-1	elliptic cylinder
	(1,1,1)	x^2-y^2+1	hyperbolic cylinder
	(1,0,2)	x^2+y	parabolic cylinder
(2,0,2)	(2,0,1)	x^2+y^2	line
	(1,0,2)	x^2+1	$\emptyset$(imaginary parallel planes)
(1,1,2)	(1,1,1)	x^2-y^2	intersecting planes
	(1,0,2)	x^2-1	parallel planes
	(0,0,3)	x	simple plane
(1,0,3)	(1,0,2)	x^2	double plane
	(0,0,3)	1	$\emptyset$

1.3 Intersecting two Quadrics

Before we turn to the exact parameterization of intersection curves in Section 1.5 it is important to understand and classify the different types in which two quadrics can intersect. The complete classification of the intersection of two quadrics in $\mathbb{P}(\mathbb{C})^n$ by Segre [72] is recalled in Section 1.3.1. The complete classification of the intersection in $\mathbb{P}(\mathbb{R})^3$ by Dupont *et al.* [24] is discussed in Section 1.3.2.

1.3.1 Classification by Segre-Characteristic

This section recalls the classical characterization of the intersection of two quadric surfaces in $\mathbb{P}(\mathbb{C})^n$. This characterization was introduced by the Italian mathematician C. Segre[2] [72]. More recent descriptions can be found in [11] and [50].

Let S and T be two real symmetric matrices, the *pencil* is the set of their linear combinations

$$\{R(\lambda,\mu) = \lambda S + \mu T \mid (\lambda,\mu) \in \mathbb{P}^1\}. \tag{1.7}$$

For the sake of simplicity we will, in some cases, refer to a member of this pencil as $R(\lambda) = \lambda S + T$, where $\lambda \in \overline{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$.

An essential observation is that the intersection of the associated quadrics Q_S and Q_T can be identified by the corresponding pencil and vice versa. This is due to the fact that the set of common solutions of $x^T S x = 0$ and $x^T T x = 0$ is stable under linear combinations, *i.e.*

$$Q_S \cap Q_T = Q_{R(\lambda_i)} \cap Q_{R(\lambda_j)},\ \forall\ \lambda_{i,j} \in \overline{\mathbb{R}},\ \text{with } \lambda_i \neq \lambda_j \tag{1.8}$$

Elementary Divisors

Given a pencil $R(\lambda,\mu) = \lambda S + \mu T$ of symmetric matrices of size n, we denote the homogeneous polynomial

$$\mathcal{D}(\lambda,\mu) = det(R(\lambda,\mu)) \tag{1.9}$$

[2] Corrado Segre (*August 20, 1863 in Saluzzo; †May 18, 1924 in Turin)

the *characteristic polynomial* or the *principal invariant* of the pencil. A pencil is denoted a *regular pencil* if $\mathcal{D}(\lambda,\mu) \not\equiv 0$. In this generic case $\mathcal{D}(\lambda,\mu)$ has n distinct complex roots. Each root corresponds to one of the n complex projective cones contained in the pencil. The pencil is denoted a *singular pencil* if $\mathcal{D}(\lambda,\mu) \equiv 0$, that is, all quadrics in the pencil are singular.

Let $R(\lambda,\mu)$ be a regular pencil, and (λ_0,μ_0) be a root of $\mathcal{D}(\lambda,\mu) \not\equiv 0$ and m_0 its multiplicity. This root corresponds to $R(\lambda_0,\mu_0)$, one of the singular quadrics in $R(\lambda,\mu)$. In general this is a cone, *i.e.* of rank $n-1$. If the rank of $R(\lambda_0,\mu_0)$ drops further, say $n-t_0$, this implies that its corresponding root (λ_0,μ_0) has at least multiplicity t_0. This is due to the fact that (λ_0,μ_0) is a root of all subdeterminants of order $n-t_0+1$ of $R(\lambda,\mu)$. However, the inverse is not true. We denote m_0 the *algebraic multiplicity* and t_0 the *geometric multiplicity* of (λ_0,μ_0).

Let (λ_i,μ_i) be some real root of $\mathcal{D}(\lambda,\mu)$ and m_i be its respective algebraic multiplicity. Indicate by m_i^j the multiplicity of which (λ_i,μ_i) appears in the *gcd* of all subdeterminants of order $n-j$ of $R(\lambda,\mu)$. Let $t_i > 0$ be the smallest integer such that $m_i^{t_i} = 0$. Note that $m_i^j > m_i^{j+1}$ for all j.

Define a sequence of indices e_i^j as follows:

$$e_i^j = m_i^{j-1} - m_i^j, \text{ for } j = 1 \ldots t_i. \tag{1.10}$$

Note that, the multiplicity $m_i = m_i^0$ of (λ_i,μ_i) is the sum $e_i^1 + \cdots + e_i^{t_i}$. We have therefore:

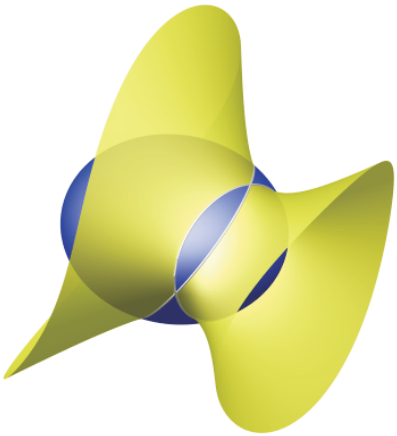
$$\begin{aligned} \mathcal{D}(\lambda,\mu) &= (\mu_i\lambda - \lambda_i\mu)^{m_i}\mathcal{D}^*(\lambda,\mu) && (1.11)\\ &= (\mu_i\lambda - \lambda_i\mu)^{e_i^1}\cdots(\mu_i\lambda - \lambda_i\mu)^{e_i^{t_i}}\mathcal{D}^*(\lambda,\mu), && (1.12) \end{aligned}$$

where $\mathcal{D}^*(\lambda_i,\mu_i) \neq 0$. The factors $(\mu_i\lambda - \lambda_i\mu)^{e_i^j}$ are called the *elementary divisors* associated to the root (λ_i,μ_i). In the literature elementary divisors are often denoted by the German word *Elementarteiler*, as their study goes back to K. Weierstrass[3] [81], who essentially coined the concept of elementary divisors.

The exponents e_i^j associated with the root (λ_i,μ_i) are denoted as the *characteristic numbers*. Segre introduced the following notation to denote the various

[3]Karl Weierstrass (*October 31, 1815 in Ostenfelde; †February 19, 1897 in Berlin)

[112]: Nodal quartic.



[11(11)]: Two secant conics.

characteristic numbers associated with the degenerate quadrics that appear in a regular pencil in $\mathbb{P}^n$:

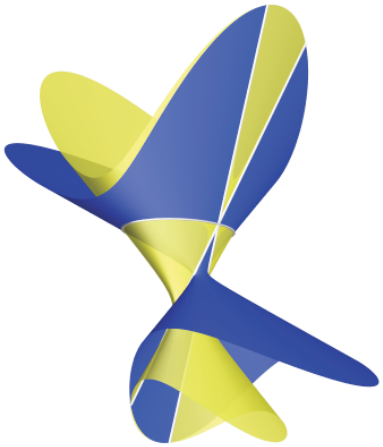
$$\sigma_n = [(e_1^1, \ldots, e_1^{t_1}), (e_2^1, \ldots, e_2^{t_2}), \ldots, (e_k^1, \ldots, e_k^{t_k})], \tag{1.13}$$

with the convention that the parentheses enclosing the characteristic numbers of (λ_i, μ_i) are dropped when $t_i = 1$. This is known as the *Segre-Characteristic* or *Segre-Symbol* of the pencil. For instance the symbol [11(11)] indicates that the polynomial $\mathcal{D}(\lambda_i, \mu_i)$ has a double root, for which the rank drops by 2. Whereas the symbol [11(2)] = [112] indicates a double root, for which the rank just drops by one. However, it does not always suffice to argue by the rank of the singular quadrics in the pencil, *e.g.* there is a difference between the Segre Characteristics [(22)] and [(31)], *i.e.* the distinction by the Segre-Characteristic is really needed.

The following theorem, essentially due to Weierstrass [81], proves that a pencil of quadrics and the intersection it defines are uniquely and entirely characterized, over the complex numbers, by its Segre-Symbol.

Theorem 2 (Segre-Characteristic). *Consider the two pencils $R(\lambda, \mu) = \lambda S + \mu T$ and $R(\lambda', \mu') = \lambda' S' + \mu' T'$ in $\mathbb{P}^n$. Suppose that $\mathcal{D}(\lambda, \mu)$ and $\mathcal{D}(\lambda', \mu')$ are not identically zero and let (λ_i, μ_i) and (λ_i', μ_i') be their respective roots ordered by their characteristic numbers. Then the two pencils are projectively equivalent if*

[(22)]: Two lines and double line.



[(31)]: Two lines crossing on conic.

and only if they have the same Segre-Characteristic and there is an automorphism of $\mathbb{P}(\mathbb{C})^n$ *taking* (λ_i, μ_i) *to* (λ'_i, μ'_i).

Hence, it is possible to enumerate all cases for $\mathcal{D}(\lambda, \mu) \not\equiv 0$ using the Segre-Characteristic, see Table 1.2 for a list of all cases in $\mathbb{P}(\mathbb{C})^3$. If $\mathcal{D}(\lambda, \mu) \equiv 0$ the pencil is singular and the above theory does not apply directly. There are two cases:

- All quadrics in the pencil share at least one singular point p. Assuming, *w.l.o.g.*, that p has coordinates $(0, \ldots, 0, 1)$ it follows that the last row and column of all matrices are filled with zeros. Hence, we can use the Segre-Symbol σ_{n-1} of their principal submatrix to sort out the different types.
- If the quadrics do not share a common singular point, a different set of invariants by Leopold Kronecker[4] is used. For a detailed discussion of these invariants see [11, p. 55-60]. However, for our application in dimension $n = 4$ and $n = 3$, it suffices to say that for each n there is only one set of such invariants, which is denoted by the strings [1{3}] and [{3}], respectively.

This process can be repeated by recursing on dimension, resulting in a classification by Segre-Symbols σ_3 and σ_2, see Table 1.3 and Table 1.4 respectively.

[4]Leopold Kronecker (*December 7, 1823 in Liegnitz; †December29, 1891 in Berlin)

Table 1.2: Classification of pencil by Segre-Symbol σ_4.

Segre-Symbol σ_4	Complex Type of Intersection
[1111]	smooth quartic
[112]	nodal quartic
[11(11)]	two secant conics
[13]	cuspidal quartic
[1(21)]	two tangent conics
[1(111)]	double conic
[4]	cubic and tangent line
[(31)]	conic and two lines crossing on the conic
[(22)]	two lines and a double line
[(211)]	two double lines
[(1111)]	any non trivial quadric of the pencil
[22]	cubic and secant line
[2(11)]	conic and two lines not crossing on the conic
[(11)(11)]	four skew lines

1.3.2 Classification by Canonical Pair Form

In some cases the characterization by the Segre-Symbol suffices to characterize the real intersection as well. For instance the Segre-Symbol $\sigma_4 = [13]$ corresponds to a cuspidal quartic, which is always real. The same holds for $\sigma_4 = [4]$, which corresponds to a cubic and a tangent line. However, in general the Segre-Symbol is not sufficient to classify the real part of the intersection.

This section summarize the characterization of the intersection of two quadrics surfaces over the reals, as it is given by Dupont *et al.* [24]. This analysis is a major building block for their algorithm computing the exact parameterization of two quadrics, see also Section 1.5. The classification is obtained by an extensive case study based on the *Canonical Pair Form* introduced by Uhlig [76, 77].

Table 1.3: Classification of pencil by Segre-Symbol σ_3.

Segre-Symbol σ_3	Complex Type of Intersection
[1{3}]	conic and double line
[111]	four concurrent lines
[12]	two lines and double line
[1(11)]	two double lines
[3]	line and triple line
[(21)]	quadruple line
[(111)]	any non trivial quadric in the pencil

Table 1.4: Classification of pencil by Segre-Symbol σ_2.

Segre-Symbol σ_2	Complex Type of Intersection
[{3}]	line and plane
[11]	quadruple line
[2]	plane
[(11)]	any non trivial quadric in the pencil
$\mathcal{D}_2(\lambda,\mu)\equiv 0$	double plane

Let M be a square matrix of the form

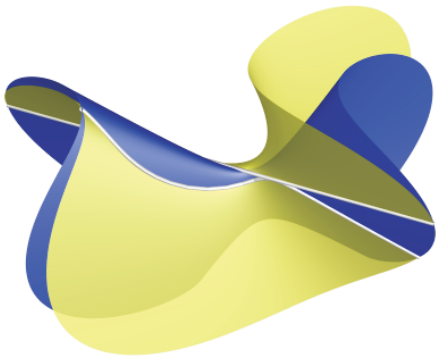
$$(\ell) \text{ or } \begin{pmatrix} \ell & e & & 0 \\ & \ddots & \ddots & \\ & & \ddots & e \\ 0 & & & \ell \end{pmatrix}. \tag{1.14}$$

If $\ell \in \mathbb{R}$ and $e = 1$, M is called a *real Jordan block* associated with ℓ. If

$$\ell = \begin{pmatrix} a & -b \\ b & a \end{pmatrix},\ a, b \in \mathbb{R},\ b \neq 0,\ e = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{1.15}$$

M is called a *complex Jordan block* associated with $\ell = a + ib$.

[31]: Cuspidal quadric.



[4]: Cubic and tangent line.

Now, recall the classical result on the Jordan Normal Form, named for the mathematician C. Jordan[5].

Theorem 3 (Real Jordan Normal Form). *Every real square matrix A is similar over the reals to a block diagonal matrix $J = diag(J_1, \ldots, J_k)$ in which each J_j is a (real or complex) Jordan block associated with an eigenvalue of A. The matrix J is called the real Jordan Normal Form of A.*

We are now ready to state the Canonical Pair Form Theorem:

Theorem 4 (Canonical Pair Form). *Let S and T be two real symmetric matrices of size n, with S nonsingular. Let $S^{-1}T$ have real Jordan normal form $diag(J_1, \ldots, J_r, J_{r+1}, \ldots, J_m)$, where $J_1, \ldots, J_r$ are real Jordan blocks corresponding to real eigenvalues of $S^{-1}T$ and $J_{r+1}, \ldots, J_m$ are complex Jordan blocks corresponding to pairs of complex conjugate eigenvalues of $S^{-1}T$. Then:*

(a) The characteristic polynomial of $S^{-1}T$ and $det(\lambda S - T)$ have the same roots λ_j with the same multiplicities m_j.

(b) S and T are simultaneously congruent by a real congruence transformation to

$$diag(\varepsilon_1 E_1, \ldots, \varepsilon_r E_r, E_{r+1}, \ldots, E_m) \tag{1.16}$$

[5]Camille Jordan (*January 5, 1838 in La Croix-Rousse, Lyon; †January 22, 1922 in Paris)

and

$$diag(\varepsilon_1 E_1 J_1, \ldots, \varepsilon_r E_r J_r, E_{r+1} J_{r+1}, \ldots, E_m J_m), \tag{1.17}$$

respectively, where $\varepsilon_i = \pm 1$ *and* E_i *denotes the square matrix*

$$\begin{pmatrix} 0 & & 1 \\ & \iddots & \\ 1 & & 0 \end{pmatrix}. \tag{1.18}$$

of the same size as J_i*. The signs* ε_i *are unique (up to permutations) for each set of indices* i *that are associated with a set of identical real Jordan blocks* J_i*.*

(c) The sum of the sizes of the blocks corresponding to one of the λ_j *is the multiplicity* m_j *for real* λ_j *or twice this multiplicity for complex* λ_j*. The number of the corresponding blocks (the geometric multiplicity of* λ_j*) is* $t_j = n - rank(\lambda_j S - T)$*,* $1 \leq t_j \leq m_j$*.*

The Canonical Pair Form maximizes the number of blocks in the diagonalization of S and T. Hence, it can be considered as the finest simultaneous block diagonal structure that can be obtained by real congruence for a given pair of real symmetric matrices. It is in some sense the real version of Theorem 2, which one can read as follows:

For a given pencil $R(\lambda) = \lambda S - T$ and its characteristic polynomial $\mathcal{D}(\lambda)$, with Segre-Symbol

$$[(e_1^1, \ldots, e_1^{t_1}), (e_2^1, \ldots, e_2^{t_2}), \ldots, (e_p^1, \ldots, e_p^{t_p})], \tag{1.19}$$

there exists a change of coordinates in $\mathbb{P}(\mathbb{C})^n$ such that, in the new frame, the pencil writes down as $R'(\lambda) = \lambda S' - T'$, where

$$S' = diag(E_1^1, \ldots, E_1^{t_1}, \ \ldots \ , E_p^1, \ldots, E_p^{t_p}) \tag{1.20}$$

and

$$T' = diag(E_1^1 J_1^1, \ldots, E_1^{t_1} J_1^{t_1}, \ \ldots \ , E_p^1 J_p^1, \ldots, E_p^{t_p} J_p^{t_p}), \tag{1.21}$$

are block diagonal matrices with blocks:

$$E_i^k = \begin{pmatrix} 0 & & 1 \\ & \iddots & \\ 1 & & 0 \end{pmatrix} \text{ and } J_i^k = \begin{pmatrix} \lambda_i & e & & 0 \\ & \ddots & \ddots & \\ & & \ddots & e \\ 0 & & & \lambda_i \end{pmatrix} \tag{1.22}$$

of size e_i^k.

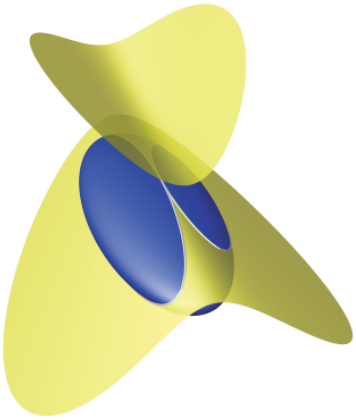
Contrariwise the Canonical Pair Form gives a form that is projectively equivalent by a real congruence transformation to the original pencil, *i.e.* the complex roots of the $\mathcal{D}(\lambda)$ are combined in complex Jordan blocks:

- If λ_i is a real root of $\mathcal{D}(\lambda)$. Let $(e_i^1, \ldots, e_i^{t_i})$ be the associated characteristic numbers, then there are t_i real Jordan blocks J_i^k of size e_i^k, $k = 1, \ldots, t_i$. This is symmetric to Theorem 2 (Segre-Characteristic).
- If λ_i is a complex root of $\mathcal{D}(\lambda)$ let λ_j be its conjugate. Since λ_i and λ_j are conjugated, it is clear that their associated characteristic numbers are equal, *i.e.* $(e_i^1, \ldots, e_i^{t_i}) = (e_j^1, \ldots, e_j^{t_j})$. While Theorem 2 (Segre-Characteristic) treats these roots separately, the Canonical Pair Form is restricted to real congruence transformations. Thus, in the Canonical Pair Form the Jordan blocks corresponding to λ_i and λ_j are reported as complex Jordan blocks J_i^k of size $2e_i^k$, $k = 1, \ldots, t_i$.

By this symmetry the Segre-Characteristic serves as the starting point for the study of real pencils using the Canonical Pair Form Theorem. For each possible Segre-Symbol, Dupont *et al.* [24] enumerate the different Canonical Pair Forms according to whether the roots of the $\mathcal{D}(\lambda)$ can be complex or not. Thereafter, for each possible Canonical Pair Form they deduce a normal form of the pencil over the reals by rescaling and translating the roots to particular simple values. This normal form of the pencil is equivalent by a real projective transformation to any pencil of quadrics with the same real and complex intersection type. We illustrate this with two short examples with Segre-Symbol [(21)1] (Two tangent conics) and [22] (Cubic and secant line). For more examples and a full table of all possible real intersection types see Dupont *et al.* [21, 24].

Example 1: $\sigma_4 = [(21)1]$ (Two Tangent Conics)

Let $R = \lambda S + T$ be a pencil in Canonical Pair Form with Segre-Symbol [(21)1]. $\mathcal{D}(\lambda)$ has a triple root λ_1 and a simple root λ_2. Obviously both roots are real roots. There are two Jordan blocks associated to λ_1, one block of size 2 and one block of size 1. And there is one block of size 1 which is associated to λ_2. Note



$\sigma_4 = [1(21)]$: Two tangent conics.

$\sigma_4 = [22]$: Cubic and secant line.

that we can assume, *w.l.o.g.*, that the sign associated to the first Jordan block is positive. Thus S and T are of the form:

$$S = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & \varepsilon_1 & 0 \\ 0 & \varepsilon_1 & 0 & 0 \\ 0 & 0 & 0 & \varepsilon_2 \end{pmatrix}, \; T = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & 0 & \varepsilon_1\lambda_1 & 0 \\ 0 & \varepsilon_1\lambda_1 & \varepsilon_1 & 0 \\ 0 & 0 & 0 & \varepsilon_2\lambda_2 \end{pmatrix} \tag{1.23}$$

The corresponding equations are

$$\begin{cases} 0 &= x^2 + 2\varepsilon_1 yz + \varepsilon_2 w^2, \\ 0 &= \lambda_1 x^2 + 2\varepsilon_1\lambda_1 yz + \varepsilon_1 z^2 + \varepsilon_2\lambda_2 w^2. \end{cases} \tag{1.24}$$

Hence, $R(\lambda_1) = \lambda_1 S - T$ and $R(\lambda_2) = \lambda_2 S - T$ are simultaneously congruent to

$$\begin{cases} 0 &= -\varepsilon_1 z^2 + \varepsilon_2(\lambda_1 - \lambda_2)w^2, \\ 0 &= (\lambda_2 - \lambda_1)x^2 + 2\varepsilon_1(\lambda_2 - \lambda_1)yz - \varepsilon_1 z^2 \\ &= (\lambda_2 - \lambda_1)x^2 + (-\varepsilon_1 z + 2\varepsilon_1(\lambda_2 - \lambda_1)y)z. \end{cases} \tag{1.25}$$

We can apply the following real projective transformation:

$$\begin{array}{rcl@{\qquad}rcl} \sqrt{|\lambda_1 - \lambda_2|}x & \mapsto & x, & -\varepsilon_1 z + 2\varepsilon_1(\lambda_2 - \lambda_1)y & \mapsto & y, \\ \sqrt{|\lambda_1 - \lambda_2|}w & \mapsto & w, & z & \mapsto & z. \end{array} \tag{1.26}$$

And we obtain:

$$\begin{cases} 0 &= -\varepsilon_1 z^2 + \varepsilon_2 \epsilon w^2 \\ 0 &= -\epsilon x^2 + yz \end{cases} \tag{1.27}$$

where $\epsilon = sign(\lambda_1 - \lambda_2)$. Finally we apply $\epsilon y \mapsto y$ and end up with:

$$\begin{cases} 0 &= z^2 + aw^2 \\ 0 &= x^2 + yz \end{cases}, \tag{1.28}$$

where $a = -\varepsilon_1\varepsilon_2\epsilon$. Hence, there are only two possible scenarios:

- $a = +1$: The inertia of $R(\lambda_1)$ is (2,0), that is, two imaginary planes intersecting in the real line $z = w = 0$. This real line touches the real cone $Q_{R(\lambda_2)}$ in the point $(0, 1, 0, 0)$, being the only real part of the intersection.
- $a = -1$: The inertia of $R(\lambda_1)$ is (1,1). There are two real planes intersecting in the real line $z = w = 0$. This real line touches the real cone $Q_{R(\lambda_2)}$ in the point $(0, 1, 0, 0)$. Hence, there are two real conics touching in $(0, 1, 0, 0)$ with common tangent $z = w = 0$.

Example 2 : $\sigma_4 = [22]$ (Cubic and Secant Line)

In case of $\sigma_4 = [22]$ the polynomial $\mathcal{D}(\lambda)$ has two double roots λ_1 and λ_2, each corresponding to a quadric of rank 3. There are two cases either λ_1 and λ_2 are both real or complex conjugate:

- **λ_1 and λ_2 are real:** In this case there are two Jordan blocks of size 2 each associated to one of the roots.

$$S = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \varepsilon \\ 0 & 0 & \varepsilon & 0 \end{pmatrix}, \; T = \begin{pmatrix} 0 & \lambda_1 & 0 & 0 \\ \lambda_1 & 1 & 0 & 0 \\ 0 & 0 & 0 & \varepsilon\lambda_2 \\ 0 & 0 & \varepsilon\lambda_2 & \varepsilon \end{pmatrix} \tag{1.29}$$

  As in the first example, we apply some transformations to $R(\lambda_1)$ and $R(\lambda_2)$. This results in the normal form:

$$\begin{cases} 0 &= y^2 + zw \\ 0 &= xy + w^2 \end{cases}. \tag{1.30}$$

  In this frame the intersection contains the line $z = w = 0$. The cubic is parameterized by

$$X(u, v) = (u^3, -uv^2, v^3, u^2v), \; (u, v) \in \mathbb{P}^1 \tag{1.31}$$

The cubic intersects the line in $(1,0,0,0)$ and $(0,0,1,0)$, the corresponding parameter values are $(1,0)$ and $(0,1)$.

- **λ_1 and λ_2 are complex conjugate:**

$$S = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \; T = \begin{pmatrix} 0 & 0 & \beta & \alpha \\ 0 & 0 & \alpha & -\beta \\ \beta & \alpha & 0 & 1 \\ \alpha & -\beta & 1 & 0 \end{pmatrix}, \tag{1.32}$$

This time we start with S and $R(\alpha)$. Again, we apply some real projective transformations. This results in the normal form:

$$\begin{cases} 0 &= xw + yz \\ 0 &= xz - yw + zw \end{cases}. \tag{1.33}$$

In this frame the intersection contains the line $z = w = 0$. The cubic is parameterized by

$$X(u,v) = (-u^2v, uv^2, u^3 + uv^2, u^2v + v^3), \; (u,v) \in \mathbb{P}^1 \tag{1.34}$$

The cubic intersects the line in the points $(1,i,0,0)$ and $(1,-i,0,0)$, *i.e.* there is no intersection of the cubic and the line in $\mathbb{P}(\mathbb{R})^3$.

1.4 Parameterization by Levin

In this section we tersely recall an approach to parameterize the intersection of two quadric surfaces, which has been introduced by Levin [59, 60]. Even tough his method is not feasible to be used in an exact algorithm, the exact approach taken by Dupont *et al.* comprises several ideas that have been introduced by Levin. The approach by Dupont is discussed in the subsequent Section 1.5.

Levin's method is based on the fact that any distinct pair of quadrics in a pencil induces the same intersection. Hence, the idea is not to adhere on the given quadrics but to chose a 'nice' quadric with a simple parameterization from the pencil. And indeed, if the intersection is not empty, it is always possible to find a quadric with a parameterization which is linear in at least one parameter, say v. Thereafter, this parameterization is plugged in another quadric of the pencil, yielding an equation which is just quadratic in v. The parameterization of the intersection curve is obtained by solving this equation with respect to v.

Since Levin works in $\mathbb{R}^3$, these 'nice' quadrics are the so called *simple ruled quadrics*, parameterizations of which are given in Table 1.5. The approach by Levin is based on the following theorem.

Theorem 5 (Levin). *Either the intersection of two quadric surfaces is empty, or it lies in a plane, a pair of planes, a hyperbolic or parabolic cylinder, or a hyperbolic paraboloid.*

For a given regular pencil $R(\lambda) = \lambda S + T$, Levin's algorithm can be summarized in the following four steps:

1. Since every simple ruled quadric has vanishing principal subdeterminant, see Table 1.1, compute the real roots λ_i of $\mathcal{D}_u(\lambda) = \det(R_u(\lambda))$. By Theorem 5 one of the according quadrics $R(\lambda_i)$ is simple ruled or the intersection is empty.

2. Let $Q_S \cap Q_T \neq \emptyset$ and R be the chosen simple ruled quadric. Bring R into diagonal form by computing an orthonormal transformation matrix P using a translation and the normalized eigenvectors of R_u:

$$R' = P^T R P. \tag{1.35}$$

Table 1.5: Parameterizations of canonical simple ruled quadrics.

quadric	canonical equation $(a, b > 0)$	parameterization $X(u,v) = [x, y, z],\ u, v \in \mathbb{R}$
simple plane	$x = 0$	$X(u,v) = [0, u, v]$
double plane	$x^2 = 0$	$X(u,v) = [0, u, v]$
parallel planes	$ax^2 = 1$	$X(u,v) = [\frac{1}{\sqrt{a}}, u, v]$, $X(u,v) = [-\frac{1}{\sqrt{a}}, u, v]$
intersecting planes	$ax^2 - by^2 = 0$	$X(u,v) = [\frac{u}{\sqrt{a}}, \frac{u}{\sqrt{b}}, v]$, $X(u,v) = [\frac{u}{\sqrt{a}}, -\frac{u}{\sqrt{b}}, v]$
hyperbolic paraboloid	$ax^2 - by^2 - z = 0$	$X(u,v) = [\frac{u+v}{2\sqrt{a}}, \frac{u-v}{2\sqrt{b}}, uv]$
parabolic cylinder	$ax^2 - y = 0$	$X(u,v) = [u, au^2, v]$
hyperbolic cylinder	$ax^2 - by^2 = 1$	$X(u,v) = [\frac{1}{2\sqrt{a}}(u + \frac{1}{u}), \frac{1}{2\sqrt{b}}(u + \frac{1}{u}), v]$

In this new frame Q_R has normal form and admits one of the parameterizations $X_{R'}$ given in Table 1.5.

3. Let $Q_S \neq Q_R$. Enhance $X_{R'}$ by a fourth variable set to 1 and plug $X_{R'}$ into $S' = P^T S P$, which results in

$$X_{R'}(u,v)^T S' X_{R'}(u,v) = a(u) + b(u)v + c(u)v^2 = 0. \tag{1.36}$$

Since $X_{R'}$ is linear in u and v this equation is quadratic in both parameters. Hence, solving for v in terms of u yields:

$$v(u) = \frac{-b(u) \pm \sqrt{b(u)^2 - 4a(u)c(u)}}{2c(u)}. \tag{1.37}$$

4. Output the final parameterization of $S \cap T$ given by:

$$X_{S \cap T}(u) = P^T X_{R'}(u, v(u)), \tag{1.38}$$

where $\{u \in \mathbb{R} \mid b(u)^2 - 4a(u)c(u) \geq 0\}$ is the domain of $X_{S \cap T}$.

Levin's method is very nice and powerful since it gives an explicit representation of the intersection of two general quadrics. However, it is not feasible to be used as an exact approach and indeed it was never meant to be used as such. In particular, if performed in an exact way, step 1 (solving a polynomial of degree 3), and step 2 (computing normalized eigenvectors of R_u) introduce a tremendous and unacceptable amount of radicals.

1.5 Exact Parameterization of Intersection Curves

In this section we summarize the approach by Dupont *et al.* [23, 24, 25] computing an exact parameterization of two quadrics in $\mathbb{P}^3$ given by implicit equations with rational coefficients of arbitrary size. The algorithm identifies, separates and parameterizes all the algebraic components of the intersection in an exact and efficient way. It gives all the information on the incidence between the components, *i.e.* it reports where and how (e.g., tangentially or not) two components intersect. The algorithm is complete, *i.e.* it places no restriction of any kind on the input quadrics and the type of their intersection.

In case of a smooth quartic the parameterization involves the square root of a polynomial. In all other cases the parameterization is given in terms of rational functions. Compared to Levin's method the major advantage is the low algebraic complexity of the parameterization on the level of coefficients which makes the parameterization suitable for further exact treatment. In most cases the coefficients are rational or defined in an extension of algebraic degree 2. In a few cases the algebraic degree of the extension is 3 or 4. For a full overview see Table 1.7 at the end of this section.

Essentially the approach by Dupont *et al.* works in the same spirit as Levin's method. From a given pencil $R(\lambda) = \lambda S + T$, it chooses a 'nice' quadric R with a 'simple' parameterization. Then this parameterization is plugged into another quadric of the pencil in order to obtain the final parameterization of the intersection curve.

The fundamental differences to Levin's method are:

- The approach does not work over $\mathbb{R}^3$ but over $\mathbb{P}^3$. This has the advantage that there are more quadrics with a 'simple' parameterization, *i.e.* all quadrics with inertia different from $(3, 1)$ are suitable. An overview is given in Table 1.6.

- The approach uses Gauß[6] reduction to compute a rational transformation matrix P sending the chosen quadric R into the canonical form $R' = P^T R P$. This is a fundamental improvement, since it removes the radicals

[6] Johann C. F. Gauß (*April 30, 1777 in Braunschweig; †February 23, 1855 in Göttingen)

Table 1.6: Parameterization in $\mathbb{P}^3$ of quadrics with *inertia* $\neq (3,1)$. $\mathbb{P}^{\star 2}$ stands for the 2-dimensional weighted real projective space defined as the quotient of $\mathbb{R}^3 \setminus \{(0,0,0)\}$ with the equivalence relation $\sim$, where $(x,y,z) \sim (\lambda x, \lambda y, \lambda^2 z),\ \forall \lambda \in \mathbb{R} \setminus \{0\}$.

Inertia of S	canonical form $a,b,c,d>0$	parameterization X =[x,y,z,w]
(4,0)	$ax^2+by^2+cz^2+dw^2$	$Q_S = \emptyset$
(3,0)	$ax^2+by^2+cz^2$	Q_S is the point $(0,0,0,1)$
(2,2)	$ax^2+by^2-cz^2-dw^2$	$[\frac{ut+avs}{a}, \frac{us+bvt}{b}, \frac{ut-avs}{\sqrt{ac}}, \frac{us+bvt}{\sqrt{bd}}],\ (u,v),(s,t) \in \mathbb{P}^1$
(2,1)	$ax^2+by^2-cz^2$	$[uv, \frac{u^2-abv^2}{2b}, \frac{u^2+abv^2}{2\sqrt{bc}}, s],\ (u,v,s) \in \mathbb{P}^{\star 2}$
(2,0)	ax^2+by^2	$[0,0,u,v],\ (u,v) \in \mathbb{P}^1$
(1,1)	ax^2-by^2	$[u, \pm\frac{\sqrt{abb}}{u}, v, s],\ (u,v,s) \in \mathbb{P}^2$
(1,0)	ax^2	$[0,u,v,s],\ (u,v,s) \in \mathbb{P}^2$

introduced by the traditional eigenvalues/eigenvectors approach in step 2 of Levin's algorithm.

Apart from that, the algorithm by Dupont *et al.* splits into two parts. The first part treats the generic case, *i.e.* pencils with Segre-Symbol [1111]. The second part treats all degenerate cases. It is based on the study of the numerous possible real intersection types as discussed in Section 1.3.2.

1.5.1 Generic Case

In this case the first polynomial invariant is square-free and the intersection curve is a smooth quartic. The algorithm can briefly be summarized by the following steps, which we subsequently discuss in more detail.

For a given pencil $R(\lambda) = \lambda S + T$ with Segre-Symbol $\sigma_4 = [1111]$ do:

1. $\mathcal{D}(\lambda) = det(R(\lambda))$ has four simple roots. Use a root isolation algorithm, *e.g.* the Descartes Method, to compute isolating intervals for the real roots of $\mathcal{D}(\lambda)$. For each interval in between these roots compute a rational representative λ_i. For each λ_i such that $\mathcal{D}(\lambda_i) > 0$, compute the inertia

of $R(\lambda_i)$. If one interval corresponds to inertia $(4,0)$, report an empty intersection. Otherwise, proceed with $R^\star = R(\lambda_i)$. Note that $R^\star$ has inertia $(2,2)$.

2. Approximate a point on $Q_{R^\star}$, not in $S \cap T$, until the quadric $Q_R \in R(\lambda)$ which goes through the approximation of the point has inertia $(2,2)$. Hence, we know a rational point on Q_R. Use that point to compute a rational transformation matrix P for Q_R such that

$$R' = diag(1,1,-1,-\delta) = P^T R P, \text{ with } \delta \in \mathbb{Q}. \tag{1.39}$$

In this frame Q_R can be parameterized by

$$X_{R'} = \left[ut + vs, us + vt, ut - vs, \frac{(us+vt)}{\sqrt{\delta}}\right]^T,$$

where $(u,v),(s,t) \in \mathbb{P}^1$. Note that X_R is *bilinear*, *i.e.* linear in both parameters.

3. Let $Q_S \neq Q_R$. Consider the equation

$${X_{R'}}^T P^T S P X_{R'} = 0. \tag{1.40}$$

Since $X_{R'}$ is bilinear, this equation is quadratic in both parameters. Solve this equation for one of the parameters in terms of the other and compute the domain of the solution.

Output the solution in the original frame.

Details Step 1:

As in the first step of Levin's algorithm, Dupont *et al.* search for a ruled quadric within the pencil. In contrast to Levin's algorithm, they avoid to choose a quadric associated to an irrational number, *i.e.* a real root of $\mathcal{D}(\lambda)$.

Step 1 is justified by the following Theorem:

Theorem 6. *Let $R(\lambda) = \lambda S + T$ be a pencil with Segre-Symbol* [1111]. *Either $S \cap T$ is empty or the pencil contains a quadric $R(\lambda_0)$, $\lambda \in R$ of inertia $(2,2)$. Moreover, there is a neighborhood around λ_0 such that any other quadric associated to a value in that neighborhood has inertia $(2,2)$.*

Proof. By Theorem 5 (Levin) the intersection is either empty or contains a simple ruled quadric. Since $\mathcal{D}(\lambda)$ is square free and the set of simple ruled quadrics is positive semidefinite, this implies that $\mathcal{D}(\lambda) > 0,\ \forall\lambda \in \overline{\mathbb{R}}$ or $\mathcal{D}(\lambda)$ has at least one simple root. Hence, there is an interval $A = \{(a, b) \mid a < b \in \mathbb{R}\}$, such that $\mathcal{D}(\lambda) > 0,\ \forall\lambda \in A$. Moreover, the inertia within A is stable, since $\mathcal{D}(\lambda)$ is the constant term of $det(R(\lambda) - Ix)$ with respect to x. Thus for all $\lambda_0 \in A$, the inertia of the associated quadric $R(\lambda_0)$ is either $(4, 0)$ or $(2, 2)$. □

Details Step 2:

Though the ruled quadric $R^\star$ chosen in step 1 is rational, it is just an ordinary quadric of inertia $(2, 2)$. Hence, as one can see in Table 1.6, the coefficients of its parameterization would be defined in an algebraic extension of degree 2×2.

The clou in step 2 is to find a rational quadric R of inertia $(2, 2)$ with a known rational point p, which is achieved by construction. Given the point p it is easy to construct a rational line $\mathcal{L}$ through this point which is not tangent to Q_R. $\mathcal{L}$ intersects Q_R in a second point p' which is rational as well. Given these two points, compute a rational transformation matrix P' that sends p and p' to $(1, \pm 1, 0, 0)$. Thereafter, compute a second rational transformation matrix P'' using Gauß reduction. In the resulting frame Q_R has the form:

$$x^2 - y^2 + \alpha z^2 + \beta w^2 = 0,\ \alpha\beta < 0 \tag{1.41}$$

And finally

$$P''' = \begin{pmatrix} 1+\alpha & 0 & 1-\alpha & 0 \\ 1-\alpha & 0 & 1+\alpha & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2\alpha \end{pmatrix} \tag{1.42}$$

sends Q_R into the form (up to a constant factor):

$$x^2 + y^2 - z^2 - \delta w^2 = 0, \tag{1.43}$$

where $\delta = \alpha\beta$. Hence, the rational transformation matrix in step 2 is given by

$$P = P'P''P'''. \tag{1.44}$$

Details Step 3:

In the frame of Equation 1.43 the quadric Q_R is parameterized by[7]

$$X_{R'}(\xi,\tau) = \left[(\xi+\tau)\sqrt{\delta}, (\xi\tau-1)\sqrt{\delta}, (\xi-\tau)\sqrt{\delta}, \xi\tau+1\right]^T, \tag{1.45}$$

where $\xi = (u,v) \in \mathbb{P}^1$, $\tau = (s,t) \in \mathbb{P}^1$. This parameterization is plugged into the implicit equation of the quadric Q_S. This results in the polynomial

$$f(\xi,\tau) = X_{R'}(\xi,\tau)^T P^T S P X_{R'}(\xi,\tau) \in \mathbb{Q}(\sqrt{\delta})[\xi,\tau]. \tag{1.46}$$

The zero set of f represents the intersection curve of Q_S and Q_T in the parameter space of Q_R. Note that the polynomial f is biquadratic due to the fact that $X_{R'}(\xi,\tau)$ is bilinear. Therefore, this polynomial can be written in the form

$$f(\xi,\tau) = a_2(\xi)\tau^2 + a_1(\xi)\tau + a_0(\xi), \tag{1.47}$$

where each $(a_i)_{i=0,1,2} \in \mathbb{Q}(\sqrt{\delta})[\xi]$ is quadratic in ξ. Hence, it is easy to solve for τ. A possible parameterization for τ is:

$$\tau_\varepsilon(\xi) = \left(2a_0(\xi), -a_1(\xi) + \varepsilon\sqrt{\Delta(\xi)}\right), \tag{1.48}$$

where $\varepsilon = \pm 1$ and $\Delta(\xi) = a_1(\xi)^2 - 4a_0(\xi)a_2(\xi)$ is of degree 4. Substituting this within the parameterization $X_{R'}$ of Q_R, the final parameterization of the intersection curve of Q_S and Q_T is defined as:

$$X_\varepsilon(\xi) = P X_{R'}(\xi, \tau_\varepsilon(\xi)) \in [\mathbb{Q}(\sqrt{\delta})[\xi, \sqrt{\Delta}]]^4. \tag{1.49}$$

The parameterization consists of two arcs, a positive arc $X_{\varepsilon=1}$ and a negative arc $X_{\varepsilon=-1}$, each defined within the domain $\mathbb{D} = \{\xi \in \mathbb{P}^1 \mid \Delta(\xi) \geq 0\}$.

Breakdown of Parameterization

Though $\tau_\varepsilon(\xi)$ is a valid parameterization for most values of ξ there is a problem for roots of $a_0(\xi)$. Let ξ_0 be a root of $a_0(\xi)$ and consider $\tau_\varepsilon(\xi_0)$ as it is given

[7]Note that for the sake of simplicity, we have switched to the 'affine-like' notation.

in 1.48.

$$\begin{aligned}
\tau_\varepsilon(\xi_0) &= (2a_0(\xi_0), -a_1(\xi_0) + \varepsilon\sqrt{\Delta(\xi_0)}) && (1.50)\\
&= (0, -a_1(\xi_0) + \varepsilon\sqrt{a_1(\xi_0)^2}) && (1.51)\\
&= (0, -a_1(\xi_0) + \varepsilon|a_1(\xi_0)|) && (1.52)\\
&= (0, \varepsilon - sign(a_1(\xi_0))) && (1.53)
\end{aligned}$$

The parameterization becomes invalid at ξ_0 for $\varepsilon = sign(a_1(\xi_0))$.[8] Though this is a removable discontinuity of $\tau_\varepsilon(\xi)$ it is a source of numerical instability. Therefore, we always consider the alternative parameterization of τ as well, namely

$$\tilde{\tau}_\varepsilon(\xi) = \left(-a_1(\xi) - \varepsilon\sqrt{\Delta(\xi)}, 2a_2(\xi)\right). \tag{1.54}$$

Note that $\tau_\varepsilon(\xi)$ and $\tilde{\tau}_\varepsilon(\xi)$ are identical up to their removable discontinuities, which are at the real roots of $a_0(\xi)$ and $a_2(\xi)$, respectively.

Of course it is essential that at least one parameterization remains valid, that is, we have to guarantee that $a_0(\xi)$ and $a_2(\xi)$ have no common root. In general $a_0(\xi)$ and $a_2(\xi)$ have no common root and nothing needs to be done. However, in the rare case where $a_0(\xi)$ and $a_2(\xi)$ have a common root ξ_0, we can rely on the fact that $a_1(\xi_0) \neq 0$. Otherwise, this would imply a vertical line at ξ_0, which contradicts the fact that f represents a smooth quartic. Hence, it is easy to find a new frame for τ such that $a_0(\xi)$ and $a_2(\xi)$ have no common root.

1.5.2 Singular Cases

In case the first polynomial invariant is not square-free, further invariants are computed to give a complete case distinction as it has been discussed in Section 1.3.2. Beside the case of a nodal quartic or cuspidal quartic, the curve might decompose into several algebraic components, *e.g.* combinations of lines, conics and cubics[9], such that the accumulated algebraic degree never exceeds 4.

[8] This effect is even independent of ε if ξ_0 is a common root of $a_0(\xi)$ and $a_1(\xi)$.
[9] It may also consist of isolated points, the real parts of complex components.

Table 1.7: Maximal algebraic extension for the different algebraic components.

component	maximal algebraic degree of extension	degree for the entire intersection	optimality
smooth quartic	2	2	near-optimal
nodal quartic	2	2	near-optimal
cuspidal quartic	1	1	optimal
cubic	1	1	optimal
conic	4 (2×2)	4	near-optimal
line	4	24	optimal
point	4 (2×2)	4	optimal

We omit a full discussion of this part and refer to Dupont *et al.* [25], which is dedicated to the degenerate cases. However, in most cases the principal idea is:

- For a given pencil $R(\lambda) = \lambda S + T$ with principal invariant $\mathcal{D}(\lambda)$: Choose the 'most rational' singular quadric in the pencil. For instance, if $\mathcal{D}(\lambda) \not\equiv 0$, choose a quadric that is associated to a multiple root of $\mathcal{D}(\lambda)$.
- If there are known rational points on that quadric, *e.g.* the rational apex of another cone in the pencil, use this point and Gauß reduction to find a very simple canonical form of the chosen quadric.
- Parameterize the quadric using one of the parameterizations given in Table 1.6. Use the knowledge about the particular case to find the different algebraic components and parameterize them within that parameter space.

In all singular cases for each algebraic component the parameterization is given in terms of rational functions. However, in most cases the coefficients of the polynomial are defined in some algebraic extension. An overview is given in Table 1.7. Note that in some cases the parameterization is just near-optimal, *i.e.* the parameterization is optimal in the number of radicals up to one possibly unnecessary square root. According to Dupont *et al.* [25], this is the best one can hope for, *i.e.* deciding whether this extra square root can be avoided or not is hard. Moreover, they give examples in which the square root is really needed.

1.6 Intersection with another Quadric

In this section we discuss the most important step of our algorithm, namely the intersection of an algebraic component $\mathcal{C} \subseteq Q_S \cap Q_T$ with another quadric Q_U. If $\mathcal{C} \subset Q_U$ this should be detected and otherwise we wish to compute the exact parameter values of all real intersection points in $C \cap Q_U$ with respect to the parameterization $X_{\mathcal{C}}(\xi)$ of $\mathcal{C}$.

In those cases in which the parameterization is given in terms of rational functions the intersection with another quadric is straightforward. These cases are discussed all at once in Section 1.6.1. The remaining case, the smooth quartic, is a bit more involved due to the fact that the parameterization involves a square root of a polynomial. This case is discussed separately in Section 1.6.2.

Note that we do not compute the multiplicity associated to each intersection point, since it is not needed by the overall algorithm and causes some overhead in the computation. However, since the multiplicity may be of independent interest, we indicate in each case how to obtain the multiplicity as well. It will turn out that in most cases we gain the multiplicity for free.

1.6.1 Rational Parameterizations

For any algebraic component but the smooth quartic, we are in the comfortable situation that the parameterization is given in terms of rational functions. Moreover, the algorithm by Dupont *et al.* guarantees that the degree of the involved polynomials is minimal with respect to the parameterized algebraic component. Hence, the intersection with a third quadric is straightforward.

Given a rational parameterization $X_{\mathcal{C}}(\xi)$ of an algebraic component $\mathcal{C}$ and another Q_U we just plug $X_{\mathcal{C}}(\xi)$ into the implicit equation of Q_U. We obtain a univariate polynomial

$$h(\xi) = X_{\mathcal{C}}(\xi)^T U X_{\mathcal{C}}(\xi). \tag{1.55}$$

If $\mathcal{C} \subset Q_U$ this is the zero polynomial. Otherwise, the degree of $h(\xi)$ is 8, 6, 4 and 2 in case of singular quartics, cubics, conics and lines, respectively. This is minimal, since the degree of $X_{\mathcal{C}}(\xi)$ is minimal as well. Hence, each real root ξ_i of $h(\xi)$ and its multiplicity m_i exactly corresponds to one intersection point

in $\mathcal{C} \cap Q_U$. The only minor exception is the nodal quartic with a non-isolated singular point. In this case the parameterization passes the points twice, once for each arc passing the singular point. However, this is in fact an advantage since an intersection at this point is computed separately for each arc. In particular the multiplicity is reported separately with respect to each arc.

In case of singular quartics, conics and cubics we use Algorithm 1 in order to compute the intersection with a third quadric. Note that the used coefficient type depends on the algebraic extension in which the parameterization $X_{\mathcal{C}}(\xi)$ is given. This is either an algebraic extension of degree 1, 2 or 2×2, see also Table 1.7 in Section 1.5. However, this is no problem since for all these cases we can use a proper instantiation of the type `NiX::Sqrt_extension` in order to represent the coefficients. For more details on implementation issues see also Section 1.9.2.

Algorithm 1 Let $\mathcal{C} \subseteq Q_S \cap Q_T$ be a singular quartic, cubic or conic and $X_{\mathcal{C}}(\xi)$ its rational parameterization. Given another quadric Q_U, if $\mathcal{C} \subset Q_U$ report $\mathcal{C}$. Otherwise, compute the exact parameter values of all real intersection points $\mathcal{C} \cap Q_U$ with respect to $X_{\mathcal{C}}(\xi)$.

(1) $h(\xi) := {X_{\mathcal{C}}}^T(\xi) U X_{\mathcal{C}}(\xi)$
(2) **if** $\{\ h(\xi) \equiv 0\ \}$ report $\mathcal{C}$ and return. **end if**
(3) $\mathcal{F} := square\_free\_fac(h)$ // `of the form` $\{(fac_0, m_0), \ldots, (fac_k, m_k)\}$
(4) **for all** $\{(fac_i, m_i) \in \mathcal{F}\ \}$ **do**
(5) compute and report all roots of fac_i
(6) **end for**

In the case of lines the algebraic extension of the coefficients can be 1, 2 and 2×2 as well. But it may also happen, that the algebraic extension is of degree 3 or 4. In these cases we use `LEDA::real` [62] or `CORE::Expr` [52] to represent the coefficients of the lines. Algorithm 2 is used for lines only and designed such that it is possible to use `LEDA::real` or `CORE::Expr` as coefficient type as well. Note that the worst operations are the sign computation and the square root of the discriminant of $h(\xi)$.

Algorithm 2 Let $\mathcal{L} \subseteq Q_S \cap Q_T$ be a line and $X_{\mathcal{L}}(\xi)$ its rational parameterization. Given another quadric Q_U, if $\mathcal{L} \subset Q_U$ report $\mathcal{L}$. Otherwise, compute the exact parameter values of all real intersection points $\mathcal{L} \cap Q_U$ with respect to $X_{\mathcal{L}}(\xi)$.

(1) $h(\xi) := {X_R}^T(\xi) U X_R(\xi) = a\xi^2 + b\xi + c$
(2) **if** { $h(\xi) \equiv 0$ } report $\mathcal{L}$ and return. **end if**
(3) $discr := b^2 - 4ac$
(4) **if** { $discr < 0$ } Report the empty intersection and return. **end if**
(5) **if** { $discr = 0$ } Report $\xi = -b/2a$ and return. **end if**
(6) **if** { $discr > 0$ } Report $\xi_\pm = -b \pm \sqrt{discr}/2a$ and return. **end if**

1.6.2 Smooth Quartic

In case of a smooth quartic $\mathcal{C}_{S\cap T} = Q_S \cap Q_T$ the situation is a bit complicated due to the fact that the parameterization involves the square root of a polynomial. Therefore, the parameterization splits into two arcs, $X_+(\xi)$ and $X_-(\xi)$. This has the consequence that a point on $\mathcal{C}_{S\cap T}$ has to be identified by its corresponding value for ξ but also by the arc it lies on.

Recall that the parameterization of $\mathcal{C}$ was obtained within the parameter space of a ruled quadric $Q_R \in pencil(Q_S, Q_T)$, *w.l.o.g.* $Q_R \neq Q_S$. Let $X_R(\xi, \tau)$ be the parameterization of Q_R. In this parameter space $Q_S \cap Q_T$ is defined by the zero set of the biquadratic polynomial

$$\begin{aligned} f(\xi,\tau) &= {X_R(\xi,\tau)}^T S X_R(\xi,\tau) && (1.56)\\ &= a_2(\xi)\tau^2 + a_1(\xi)\tau + a_0(\xi), && (1.57)\end{aligned}$$

where $(a_i)_{i=0,1,2} \in \mathbb{Q}(\sqrt{\delta})[\xi]$ are of degree 2. Hence, the parameterization is given by

$$\tau_\varepsilon(\xi) = \left(2a_0(\xi)\ ,\ -a_1(\xi) + \varepsilon\sqrt{\Delta(\xi)}\right), \qquad (1.58)$$

where $\varepsilon \in \{\pm 1\}$ and $\Delta(\xi) = a_1(\xi)^2 - 4a_0(\xi)a_2(\xi) \in \mathbb{Q}(\sqrt{\delta})[\xi]$, see also Section 1.5. Subsequently, we will compute the intersection of $\mathcal{C} \cap Q_U$ within the parameter space of Q_R using the fact that

$$Q_S \cap Q_T \cap Q_U = Q_S \cap Q_R \cap Q_U = (Q_R \cap Q_S) \cap (Q_R \cap Q_U). \qquad (1.59)$$

In the parameter space of Q_R the intersection $Q_R \cap Q_U$ is given by the zero set of

$$\begin{aligned} g(\xi,\tau) &= X_R(\xi,\tau)^T U X_R(\xi,\tau) && (1.60)\\ &= b_2(\xi)\tau^2 + b_1(\xi)\tau + b_0(\xi), && (1.61)\end{aligned}$$

where $(b_i)_{i=0,1,2} \in \mathbb{Q}(\sqrt{\delta})[\xi]$. Since we want to compute the parameter values of $\mathcal{C}_{S\cap T} \cap Q_U$, we are first of all interested in the ξ-coordinates of the common solutions of f and g. Hence, we use a classical resultant approach to eliminate τ. The resultant of f and g with respect to τ is given by:

$$\begin{aligned} res(\xi) &= resultant(f,g,\tau) && (1.62)\\ &= s_{02}(\xi)^2 - s_{01}(\xi)s_{12}(\xi), && (1.63)\end{aligned}$$

where $s_{ij} = a_i b_j - a_j b_i \in \mathbb{Q}(\sqrt{\delta})[\xi]$ are the relevant minors of the Sylvester matrix.

Proposition 7. *Let f, g and res be defined as in Equations (1.57), (1.61) and (1.63) respectively. $Q_R \cap Q_S = Q_R \cap Q_U$ iff $res \equiv 0$.*

Proof. First of all it is clear that $Q_R \cap Q_S = Q_R \cap Q_U$ implies $res \equiv 0$. Now, note that f and g are both biquadratic and that f is irreducible since it is representing a smooth quadric. Moreover, $res \equiv 0$ implies that f and g have a common factor of positive degree. Hence, f and g are equal up to a constant factor. □

In general, *i.e.* $\mathcal{C} \not\subset Q_U$, the resultant res is a polynomial of degree 8. By Bezout's Theorem the roots of res are the ξ-coordinates of the intersection points of f and g, multiplicities counted. It remains to discard the complex intersection points and to determine the correct arc for the real intersection points. This is embodied in Theorem 8.

Theorem 8. *Let f, τ_ε, g and $res \neq 0$ be defined as in Equations (1.57), (1.58), (1.61) and (1.63) respectively. And let ξ_0 denote a real root (if any) of res. Moreover, let $\tau_\varepsilon(\xi)$ be a valid parameterization for ξ_0, that is, ξ_0 is not a root of $a_0(\xi)$.*

There are 3 cases:

1. $\Delta(\xi_0) < 0$*: ξ_0 corresponds to two complex intersection points.*
2. $\Delta(\xi_0) = 0$*: ξ_0 corresponds to a real endpoint of both arcs.*

3. $\Delta(\xi_0) > 0$*:*

- *If* $s_{01}(\xi_0) \neq 0$*, then* ξ_0 *corresponds to one real intersection point on arc* $X_\varepsilon(\xi)$*, with* $\varepsilon = -sign(s_{01})sign(2a_0s_{02} - a_1s_{01})_{|\xi_0}$.

- *If* $s_{01}(\xi_0) = 0$*, then* ξ_0 *corresponds to two real points, one on each arc.*

Proof. $f_{|\xi=\xi_0}(\tau)$ is a quadratic polynomial in τ and $\Delta(\xi_0)$ is its discriminant. Since this proves the first two statements consider the third. Due to the fact that $\Delta(\xi_0)$ is positive there are two possible solutions, namely $(\xi_0, \tau_{\varepsilon=+1}(\xi_0))$ and $(\xi_0, \tau_{\varepsilon=-1}(\xi_0))$. First of all, it is clear that at least one of these two possibilities is a valid solution. Now observe that $g(\xi, \tau_\varepsilon(\xi))_{|\xi_0}$ is independent of ε iff $s_{01}(\xi_0) = 0$, since $g(\xi, \tau_\varepsilon(\xi))$ can be written as:

$$\begin{aligned} g(\xi, \tau_\varepsilon(\xi)) &= b_2\tau_\varepsilon^2 + b_1\tau_\varepsilon + b_0 \\ &= b_2(2a_0)^2 + b_1(2a_0)(-a_1 + \varepsilon\sqrt{\Delta}) + b_0(-a_1 + \varepsilon\sqrt{\Delta})^2 \\ &= 4a_0^2b_2 - 2a_0a_1b_1 + 2a_0b_1\varepsilon\sqrt{\Delta} + a_1^2b_0 - 2a_1b_0\varepsilon\sqrt{\Delta} + a_1^2b_0 - 4a_0a_2b_0 \\ &= 4a_0(a_0b_2 - a_2b_0) - 2a_1(a_0b_1 - a_1b_0) + 2(a_0b_1 - a_1b_0)\varepsilon\sqrt{\Delta}) \\ &= (4a_0s_{02} - 2a_1s_{01}) + \varepsilon(2s_{01}\sqrt{\Delta}). \end{aligned}$$

From this it follows that both possible solutions are valid if $s_{01}(\xi_0) = 0$. Otherwise, there is only one solution and we have to choose ε such that the expressions $(4a_0s_{02} - 2a_1s_{01}) \neq 0$ and $\varepsilon(2s_{01}\sqrt{\Delta}) \neq 0$ have opposite sign. □

Remark 1. *In case* $\tau_\varepsilon(\xi)$ *is not a valid parameterization for* ξ_0 *a symmetric consideration for* $\tilde{\tau}_\varepsilon(\xi)$ *leads to:*

$$\begin{aligned} g(\xi, \tilde{\tau}_\varepsilon(\xi)) &= b_2\tilde{\tau}_\varepsilon^2 + b_1\tilde{\tau}_\varepsilon + b_0 \\ &= b_2(-a_1 - \varepsilon\sqrt{\Delta})^2 + b_1(-a_1 - \varepsilon\sqrt{\Delta})(2a_2) + b_0(2a_2)^2 \\ &= (2a_1s_{12} - 4a_2s_{02}) + \varepsilon(2s_{12}\sqrt{\Delta}) \end{aligned}$$

Hence, in this case there are two real points if $s_{12}(\xi_0) = 0$*. Otherwise,* ε *is chosen such that* $(a_1s_{12} - 2a_2s_{02})$ *and* (εs_{12}) *have opposite sign. Note that at least one parameterization remains valid, see also Section 1.5.*

Algorithm

We are now ready to state Algorithm 3 intersecting a smooth quartic $\mathcal{C}$ with a third quadric Q_U. Note that in general all operations are performed over an algebraic extensions of degree 2, which in particular complicates an efficient implementation of the square free factorization (line 9), the root isolation (line 11) and the 'is-root-of' tests (line 15, 19, 21). However, the algorithm is designed such that all sign computations (line 17, 22, 25) are known to have results different from zero. Hence, we can use multiprecision floating point interval arithmetic (MPFI) in order to compute the signs. We start with a low precision and just double the precision of the floating point arithmetic until an unambiguous sign is computed. All required tools are provided by the library NumeriX, which is part of the Exacus project, see also Section 1.9.2.

Algorithm 3 Given the parameterization $X_{\mathcal{C}}(\xi)$ of a smooth quartic $\mathcal{C} = Q_S \cap Q_T$ and another quadric Q_U. If $\mathcal{C} \subset Q_U$ report $\mathcal{C}$. Otherwise, compute the exact parameter values of all real intersection points $\mathcal{C} \cap Q_U$ with respect to $X_{\mathcal{C}}(\xi)$.

// All coefficients are defined in $\mathbb{Z}$ or $\mathbb{Z}[\sqrt{\delta}],\ \delta \in \mathbb{Z}$

(1) $f(\xi,\tau) := {X_R}^T(\xi,\tau) S X_R(\xi,\tau) = a_2(\xi)\tau^2 + a_1(\xi)\tau + a_0(\xi)$
(2) $g(\xi,\tau) := {X_R}^T(\xi,\tau) U X_R(\xi,\tau) = b_2(\xi)\tau^2 + b_1(\xi)\tau + b_0(\xi)$
(3) $\Delta(\xi) := a_1(\xi)a_1(\xi) - 4a_0(\xi)a_2(\xi)$
(4) $s_{01}(\xi) := a_0(\xi)b_1(\xi) - a_1(\xi)b_0(\xi)$
(5) $s_{02}(\xi) := a_0(\xi)b_2(\xi) - a_2(\xi)b_0(\xi)$
(6) $s_{12}(\xi) := a_1(\xi)b_2(\xi) - a_2(\xi)b_1(\xi)$
(7) $res(\xi) := s_{02}(\xi)s_{02}(\xi) - s_{01}(\xi)s_{12}(\xi)$

(8) **if** { $res \equiv 0$ } report $\mathcal{C}$ and return. **end if**

(9) $\mathcal{F} := square\_free\_fac(res)$ // of the form $\{(fac_0, m_0), \ldots, (fac_k, m_k)\}$
(10) **for all** $\{(fac_i, m_i) \in \mathcal{F}\}$ **do**
(11) isolate all roots of fac_i // e.g. the Descartes Method
(12) store all these roots together with multiplicity m_i in $\mathcal{R}$.
(13) **end for**

(14) **for all** $\{(\xi_i, m_i) \in \mathcal{R}\}$ **do**
(15) **if** { ξ_0 is root of $\Delta(\xi)$ }
(16) Report $(\xi_i, 0)$ and continue with next root.
(17) **else if** { $m_i > 0$ and $sign(\Delta(\xi_0)) < 0$ } // use MPFI
(18) Reject ξ_i and continue with next root.
(19) **else if** { $m_i > 0$ and ξ_0 is root of s_{01} and s_{12} }
(20) Report $(\xi_i, +1)$ and $(\xi_i, -1)$ and continue with next root.
(21) **else if** { ξ_0 is not a root of a_0 }
(22) $\varepsilon := -sign(s_{01})sign(2a_0s_{02} - a_1s_{01})_{|\xi_0}$ // use MPFI
(23) Report (ξ_i, ε) and continue with next root.
(24) **else** // ξ_0 is not a root of a_2
(25) $\varepsilon := -sign(s_{12})sign(a_1s_{12} - 2a_2s_{02})_{|\xi_0}$ // use MPFI
(26) Report (ξ_i, ε) and continue with next root.
(27) **end if**
(28) **end for**

Multiplicities

As discussed in Section 1.1.2 our overall algorithm does not require the multiplicity of the intersection points. Hence, Theorem 8 omits statements about the multiplicity of the intersection points. The following Theorem is given in order to close this gap and may be seen as an addendum to Theorem 8.

Theorem 9. *Let f, τ_ε, g and $res \neq 0$ be defined as in Equations (1.57), (1.58), (1.61) and (1.63) respectively. Let ξ_0 denote a real root of the resultant such that $\Delta(\xi_0) > 0$ and let τ_ε be a valid parameterization for ξ_0. Moreover, let m_0 denote the multiplicity of ξ_0 and let $i_c > 0$ denote the smallest integer such that $s_{01}^{(i_c)}(\xi_0) \neq 0$. Then:*

1. *There are two real intersection points corresponding to ξ_0 with multiplicity i_c and $m_0 - i_c$ respectively.*
2. *If $i_c \neq m_0 - i_c$, the point with multiplicity $m_0 - i_c$ is the one on arc $X_\varepsilon(\xi)$, where $\varepsilon = -sign(s_{01}^{(i_c)})sign((2a_0s_{02} - a_1s_{01})^{(i_c)})|_{\xi_0}$.*

Proof. Observe that $\forall i \in \{i \in \mathbb{N} \mid i < i_c\}$ the i-th derivative of $g(\xi, \tau_\varepsilon(\xi))$

$$\begin{aligned} g(\xi, \tau_\varepsilon(\xi))^{(i)} &= (4a_0s_{02} - 2a_1s_{01} + 2\varepsilon s_{01}\sqrt{\Delta})^{(i)} \\ &= (4a_0s_{02} - 2a_1s_{01})^{(i)} + (2\varepsilon s_{01}\sqrt{\Delta})^{(i)} \\ &= (4a_0s_{02} - 2a_1s_{01})^{(i)} + 2\varepsilon s_{01}^{(i)}\sqrt{\Delta} + \cdots + 2\varepsilon s_{01}(\sqrt{\Delta})^{(i)} \end{aligned}$$

evaluated at ξ_0 is independent of ε. That is, both points have at least multiplicity i_c. If $i_c = m_0 - i_c$ it is clear that both points have exactly multiplicity i_c. Otherwise, consider $g(\xi, \tau_\varepsilon(\xi))^{(i_c)}$ evaluated at ξ_0:

$$g(\xi_0, \tau_\varepsilon(\xi_0))^{(i_c)} = (4a_0s_{02} - 2a_1s_{01})^{(i_c)}(\xi_0) + 2\varepsilon s_{01}^{(i_c)}(\xi_0)\sqrt{\Delta(\xi_0)} + 0$$

This is not independent from ε. And since $i_c \neq m_0 - i_c$ it is clear that

$$g(\xi_0, \tau_\varepsilon(\xi_0))^{(i_c)} = 0$$

has exactly one solution, namely

$$\varepsilon = -sign(s_{01}^{(i_c)})sign((2a_0s_{02} - a_1s_{01})^{(i_c)})|_{\xi_0}$$

□

Remark 2. *In case $\tau_\varepsilon(\xi)$ is not a valid parameterization for ξ_0 a symmetric consideration for $\tilde{\tau}_\varepsilon(\xi)$ leads to:*

$$g(\xi_0, \tilde{\tau}_\varepsilon(\xi_0))^{(i_c)} \quad = \quad (2a_1 s_{12} - 4a_2 s_{02})^{(i_c)}(\xi_0) + 2\varepsilon(s_{12}^{(i_c)}(\xi_0)\sqrt{\Delta(\xi_0)})$$

Hence, if $m_0 - i_c \neq i_c$ the point with multiplicity $m_0 - i_c$ is the one on arc $X_\varepsilon(\xi)$, where $\varepsilon = -sign((2a_1 s_{12} - 4a_2 s_{02})^{(i_c)})sign(s_{12}^{(i_c)})|_{\xi_0}$. Note that at least one parameterization remains valid, see also Section 1.5.

1.7 Compare Points

We will now focus on how to compare two intersection points, as they have been defined in section 1.6. This problem actually consists of two parts.

- The first part is how to compare points which are defined on the same algebraic component, that is, how to sort points along a common algebraic component. This is discussed in Section 1.7.1.
- The second part comes from the fact that an intersection point in general has several representations, one for each algebraic component it lies on. The task is to match these representations in an efficient way. Our solution is presented in Section 1.7.2.

1.7.1 Sorting Points on Algebraic Components

The representation of the intersection points by their parameter value has the fundamental advantage that it is very easy to sort them along a common component.

In principal we just have to compare real roots of univariate polynomials, which is provided by the library NumeriX. In almost all cases the domain of the parameterization is $\mathbb{P}^1$. This induces a cyclic order along the component. Hence, the domain is broken up at $(1,0) \in \mathbb{P}^1$ yielding a total order along the component, *i.e.* the one induced by $\mathbb{R}^1$. The only two cases that need a bit more attention are the nodal quartic and the smooth quartic.

Sorting on Nodal Quartic

As in the other cases the parameterization $X_{\mathcal{C}}$ of a nodal quartic $\mathcal{C}$ is given in terms of rational functions and the domain of definition is $\mathbb{P}^1$. However, due to the singular point the situation becomes a bit more involved. There are two cases:

1. The singular point is an isolated point:
   In this case the parameterization does not reach the singular point at all. Therefore, the point is represented by a separate value and excluded from the normal sorting process.
2. The singular point is not isolated:
   Since there are two arcs passing through the singular point there are two parameter values[10] representing this point. Removing one of these values from the domain makes it isomorphic to $\mathbb{R}^1$ and hence capable for sorting. We just have to keep track of this artifact in the final data structure.

Sorting on Smooth Quartic

First of all recall that the parameterization consists of two arcs, a positive arc $X_{\varepsilon=1}$ and a negative arc $X_{\varepsilon=-1}$. Each arc is defined within the domain

$$\mathbb{D} = \{\xi \in \mathbb{P}^1 \mid \Delta(\xi) \geq 0\}, \tag{1.64}$$

where $\Delta(\xi)$ is a univariate square free polynomial of degree 4. Hence, the number of possible real roots of $\Delta(\xi)$ is 0, 2 or 4. This induces 3 different types of smooth quartic curves, according to the possible real roots of Δ, as illustrated in Figure 1.1. In the sequel we discuss the sorting according to the number of real roots of Δ:

0 In this case $X_{\varepsilon=1}(\xi)$ and $X_{\varepsilon=-1}(\xi)$ never touch. There are two connected components defined on $\mathbb{P}^1$, each formed by one of the arcs. The points on the different arcs are treated separately.

2 In this case the arcs touch at the two roots of Δ and form one connected component. We define a cyclic order on the component by reversing the order on the negative arc. The cyclic order is broken up at the first root of Δ.

4 In this case the domain is broken up into two intervals. The two arcs are connected at the endpoints, *i.e.* the roots of Δ. Hence, there are two connected components treated separately, the sorting is performed as in the case of one connected component.

[10] Defined by a known quadratic polynomial used in the parameterization, see Dupont *et al.* [25].

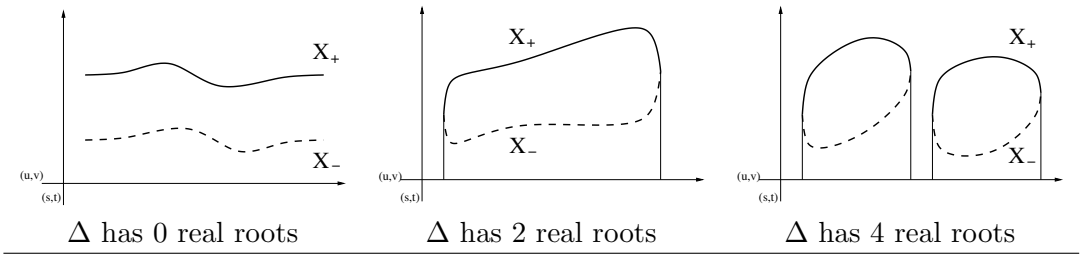


Figure 1.1: Different types of smooth quartics illustrated in parameter space.

1.7.2 Matching of Intersection Points

The representation of the intersection points by their parameter value has the fundamental advantage that it is very easy to sort the intersection points with respect to a common component. However, the disadvantage is that for each point we have to keep one representation for each component it lies on. Moreover, it must be possible to identify representations representing the same intersection point, which is the focus of this section.

First of all it should be clear that given just two representations on different components, it is very hard to decide whether they represent the same intersection point or not, that is, it is hard to detect equality. This is due to the fact that we are forced to compare the points by their coordinates in $\mathbb{P}^3$, since the representations are not given in the same parameter space. Now observe that the degree of the involved algebraic expressions is tremendous. In the worst case, that is, each point is defined on another smooth quartic, each coordinate involves, among others, the evaluation of a square root of a polynomial of degree 4 evaluated at a parameter value defined as the real root of a polynomial of degree 8 and all this over an algebraic extension of degree 2. Thus, to make the long story short, it is at least very inefficient to compare two representations via their exact coordinates in $\mathbb{P}^3$. And this holds for an explicit algebraic approach as well as for an approach based on root bounds, *e.g.* provided by types as `LEDA::real` [62] or `CORE::Expr` [52].

On the other hand it is very easy to detect inequality of two points using interval arithmetic based on multiprecision floats (MPFI). The procedure is as follows:

Start with a low precision and compute the coordinates of each point using MPFI. By the inclusion property of MPFI this results in two bounding boxes, one for each point. If the boxes still overlap, double the precision and start again. In most cases the two points are far apart and the process terminates after a few rounds. Note that the process does not terminate if the points are equal. However, in case of inequality this is fast and leads to the following idea.

Given a representation of a point p and a sequence of representations seq. If we know that p has exactly one corresponding representation in seq we can find this representation very fast using MPFI arithmetic as discussed above. We just increase the precision until we can exclude all but one representation in seq, which must be the one we are looking for. This idea is comprised in Algorithm 4.

Algorithm 4 Given the sequences $seq_1 = \{p_1, \ldots, p_m\}$ and $seq_2 = \{q_1, \ldots, q_n\}$ representing the point sets $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively. Given the fact that $\mathcal{S}_1 \subseteq \mathcal{S}_2$ compute the injective map φ_{12} matching the representations in seq_1 with their counterparts in seq_2.

```
// start with the empty map
φ12 := {}
// number of used bits in all MPFI computations
mpfi_precision := 2
while {seq1 ≠ ∅ } do
    for all points in seq1 and seq2 do
        precompute a bounding box using MPFI
    end for
    if a BBOX(pi) intersects only one BBOX(qj) then
        add (i, j) to φ12
        remove pi from seq1
        remove qj from seq2
    end if
    // double the precision for MPFI computations
    mpfi_precision := 2 · mpfi_precision
end while
Report φ12 and return.
```

Note that Algorithm 4 is very efficient since it is using MPFI arithmetic in an adaptive way. We consider it as one of the main sources of efficiency in our approach. In fact the overall algorithm, see Section 1.1.2, is designed such that it is possible to apply Algorithm 4. However, its Achilles' heel is that it can not effectively check the precondition, namely that S_1 is contained in S_2 and that the points in each set are in fact distinct. In many cases a violation of the precondition is not detected at all. Instead the algorithm just produces some meaningless mapping φ_{12} or does not terminate at all.

1.8 Unify Equal Algebraic Components

All intersection points are represented by the parameter values with respect to the algebraic components they lie on. Hence, it is very important to guarantee a unique parameterization of each algebraic component that appears in the data structure. Unfortunately, it is not possible to avoid unnecessary constructions of algebraic components, since a component may be defined by several pencils. A conic, for instance, may be contained in the intersection of three linear independent quadrics. Moreover, the algorithm by Dupont *et al.* [23, 24, 25, 58] can not guarantee a unique parameterization of algebraic components by construction. This has the consequence that before we can start to intersect algebraic components with other quadrics, we have to ensure that within the data structure the same algebraic component is not represented more than once.

First of all we avoid a lot of unnecessary construction using a cache. Given two quadrics Q_S and Q_T, we construct a unique representation of the pencil defined by Q_S and Q_T. This pencil is used as the key for the cache. Moreover, the cache stores the result of the intersection of the quadrics Q_S and Q_T. Note that due to the fact that a quartic is unambiguously defined by its pencil and vice versa the cache avoids reconstructions for all quartics, *i.e.* smooth quartics, nodal quartics and cuspidal quartics. However, for the other algebraic component types the cache is not sufficient in order to avoid unnecessary constructions.

For the other algebraic component types, *i.e.* cubics, conics and lines we in principal proceed as follows. For a new parameterization $X'_{\mathcal{C}}$ of $\mathcal{C}$ we check whether there exists an old parameterization $X_{\mathcal{C}}$ of $\mathcal{C}$. If this is the case, we replace the new parameterization $X'_{\mathcal{C}}$ by the old parameterization $X_{\mathcal{C}}$. But before we can do this, we have to 'rescue' the points defined on $X'_{\mathcal{C}}$ and redefine them with respect to $X_{\mathcal{C}}$. Therefore, the overall algorithm has been designed such that this is only needed in a very early stage, namely right after the construction of an algebraic component. This has the advantage that the only points defined on $X'_{\mathcal{C}}$ are the intersection points with other components in the pencil of $X'_{\mathcal{C}}$.

The comparison of algebraic components is discussed in Section 1.8.1. The mapping of the singular points is discussed in Section 1.8.2.

1.8.1 Comparing Algebraic Components

Even though the cache avoids a second construction of quartics, we discuss the comparison for all algebraic component types. Moreover, note that the cache is never used as a precondition.

- **Quartics:**
  For quartics, as well as for cubics and conics, the comparison is based on the construction history of each algebraic component, that is, each algebraic component stores at least the pencil it has been constructed from. Due to the fact that a quartic unambiguously defines its pencil it suffices to test the equality of the pencil.

- **Cubics:**
  In the case of cubics we use the fact that each pencil contains up to one cubic only. Hence, it is sufficient to test whether the first cubic is contained in the pencil of the second cubic. To test this the parameterization of the first cubic is plugged into the implicit equations of the two defining quadrics in the pencil of the second cubic. The cubics are equal if both resulting polynomials vanish.

- **Conics**
  For conics the situation gets a bit more involved due to the fact that a pencil may contain up to two regular conics. However, if one of the pencils contains only one regular conic we proceed as in the case of cubics.

  It remains to discuss the case that both pencils contain two regular conics. Let $\mathcal{C}$ and $\mathcal{C}'$ be the two compared conics and $\mathcal{P} = \mathit{pencil}(Q_S, Q_T)$ and $\mathcal{P}' = \mathit{pencil}(Q_{S'}, Q_{T'})$ be their defining pencils, respectively.

  - If $\mathcal{P} \neq \mathcal{P}'$ we know that both pencils contain up to one common conic only. This is due to the fact that two conics unambiguously define a pencil. Hence, the two conics are equal if $\mathcal{C} \subset Q_{S'} \cap Q_{T'}$ and $\mathcal{C}' \subset Q_S \cap Q_T$. This is tested in the same way as in the case of cubics.

  - If $\mathcal{P} = \mathcal{P}'$ the pencils have Segre-Symbol [11(11)] or [1(21)], see Table 1.2.[11] In both cases the algorithm by Dupont *et al.* [25] will choose

[11] The double conic in a pencil with Segre-Symbol [1(111)] is considered as one conic.

the only quadric of rank 2 in order to parameterize the conics. And since the rest of the algorithm is deterministic it will always produce exactly the same parameterization for the conics. Hence, comparing the conics in this case is trivial.

Lines

In the case of lines, we have to compare lines defined in an algebraic extension of similar degree only, since the degree is guaranteed to be optimal (see Table 1.7). In case of degree one and two, we compare the lines via Plücker Coordinates [75] using explicit arithmetic[12]. In case of a degree 4 extension we can guarantee a unique representation of the line by construction due to the fact that the pencil is uniquely defined by the line and its three algebraic conjugates. In the case of lines defined in a degree 3 extension the situation is more involved, since the pencil is not unique due to the fourth rational line.

Lines in an Algebraic Extension of Degree 3

First of all note that this is a very rare case. The only case in which such a line can appear is the one with Segre-Symbol [111] and only if exactly one of the other four lines in this pencil is rational. Due to the low relevance of this case we abstained from implementing a very efficient method. Instead, we currently just use Plücker Coordinates with `leda::real` or `CORE::Expr` as coefficient type. Note that this is quite inefficient in case of equality due to the fact that the involved numbers have to be refined until the separation bound is reached.

Hence, for a mature implementation we propose the following approach. Let $\mathcal{L}$ and $\mathcal{L}'$ be the two lines to be compared. Moreover, let $\mathcal{P} = pencil(Q_S, Q_T)$ and $\mathcal{P}' = pencil(Q_{S'}, Q_{T'})$ be the defining pencils of $\mathcal{L}$ and $\mathcal{L}'$ respectively. First of all note that $\mathcal{L}$ is one of three algebraic conjugated lines in the pencil $\mathcal{P}$. The fourth line in $\mathcal{P}$ is rational. All lines intersect in one real rational point p, namely the singular locus of $\mathcal{P}$. Since $\mathcal{L}$ is not rational this is the only rational point of $\mathcal{L}$. Of course the same holds for $\mathcal{L}'$ and its rational point p'. Hence, $p \neq p'$ implies that $\mathcal{L} \neq \mathcal{L}'$.

[12]For instance, `leda::rational` or `NiX::Sqrt_extension` (Section 1.9).

From now on assume that $p = p'$. Construct a rational plane $\mathcal{H}$ not containing p. Thereafter, compute the conics $\mathcal{C}_S = \mathcal{H} \cap Q_S$ and $\mathcal{C}_T = \mathcal{H} \cap Q_T$. Use the ConiX library [7] to compute their intersection $\mathcal{S} = \mathcal{C}_S \cap \mathcal{C}_T$. Each point in $\mathcal{S}$ corresponds to one intersection of a line in $\mathcal{P}$ with $\mathcal{H}$. Use Algorithm 4 to identify the point $p_{\mathcal{L}} \in \mathcal{S}$ corresponding to $\mathcal{L}$. In the same way compute $p_{\mathcal{L}'}$ corresponding to $\mathcal{L}'$. The lines are equal iff $p_{\mathcal{L}} = p_{\mathcal{L}'}$.

1.8.2 Redefinition of Points on $X'_{\mathcal{C}}$

Let $\mathcal{C}$ be a cubic, conic or line that has just been constructed a second time. Let $X'_{\mathcal{C}}$ be the new and $X_{\mathcal{C}}$ the old parameterization of $\mathcal{C}$ respectively. Since we want to delete $X'_{\mathcal{C}}$ we first of all have to redefine the points on $X'_{\mathcal{C}}$ with respect to $X_{\mathcal{C}}$.

In general it is not trivial to provide a function that just maps the parameter values of points on $X'_{\mathcal{C}}$ to the corresponding values on $X_{\mathcal{C}}$. This is hindered by two reasons. First, in case of conics and cubics the mapping is not linear. Second, $X'_{\mathcal{C}}$ and $X_{\mathcal{C}}$ may be given in two different algebraic extensions. Therefore, we decided to follow a general approach which is applicable to all but a very few special cases which just involve rational lines or rational points. To begin with, we start with these two special cases. Thereafter, we identify the remaining cases and discuss the general approach.

In the sequel let $\mathcal{P}'$ be the pencil that led to the construction of $X'_{\mathcal{C}}$. Moreover, let $\mathcal{S}$ be the set of m points on $\mathcal{C}$ that have been induced by $\mathcal{P}'$ and let $seq' = \{p'_1, .., p'_m\}$ be the sequence of representations for these points with respect to $X'_{\mathcal{C}}$.

Rational Line

Let $\mathcal{C}$ be a rational line. It is clear that $X_{\mathcal{C}}$ and $X'_{\mathcal{C}}$ are rational as well. Let the new parameterizations of $\mathcal{C}$ be given by

$$X'_{\mathcal{C}}(\lambda, \mu) \;=\; \lambda p' + \mu q', \quad \text{with } p', q' \in \mathbb{P}^3(\mathbb{Q}) \tag{1.65}$$

Since the points p' and q' are rational we can compute their parameter values for $X_{\mathcal{C}}$ by solving a linear equation. Let these values be given by $(\lambda_{p'}, \mu_{p'}) \in \mathbb{P}^1(\mathbb{Q})$ and $(\lambda_{q'}, \mu_{q'}) \in \mathbb{P}^1(\mathbb{Q})$ for p' and q' respectively. Then the matrix

$$A = \begin{pmatrix} \lambda_{p'} & \lambda_{q'} \\ \mu_{p'} & \mu_{q'} \end{pmatrix} \in \mathbb{Q}^{2\times 2} \tag{1.66}$$

provides the mapping of the parameter values for $X'_{\mathcal{C}}$ onto those for $X_{\mathcal{C}}$. Since A is rational it can be used to convert all representations in seq' to the desired representations with respect to $X_{\mathcal{C}}$, even though the parameter values are defined in some algebraic extension.

Rational Point

Let $p \in \mathcal{S}$ be a rational point on $X'_{\mathcal{C}}$. In order to compute its parameter value on $X_{\mathcal{C}}$ we just recompute the parameter value on $X_{\mathcal{C}}$ without using the parameter value on $X'_{\mathcal{C}}$ at all. Since $\mathcal{C}$ is not a smooth quartic, $X_{\mathcal{C}}(\xi)$ is given in terms of rational functions:

$$X_{\mathcal{C}}(\xi) = [f_1(\xi), f_2(\xi), f_3(\xi), f_4(\xi)]^T, \tag{1.67}$$

where $f_i \in \mathbb{K}[\xi]$ and where $\mathbb{K}$ is the algebraic extension field over which $X_{\mathcal{C}}(\xi)$ is defined. Since p is a valid point in $\mathbb{P}^3$, let *w.l.o.g.* p_4 be the coordinate that is not zero. Then the parameter values for p are given by the real roots of

$$g(\xi) = gcd(s_{14}(\xi), s_{24}(\xi), s_{34}(\xi)), \text{ where } s_{ij}(\xi) = f_i(\xi)p_j - f_j(\xi)p_i. \tag{1.68}$$

due to the fact that p is rational this is defined over the algebraic extension of $X_{\mathcal{C}}$, that is, $g \in \mathbb{K}[\xi]$.

Remaining Cases

The principal idea for the remaining cases is to characterize $\mathcal{S}$ as the intersection of $\mathcal{C}$ with the singular locus of $\mathcal{P}'$ and to use this characterization to compute a sequence $seq = \{p_1, .., p_m\}$ representing the m points in $\mathcal{S}$ with respect to $X_{\mathcal{C}}$. Thereafter, these representations are matched with those in $seq' = \{p'_1, .., p'_m\}$ using Algorithm 4 as discussed in Section 1.7.2. It remains to compute $seq =$

$\{p_1, .., p_m\}$ representing the points in $\mathcal{S}$ with respect to $X_\mathcal{C}$. However, this is only possible if $\mathcal{C}$ itself is not part of the singular locus of $\mathcal{P}'$.

We shall verify that this is true for the remaining cases. Recall the characterization by Segre, Section 1.3.1. First of all observe that for all singular pencils the set $\mathcal{S}$ is comprised of one rational point only. Hence, we can chop of all cases in Table 1.3. It remains to check those cases in Table 1.2 in which the singular locus contains an algebraic component. These cases are:

- [1(111)] double regular conic:
  There is nothing to do due to the fact that $\mathcal{S}$ is the empty set.
- [(211)] two double lines:
  The two lines are defined in a rational plane and intersect in a rational point.
- [(22)] Two lines and a double line.
  The double line is a rational line. The two other lines are not contained in the singular locus of $\mathcal{P}'$.

Hence, from now on we can assume that $\mathcal{C}$ is not part of the singular locus of $\mathcal{P}'$ and that $\mathcal{P}'$ is a regular pencil. We shall now give a proper characterization of S with respect to $X_\mathcal{C}$. The set $\mathcal{S}$ is comprised of the intersection points of $\mathcal{C}$ with the other algebraic components induced by $\mathcal{P}'$. Since $\mathcal{C}$ itself is not part of the singular locus these are exactly the singular points on $\mathcal{C}$. Let Q_S and Q_T be two regular quadrics contained in $\mathcal{P}'$, the singular locus of $\mathcal{P}'$ is defined as $\{p \in Q_S \cap Q_T \mid \nabla Q_S(p) = \nabla Q_T(p)\}$, this is well defined due to the fact that Q_S and Q_T are regular. By the fact that $\nabla Q_S(p) = S \cdot p$ we can define $\mathcal{S}$ as follows:

$$\mathcal{S} := \{p \in C \mid Sp = Tp\} \tag{1.69}$$

In terms of the parameterizations $X_\mathcal{C}$ this is:

$$\mathcal{S} := \{X_\mathcal{C}(\xi) \mid SX_\mathcal{C}(\xi) = TX_\mathcal{C}(\xi),\ \xi \in \mathbb{P}^1\} \tag{1.70}$$

Now write $SX_\mathcal{C}(\xi)$ and $TX_\mathcal{C}(\xi)$ as follows:

$$[f_1(\xi), f_2(\xi), f_3(\xi), f_4(\xi)]^T := SX_\mathcal{C}(\xi) \tag{1.71}$$
$$[g_1(\xi), g_2(\xi), g_3(\xi), g_4(\xi)]^T := TX_\mathcal{C}(\xi) \tag{1.72}$$

Then the parameter values for the points in $\mathcal{S}$ are given by the real roots of

$$gcd(s_{12}(\xi), s_{13}(\xi), s_{14}(\xi), s_{23}(\xi), s_{24}(\xi), s_{34}(\xi)), \tag{1.73}$$

where $s_{ij}(\xi) = f_i(\xi)g_j(\xi) - f_i(\xi)g_j(\xi)$.

This results in the desired sequence $seq = \{p_1, .., p_m\}$ representing the points in $\mathcal{S}$ with respect to $X_{\mathcal{C}}$.

1.9 Implementation and Benchmarks

Our software consists of two parts. The first part is an adaptor to the software QI [58] which implements the approach by Dupont *et al.* [23, 24, 25] as discussed in Section 1.5. The second part is based on the intersection of an algebraic component with another quadric as discussed in Section 1.6. This part is implemented within the QUADRIX library, which is part of the EXACUS project [5]. It provides the computation of the adjacency graph of a set of quadrics. Section 1.1.2 completes the brief discussion of the overall algorithm as it was given in Section 1.1.2.

As the whole EXACUS project, our software follows the generic programming paradigm with C++ templates similar to well-established design principles, for example, in the STL [2] and in CGAL [13]. The algebraic tools needed by our approach are provided by the NUMERIX library, which is also part of the EXACUS project. Section 1.9.2 identifies the most important tools of our approach and gives a brief description of these tools insofar as they are relevant for the subsequent discussion of the benchmarks in Section 1.9.3.

1.9.1 Details Overall Algorithm

Given a set $\mathcal{S}$ of quadric surfaces, defined by rational coefficients of any size. Our algorithm computes the adjacency graph $\mathcal{G}(\mathcal{S})$ of the arrangement $\mathcal{A}(\mathcal{S})$, that is, it computes all vertices and their connectivity along the edges of $\mathcal{A}(\mathcal{S})$.

Data Structure

The data structure representing the adjacency graph $\mathcal{G}(\mathcal{S})$ is meant to be a preliminary stage towards the arrangement $\mathcal{A}(\mathcal{S})$. We plan to store the arrangement in a variant of a structure used to represent *Nef polyhedra*[13] as presented in [35]. This structure is a vertex oriented structure, that is, the information stored within the vertices is in principal sufficient to represent the arrangement. In particular, each vertex stores its local neighborhood in a so called *sphere map*.

[13] Nef polyhedra in d-dimensional space are the closure of half-spaces under boolean set operations.

The data structure representing $\mathcal{G}(\mathcal{S})$ is organized in the same spirit. Beside the connectivity information to the other vertices, each vertex already stores all information needed to determine its local neighborhood.

For each vertex v we store:

- All quadrics the vertex v lies on.
- For each algebraic component $\mathcal{C}$ the vertex v lies on:
  - the parameterization $X_{\mathcal{C}}$ of $\mathcal{C}$
  - the parameter value with respect to $X_{\mathcal{C}}$
  - the next vertex with respect to the sorting on $\mathcal{C}$
  - the previous vertex with respect to the sorting on $\mathcal{C}$
- An explicit representation of the coordinates in $\mathbb{P}^3$ if available.

For each algebraic component $\mathcal{C}$ we store:

- The parameterization of the component.
- All quadrics the component $\mathcal{C}$ lies on.
- A sorted list of all vertices on that component.

Algorithm

Though the final data structure represents each vertex by exactly one object, we do not guarantee this through out the algorithm. In particular, we can not keep one sorted list of all vertices constructed so far, since this involves the comparison of two arbitrary vertices. Note that we have to avoid this kind of comparison since this can be too expensive, see Section 1.7.2. Instead we take advantage of the fact that we can efficiently match sequences of vertices representing the same set of points and that it is easy to compare vertices if they are defined on the same component.

In principal the overall algorithm has the following phases:

0. Initialization
1. Computation of all features induced by one quadric
2. Computation of all features induced by all pairs of quadrics
3. Computation of all features induced by all triples of quadrics
4. Sorting of vertices along algebraic components

In the sequel we discuss the details of the algorithm along the different phases. The most important phases are the second phase constructing all algebraic components and the third phase constructing nearly all vertices, *i.e.* all vertices induced by the intersection of three quadrics. The unique representation of a vertex is guaranteed by the fourth phase, since it detects equality among vertices while sorting them along the algebraic components.

Phase 0: Initialization
This phase just ensures that the intersection of two or more surfaces from $\mathcal{S}$ will result in low dimensional features only, that is, it ensures that all surfaces are coprime. Consequently, $\mathcal{S}$ may also contain rational planes but this has no effect on the overall algorithm since we could interpret each rational plane as a quadric by defining it as a double plane. In the sequel we will not mention rational planes in particular.

Phase 1: Features induced by one Quadric
This phase introduces lower dimensional features of singular input quadrics. In case of a cone this is just the rational point representing the apex of the cone. In the case of two intersecting planes we have to construct the rational line, which is the intersection of the two planes. Though the construction of these entities is rather trivial we have to keep track of these entities in the second phase.

For each quadric $Q \in \mathcal{S}$ do:

1.1 If Q has inertia $(3,0)$ or $(2,1)$: Q is a complex or real cone with a rational point p being its singular locus. Construct a vertex representing p.
1.3 If Q has inertia $(2,0)$ or $(1,1)$: Q represents two complex or real planes intersecting in a real rational line $\mathcal{L}$. Construct an algebraic component representing $\mathcal{L}$.

Phase 2: Intersection of all Pairs of Quadrics
First of all this phase iterates over all pairs of quadrics and constructs all further algebraic components. Thereby, it ensures that each algebraic component is represented exactly once, see Section 1.8.1. The constructed vertices in this phase are the intersection points of two algebraic components, self intersections or isolated points.

Remark: An isolated point is first of all considered as a separate algebraic component. The vertex representing the isolated point is defined as the only point on that algebraic component. At the first glance this seems to be a strange gimmick,

but it avoids a lot of special cases in the actual code as well as in the subsequent discussion of the algorithm.

Moreover, the second phase has to incorporate the entities which have been constructed in the first phase. Let $\mathcal{P}$ be the pencil defined by Q_S and Q_T.

- If one of the quadrics is a cone let p be its singular point. If $p \notin Q_S \cap Q_T$ there is nothing to do. Otherwise, we have to locate p within $Q_S \cap Q_T$. First of all note that $Q_S \cap Q_T$ is not a smooth quartic due to the fact that $Q_S \cap Q_T$ has at least one singular point, namely p. Consequently, each algebraic component $\mathcal{C} \subset Q_S \cap Q_T$ is given by a rational parameterization $X_{\mathcal{C}}$. Hence, for each algebraic component $C \subset Q_S \cap Q_T$ it is easy to check whether p lies on $\mathcal{C}$ and if so to compute its parameter value with respect to $X_{\mathcal{C}}$, see also Section 1.8.2.

- If one of the quadrics represents two intersecting planes let $\mathcal{L}$ be its singular rational line of Q_S. In this case we first of all compute the intersection of $\mathcal{L}$ with the second quadric Q_T. If this is empty or the full line $\mathcal{L}$ there is nothing to do. Otherwise let $seq_{\mathcal{L}}$ be the sequence of intersection points representing $\mathcal{L} \cap Q_T$. Given the fact that $\mathcal{L}$ is not part of the singular locus of $\mathcal{P}$ it is clear that the $seq_{\mathcal{L}}$ is representing at least a subset of all singular points induced by $\mathcal{P}$. Hence, we can use Algorithm 4 to match $seq_{\mathcal{L}}$ with the singular points in $\mathcal{P}$.

For each pair (Q_S, Q_T) of quadrics out of $\mathcal{S}$ do:

2.1 Compute a unique representation of the pencil $\mathcal{P}$ defined by Q_S and Q_T. Use the cache, see Section 1.8.1 to avoid unnecessary reconstructions of algebraic components
2.2 For a new pencil, use Dupont's algorithm to compute the intersection of Q_S and Q_T, see Section 1.5.
2.3 For each constructed cubic, conic or line check that the component is new. If it is not, unify it with the old representation as discussed in Section 1.8.1.
2.4 If applicable, let p be the singular point of, *w.l.o.g.*, Q_S constructed in step 1.1. If $p \notin Q_T$, do nothing. Otherwise, for each component $\mathcal{C} \subset Q_S \cap Q_T$ check if p lies on $\mathcal{C}$. If $p \in \mathcal{C}$ compute the parameter value with respect to the parameterization of $\mathcal{C}$ and store the result within the list of representations of p.

2.5 If applicable, let $\mathcal{L}$ be the singular line of, *w.l.o.g.*, Q_S constructed in step 1.2. If $\mathcal{L} \cap Q_T = \emptyset$ or $\mathcal{L} \cap Q_T = \mathcal{L}$, do nothing. Otherwise, let $seq_{\mathcal{L}}$ be the sequence of vertices representing $\mathcal{L} \cap Q_T$ and $seq_{\mathcal{P}}$ be the set of singular points induced by $\mathcal{P}$. Since $seq_{\mathcal{L}}$ represents a subset of $seq_{\mathcal{P}}$ we can use Algorithm 4 to match the points in $seq_{\mathcal{L}}$ and $seq_{\mathcal{P}}$. Unify the representations of the matched points.

Phase 3: Intersection of all Triples
This is the most important phase since it constructs nearly all vertices, *i.e.* all vertices defined by the intersection of three quadrics. Note that it iterates over all triples of quadrics and not over all algebraic components. In this way we obtain all representations of points induced by three quadrics at once. This has the advantage that we can efficiently match these representations using Algorithm 4 as it has been discussed in Section 1.7.2.

For each triple (Q_1, Q_2, Q_3) of quadrics out of $\mathcal{S}$ do:

3.1. Intersect all algebraic components in $Q_1 \cap Q_2$ with Q_3. For all resulting intersection points store their representations into the sequence seq_{12}.
3.2. Unify representations in seq_{12} representing the same point. This is necessary due to the fact that an intersection point in $Q_1 \cap Q_2 \cap Q_3$ may be common to several components in $Q_1 \cap Q_2$. In this case this point is reported several times, once for each component it lies on. However, this is no problem, since the problematic points are already reported by the algorithm computing $Q_1 \cap Q_2$.
3.3. In the same way compute the sequences seq_{13} and seq_{23}.
Use Algorithm 4 to match the representations in seq_{12}, seq_{13} and seq_{23}.
3.5. For each algebraic component a vertex $v \in Q_1 \cap Q_2 \cap Q_3$ lies on, add v to the list of vertices of that algebraic component.

So far we can not guarantee that each point is represented by exactly one vertex. For instance, it may happen that a point is the intersection point of four quadrics. In this case the third phase constructs four vertices representing this point, one vertex for each possible triple out of the four quadrics. The equality of these vertices is detected in the fourth phase.

Phase 4: Sorting and Unification of Vertices
First of all this phase sorts all points along their algebraic components, which is the main building block in order to initialize the adjacency graph. As discussed

in Section 1.7.1 this comparison is based on the parameter values of the vertices with respect to their common component. In principal this is comparison of algebraic numbers, see also Section 2.4.1.

An important side effect of this process is that it also detects equality among the vertices. If two vertices are equal we merge all the information stored in the representations of both vertices into one. At the end of the sorting process all equal vertices have exactly the same representation. This is due to the fact that all vertices have at least one algebraic component in common and that we sort along all algebraic components. This process is facilitated by the fact that a vertex is represented by a handle class[14] with a union find data structure. Hence, all equal vertices are finally pointing to exactly one and the same representation in memory.

4.1 For each smooth quartic $\mathcal{C}$ split the list of vertices on that smooth quartic according to the connected components of $\mathcal{C}$. All other components consist of exactly one connected component.
4.2 For each connected component $\mathcal{CC}$,
- sort the list of vertices on that component.
- erase duplicates from the sorted list of vertices.
- initialize previous and next entries of the vertices with respect to $\mathcal{CC}$

1.9.2 Algebraic Tools

In order to give a certain background for the benchmarks in Section 1.9.3 we now discuss the most important types and methods used within our implementation. All algebraic tools are provided by the library NumeriX, which is also part of the Exacus project. The library has been developed in parallel to our implementation. As a consequence the NumeriX library has been influenced a lot by the needs of our approach. Note that a detailed discussion of the NumeriX library itself is an essential topic in Chapter 2.

[14] This handle class is a class template provided by the Library Support Layer of Exacus. The class has been introduced by Lutz Kettner [53].

Number Types

As the whole EXACUS project, our software follows the generic programming paradigm with C++ templates. In particular we can instantiate our algorithms on such a small scale as number types and arithmetic operations. In order to exchange the underlying number types our algorithm and classes can be instantiated by a so called `Arithmetic_traits`. This class is supposed to provide a collection of basic number types. These number types are expected to be connatural such that they interoperate in a well-defined way, see also Section 2.2.4. Currently, the NUMERIX library supports two possible sets:

- `leda::integer`, `leda::rational`, `leda::real`, `leda::bigfloat`
- `CORE::BigInt`, `CORE::BigRat`, `CORE::Expr`, `CORE::BigFloat`

However, in general the intersection of two quadrics is a smooth quartic. And in most cases the parameterization of a smooth quartic is given over an algebraic extension of degree 2, namely $\mathbb{Q}$ extended by the square root of the determinant of the chosen ruled quadric. As a consequence for each parameterization all subsequent arithmetic operations are carried out over this algebraic extension. Though `leda::real` [62] or `CORE::Expr` [52] are capable to represent such an algebraic extensions, it is far too expensive to use these types in this context. This is due to the fact that they are designed for a more general purpose and that the internal expression tree, which is used to represent a value, grows with each arithmetic operation. In particular, these types should not be used as the coefficient type in gcd computations of polynomials.

To represent algebraic extensions of degree 2 we introduced the class template `Sqrt_extension<NT,ROOT>`. The type has been designed such that only values from the same extensions are interoperable. This has the advantage that it is possible to keep the representation of objects very simple. In particular it can be used as coefficient type in resultant and gcd computations. An object of this type stores two coefficients `a1` and `a2` of type `NT` and the extension `root` of type `ROOT` representing the value $a_1 + a_2\sqrt{root}$. Note that both `NT` and `ROOT` can themselves be an instance of `NiX::Sqrt_extension`, yielding a nested extension. In particular, it is capable to represent an extension of degree 2×2.

The type which is used as the coefficient type within an algebraic component depends on the algebraic extension the algebraic component is defined in. If the

coefficients are not extended, as it always is the case for cubics, we use the integer type provided by the arithmetic traits. If the extension is of degree 2 or 2×2 we use the type `Sqrt_extension` in its normal or nested form, respectively. In the rare cases of lines in which the algebraic extension is of degree 3 or 4 we use `leda::real` or `CORE::Expr`, see also Section 1.6.1.

MPFI Arithmetic

An important filter within our approach is the use of interval arithmetic based on multiprecision floating point arithmetic or *MPFI Arithmetic* for short. Interval arithmetic is a well known technique to control accumulated rounding errors of floating point computations at run time.

In general an interval is represented as a pair of numbers and acts in place of all the numbers in between these two:

$$[a, b] := \{x \in \mathbb{R} \mid a \leq x \leq b\}, \text{ where } a,\ b \in \mathbb{R}. \tag{1.74}$$

Now, let f be some function from $\mathbb{R}$ to $\mathbb{R}$, then this function can be generalized to a function F on intervals such that

$$f(x) \in F([a, b]),\ \forall x \in [a, b]. \tag{1.75}$$

This fundamental property of interval arithmetic is called the *inclusion property*. Of course it is not useful to implement the interval arithmetic such that the new functions always answer $[-\infty, +\infty]$. Whenever possible, the result should be the smallest interval which is able to satisfy the property. For an implementation based on floating point arithmetic it is clear that, due to the presence of rounding errors, it is not possible to implement F such that the result is always the smallest possible interval. However, the inclusion property is guaranteed by a careful choice of the rounding modes for the involved arithmetic operations. Note that this scheme can be extended to functions with more than one argument as well.

Based on the inclusion property it is possible to use interval arithmetic as a filter. For instance it is possible to deduce the sign of an arithmetic expression using interval arithmetic. If the borders of the interval have the same sign, this is the sign of the true value as well. Otherwise, the sign is still unclear. If the precision of the floating point arithmetic is fixed, *e.g.* in case of double arithmetic, we have

to evaluate the expression in some exact way. However, in case a multiprecision floating point type is available it is also possible to evaluate the expression with an increased relative precision, that is, we increase the number bits used to represent the mantissa. If we, for some reason, know that the exact result is not zero we can even repeat this process until the resulting interval has an unambiguous sign.

In case our algorithm is instantiated by the LEDA number types we use the class template `interval<T>` provided by the BOOST library, which we instantiate by `leda::bigfloat`. In case we use CORE the type `CORE::BigFloat` itself can be interpreted as an interval since it stores a central and an error value. However, for the way we use the MPFI arithmetic in our code this has no effect since the NUMERIX library provides a smooth interface which hides these technical details.

Real Roots

Nearly all intersection points are defined by a parameter value with respect to the parameterization of the algebraic component they are defined on. In general such a parameter value is characterized as the real root of a univariate polynomial. Fortunately, we have been able to design our algorithms such that we do not need any arithmetic operations on these numbers. The only required functionalities are:

- Exact comparison with another real root of a univariate polynomial defined over the same algebraic extension.
- Test whether the value is also the root of another polynomial defined over the same algebraic extension. (is root of)
- Providing an approximation with respect to a given relative precision.

To represent these roots we use the class template `Algebraic_real`. This class has been introduced to represent x-coordinates within planar arrangement computations of algebraic curves [7]. It has been designed to provide efficient comparisons, but arithmetic operations are not supported. An object of this type stores a real algebraic number in the form of a square free polynomial and an interval. This interval isolates the represented real root from the other real roots

of the polynomial. The type is implemented as a class template. Template parameters are the coefficient type of the polynomial and the boundary type of the isolating interval. The most time critical internal operations are:

- The gcd computation of two polynomial within the comparison of two numbers.
- The evaluation of the sign of the defining polynomial at values defined by the boundary type. This is needed to refine the defining interval in order to obtain a better approximation.

For the approach presented here the type has been revised in order to support the type `NiX::Sqrt_extension` as coefficient type as well. Moreover, the sign evaluation has been filtered using interval arithmetic based on the multiprecision floating point type provided by the arithmetic traits, that is, `CORE::BigFloat` or `leda::bigfloat`. Note that we currently just half the interval within each refinement step. It is planned to integrate a quadratic refinement as indicated by Abbott [1]. So far this has been postponed, since this is not a bottleneck of our approach. For a more detailed discussion of the type `NiX::Algebraic_real`, *e.g.* on more general optimizations, see Section 2.4.1.

Greatest Common Divisor and Modular Methods

An essential and unfortunately also very costly operation is the computation of the greatest common divisor of two polynomials. The gcd is most notably needed within the square free factorization[15] of the resultant polynomial, but also within the comparison of two algebraic real numbers. In our case this is especially grave due to the fact that the gcd is required over an algebraic extension. In the generic case this is an algebraic extension of degree 2.

First of all note that our polynomials are represented using integer coefficients. This is much more efficient than using rationals due to the fact that it saves simplifying, *i.e.* canceling down, the coefficient one by one. For the polynomial used within the parameterizations this is possible due to the fact that we are working in projective space. For the other polynomials it is possible because we are mainly interested in their zero set. For algebraic extension of degree 2 or 2×2 it turned out that it is also more efficient to use integer as the inner coefficient of

[15] We use Yun's algorithm [85] to compute the square free factorization.

the type `Sqrt_extension`. However, while each irreducible polynomial in $\mathbb{Z}[x]$ is still irreducible in $\mathbb{Q}[x]$, this is not true for polynomials in $\mathbb{Z}[\sqrt{\delta}][x]$ and $\mathbb{Q}[\sqrt{\delta}][x]$, respectively.[16] In particular, there is no proper definition of a gcd in an algebraic extension of $\mathbb{Z}$, *e.g.* see [42, 43, 78]. However, insofar as gcd computations are concerned, we are just interested in the common roots of polynomials and not in the constant factors. Therefore, we changed our algorithms such that they just need a gcd *up to constant factor* which is well defined for polynomials over algebraic extensions of $\mathbb{Z}$ as well.

In the rough, the gcd for polynomials over integer coefficients had been implemented using the subresultant algorithm as presented in [15]. For the polynomials with extended coefficients we used a classical implementation that removes all possible constant factors after each pseudo-division of the Euclidean algorithm. Though both algorithms are better than a naive implementation it turned out that the gcd computation is a major bottleneck.

As a first step, we introduced the so called *Modular Filter* [47] in order to avoid unnecessary gcd computations, that is, the computed gcd is just a constant polynomial. The principal idea is to consider the gcd of the polynomials over $\mathbb{Z}/p\mathbb{Z}$. If the gcd over $\mathbb{Z}/p\mathbb{Z}$ is constant this implies that the gcd over $\mathbb{Z}$ is also constant. Note that this can be generalized to algebraic extensions as well, see Hemmer *et al.* [47]. Though the modular filter avoids almost all unnecessary gcd computations it is not able to improve the performance for the necessary ones. Therefore, we integrated the number theory library NTL [73], which provides fast gcd computations based on modular methods, as for instance presented by Brown [14]. Unfortunately, the provided gcd is capable to handle only polynomials defined over integer coefficients. To the best of our knowledge there is no open source software that provides a gcd based on modular methods over algebraic extensions yet. Therefore, we decided to implemented our own gcd for algebraic extensions base on modular methods as indicated in [14, 31, 79]. So far the implementation is capable to handle univariate polynomials over algebraic extensions of degree 2 and 2×2. For integer polynomials of low degree, as it is the cases in the context of quadrics, our implementation is competitive to the one provided by the NTL. In fact it is better to use our own implementation since this saves the conversion from and to NTL types. For more details see also

[16] As an example take the polynomial: $2x^2 + 2\sqrt{2}x + 1$, which is irreducible in $\mathbb{Z}[\sqrt{2}][x]$, but factors into $1/2(2x + \sqrt{2})^2$ in $\mathbb{Q}[\sqrt{2}][x]$.

Section 2.3 or [46].

Real Root Isolation:

To define our parameter values, *i.e.* algebraic real numbers, we have to compute isolating intervals of the real roots of the square free factors of the resultant polynomial. For polynomials in $\mathbb{Z}[x]$ we use the Descartes Method [18] to isolate the roots. For polynomials over algebraic extensions of degree 2 it turned out that the original Descartes Method is very slow. This is due to the fact that the arithmetic operations within the algorithm are not filtered which has the effect that the algorithm suffers from the overhead due to the algebraic extension.

For polynomials in an algebraic extension we use a new variant presented by Eigenwillig *et al.* [28], the so called *Bitstream Descartes*: the coefficients of the polynomial are converted to (potentially infinite) bitstreams and a variant of the Descartes Method is used to determine the isolating intervals of the real roots. This method performs better than the original Descartes Method on polynomials in $\mathbb{Z}[\sqrt{\delta}][x]$ since the bitstream approach almost removes the overhead caused by the algebraic extensions. The only additional costs compared to integer coefficients is the conversion to bitstreams which is more expensive for algebraic extensions than for integers.

1.9.3 Benchmarks

We have not analyzed the worst case in the bit-complexity model, since we argue that our algorithms are adaptive in the bit-complexity and a worst-case analysis would not be representative. Instead, we want to show that our parameterization method is practical, *i.e.* efficient, for computing arrangements. In particular we want to analyze the competitiveness of our approach with respect to the projection approach. Since both approaches were implemented within the Exacus project, we were able to benchmark them in parallel.

We next give a short description of the projection approach by Berberich *et al.* [9]. Thereafter, we present the benchmarks comparing our approach with the projection approach.

The Projection Approach by Berberich et al.

As in our case the goal of the projection approach is to compute the 3D arrangement which is induced by a given set $\mathcal{S} = \{Q_1, \ldots, Q_n\}$ of quadrics defined by rational coefficients of arbitrary size. The principal idea is to split the computation of the 3D arrangement into two steps. The first step, is to compute for every quadric $\mathbb{Q}_i$ in $\mathcal{S}$ the 2D arrangement which is induced on the surface of Q_i by all other quadrics in $\mathcal{S}$. This step is based on *projection* of the appearing intersection curves onto the xy-plane. Note that these arrangements are computed separately for each quadric. In the second step, the plan is to use these arrangements in order to deduce the complete 3D arrangement. Though only the first step is presented in Berberich *et al.* [9], we consider this as a very promising approach.

In order to compute the arrangement on one quadric, say Q_1, all intersection curves $Q_1 \cap Q_i$, $2 \leq i \leq n$, as well as the silhouette curve of Q_1 are *projected* onto the xy-plane by classical resultant computations eliminating z. The resulting curves are represented by polynomials over integer coefficients. The degree of these polynomials is at most 4. Due to the projection they loose the spatial information: branches on the upper and lower part of Q_1 are projected on top of each other. Therefore, each projected curve is first of all decomposed into arcs with respect to the lower and upper part of Q_i. Thereafter, the planar arrangements for the upper and the lower part are computed separately by a variant of the Bentley-Ottmann sweep-line algorithm [4]. Both arrangements together completely describe the arrangement of intersection curves on the surface of Q_1.

The computation of the predicates which are needed by the sweep-line algorithm is reduced to the analysis of up to two projected curves at the same time, which is called a *Curve Pair Analysis* (CPA). In principal this is a cylindrical algebraic decomposition [17] of the two curves. First of all all event points are projected onto the x-axis by further resultant computations. The event points are the x-critical points of the curves and the intersection points of both curves. Although the projected curves are of algebraic degree 4, the coordinates of the intersection points of two curves are of algebraic degree 8. This is due to the fact that the up to 16 intersection points split into two groups of 8 points. The first group are the true intersection points in $\mathbb{R}^3$ the others are the intersection points caused

by the projection of the curves. Therefore, only points in the same group may be algebraic conjugates. The polynomials of degree 8 defining the coordinates of the intersection points are deduced by a multi-resultant computation using the three involved quadrics. After the computation of x-coordinates the CPA is completed by computing the intersections of y-slices with the curves. These slices are in general computed in between the x-coordinates, which has the advantage that involved polynomials are not defined over any algebraic extension. Hence, the basic operations, such as root isolation or gcd computation, are in general performed over univariate polynomials with integer coefficients of degree up to 8.

The main disadvantage of the approach is that in case of covertical events on the same curve a shear of the coordinate system must take place and everything has to be recomputed with respect to the new coordinate system. Moreover, it does not provide an explicit parameterization of the appearing intersection curves.

Benchmark Instances

In order to show different aspects within the benchmarks we generated three different families of instances:

rnd Each instance in this family contains random quadrics of a fixed bit-size. The instances differ in the number of contained quadrics. The quadrics are given by random integer coefficients of 50 bits each. In order to avoid empty quadrics or empty intersections, each quadric is guaranteed to intersect the $[-100, 100]^3$ cube at least once. All other quadrics have been discarded.

deg As for the rnd-instances this family varies in the number of quadrics and has also been used within Berberich *et al.* [9]. The quadrics within this family have been generated by interpolation. In order to achieve degenerate situations, several quadrics share values for partial derivatives and higher-order derivatives at common intersection points. The coefficients of this family are given by integer coefficients with 73 bits on average.

bits Each instance of the family contains a constant number of random quadrics. The number of quadrics is 50. The instances differ in the bit-size of the coefficients of the quadrics. The number of bits varies from 10 to 190 bits. The quadrics are generated in the same way as for the rnd-family.

All benchmarks were measured on a Pentium(R) M processor 1.7 GHz with 512 kB cache under Debian Linux and the GNU C++ compiler v4.1 with optimizations (`-O2`).

Parameterization vs. Projection

For a given set $\mathcal{S} = \{Q_1, \ldots, Q_n\}$ of quadrics both approaches aim for the computation of the full 3D arrangement induced by the set $\mathcal{S}$. Unfortunately, non of them has reached this goal yet. Hence, it was not possible to compare the approaches on this final level. Our approach is currently only capable to compute the 3D adjacency graph of the arrangement, while the projection approach only computes the different arrangements on the surface of each quadric.

Both approaches differ in their layout. Therefore, we first of all had to find a common level in order to compare both algorithms in a fair way. In case of the projection approach we decided to measure the computation of the arrangement on the first quadric. Hence, for our approach we modified the algorithm such that it also just computes the adjacency graph of the arrangement on the first quadric. More precisely, the modified algorithm computes:

- The parameterization of all intersection curves $Q_1 \cap Q_i$ for $i = 2, \ldots, n$.
- The intersection of these curves with the other quadrics, which results in different representations of the intersection points with respect to the different intersection curves they lie on.
- The matching of these representations and their merging into vertices.
- The sorting of the vertices along the curves.

We compared our approach to the projection approach on the random instances (rnd) as well as on the degenerate instances (deg). The results are presented in Figure 1.2. First of all note that the runtime of both algorithms seems quadratic in the number of quadrics. This is due to the fact that the runtime of both algorithms is dominated by the quadratic number of intersections. In case of the projection algorithm these are the pairwise intersections of the n projected curves. In our case these are the intersections of all algebraic components on Q_1 with the other quadrics in $\mathcal{S}$. For the chosen instances the number of vertices in the arrangement on Q_1 is quadratic in the number of quadrics. Therefore,

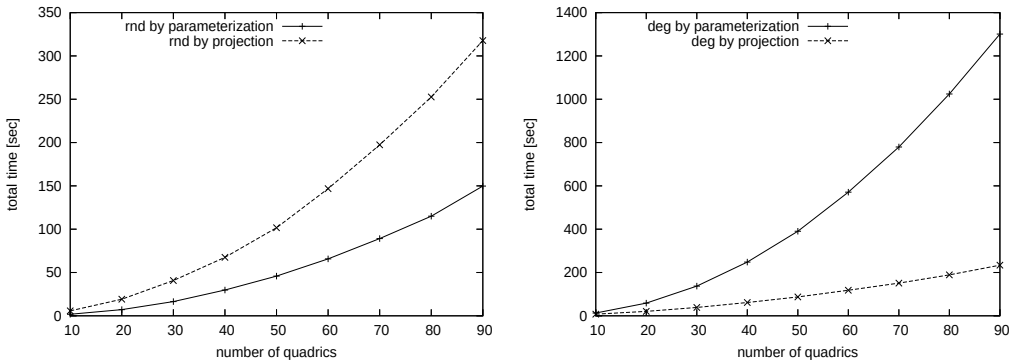


Figure 1.2: The parameterization approach compared to the projection approach. The left plot shows the timings for the random instances (rnd). The right plot shows the timings for the degenerate instances (deg).

both algorithms seem to perform linear in the number of vertices.[17] Note that we can expect that the projection approach is output sensitive due to the use of the sweep-line algorithm. However, we can not state this for our approach because we have not yet integrated any geometric filtering, *e.g.* a Bounding Volume Hierarchy as indicated in Schömer *et al.* [70].

The right plot in Figure 1.2 shows that for the degenerate instances the projection approach performs better than our parameterization approach. This is due to the fact that for the degenerate situations it is not possible to avoid the computation of very expensive gcds, as is the case for equal algebraic numbers (*e.g.* equal intersection points) or non-square-free resultants (*e.g.* tangential intersection points). These computations can not be avoided by both approaches. However, in our case this is much more expensive for the following two reasons.

1. Our intersection curves are represented in the parameter space of the chosen quadric Q_R. This introduces a considerable amount of extra bits, *e.g.*, the number of bits in the resultant polynomials for input quadrics with 50 bit integers is 4500 compared to just 1000 bits for the corresponding polynomial in the projection approach.

2. In our case most polynomials are defined over an algebraic extension of

[17] Due to the fact that the projection approach induces more vertices, it was not possible to show this effect within one plot.

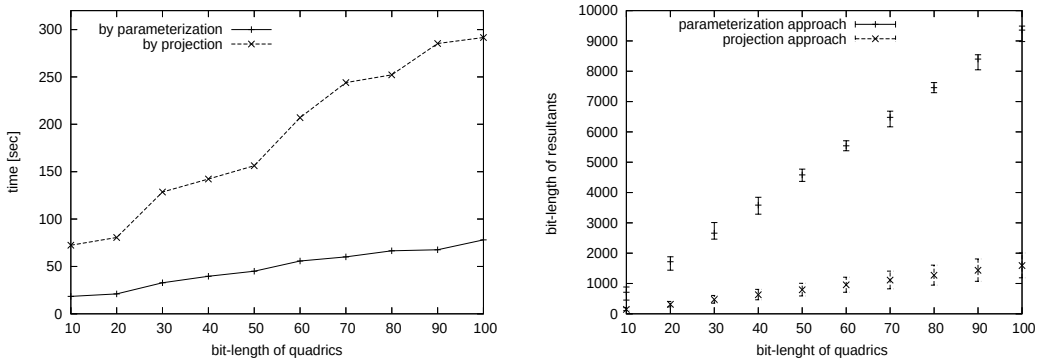


Figure 1.3: Projection approach compared to the parameterization approach on random instances with growing bits. The left plot shows the timings. The right plot shows the bit size of the coefficients in the resultant polynomial.

degree 2, which increases the number of exact arithmetic operations significantly. For instance the multiplication of two numbers of the type `Sqrt_extension` requires 5 multiplications and 2 additions.

$$(a + b\sqrt{c})(a' + b'\sqrt{c}) = (aa' + bb'c) + (ab' + ba')\sqrt{c} \tag{1.76}$$

Moreover, we can not apply the more efficient algorithms for square-free factorization and gcd computation as they are used for the integer polynomials in the projection approach. Indeed, for the degenerate instances, the time for the gcd computation contributes about 80% to the total runtime of the parameterization approach.

The left plot in Figure 1.2 shows that for the random instances the parameterization approach is significantly faster than the projection approach. This effect is even amplified if the number of bits increases as it is shown in the left plot of Figure 1.3. Note that this is paradoxical to the fact that in our case the number of bits increases 4.5 times faster than for the projection approach. This can be observed in the right plot of Figure 1.3. This shows that for the generic case we have been able to widely decouple our approach from the bit-size of the input. This can be first of all explained by the consistent use of multiprecision floating point interval arithmetic (MPFI) within our approach. This is possible due to the available parameterization of the intersection curves. By contrast, the projection

approach has to apply a root isolation algorithm to the resultant polynomial on the x-axis but also to the polynomials appearing due to the y-slices within the CPA.

Generic Instances Details

We chose the random instances with growing bits in order to give a more detailed analysis for the generic case. The left plot in Figure 1.4 shows that the total runtime is dominated by the time spent within the algebraic component/quadric intersection. The other important steps within our algorithm, namely the intersection of quadrics as well as the matching and sorting of intersection points can be neglected.

- For the quadric/quadric intersection this is first of all caused by the fact that the coefficient size within this step is still quite small. Moreover, this step is only performed a linear[18] number of times.
- For the sorting and matching this is due to the efficient implementation of the type `Algebraic_real` and the use of MPFI arithmetic in Algorithm 4 as discussed in Section 1.7.2. In particular, these steps can be considered as independent from the bit size.

The right plot in Figure 1.4 shows a detailed analysis of the time spent within the intersection of the algebraic components with the other quadrics. The total time is broken down to the times for the resultant computation, the square free factorization and the root isolation. The last curve covers the remaining time.

- The curve representing the time spent within the resultant computation is the most significant curve and shows that the performance of the resultant computation is still depending on the number of involved bits. For the biggest instance, with input quadrics represented by 170 bit integer coefficients, the computation of the resultant contributes about 70% to the total runtime. This is due to the fact that all computations leading to the resultant polynomial are performed using traditional exact integer arithmetic,

[18] Note that this refers to the modified algorithm computing the adjacency graph on the first quadric only.

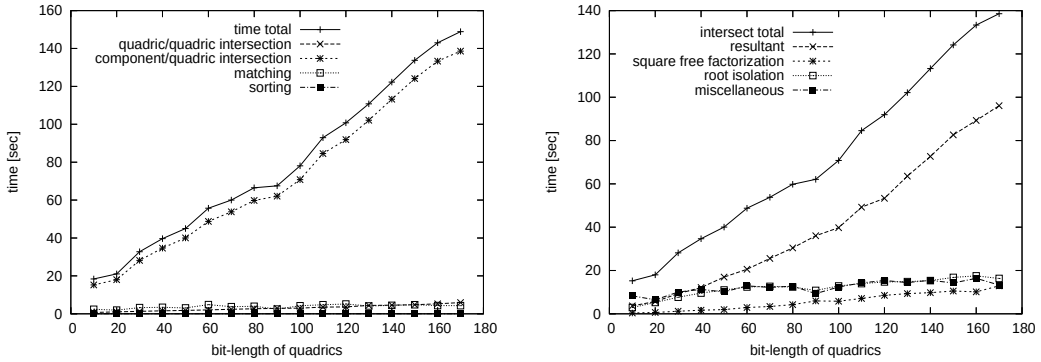


Figure 1.4: Detailed timings for the parameterization approach on the random instances with growing bits. The left plot shows the apportionment for the overall runtime. The right plot shows the apportionment for the curve/quadric intersection. In each plot the first curve is the sum of the others.

i.e. the used coefficient type within `Sqrt_extension` is `leda::integer` or `CORE::BigInt`.

- The next important step is the square free factorization of the resultant. First of all note that for the used instances (bits) all appearing polynomials are already square free. Hence, the time spent within the square free factorization is just the time spent for the modular filter [47] detecting this fact, *i.e.* a true square free factorization is never performed. However, the plot shows that the modular filter itself depends on the bit size as well. This is no surprise since all integer coefficients have to be converted into their modular images first, which is nothing else than computing $x \ mod \ p$, where x is the coefficient and p the chosen prime number.

- The used approach within the root isolation is the Bitstream Descartes Method by Eigenwillig *et al.* [28]. The plot shows that this method is independent from the bit size of the coefficients. This is due to the fact that the method as been designed such that it uses only as many leading bits of the coefficients as needed in order to isolate the roots. We have been able to apply this approach for integer coefficients and for `Sqrt_extension` coefficients as well.

- The last curve shows the time spent within the remaining task in order

to compute the correct arc corresponding to a real root of the resultant. These are mainly some 'is root of' tests and some sign computations of polynomials at the root, see also Algorithm 3 in Section 1.6.2. The plot shows that this is almost independent from the bit size as well. This has been achieved by the use of the MPFI arithmetic in the sign computations and the modular filter used within the 'is root of' tests.

Degenerate Instances Details

The left plot of Figure 1.5 shows that, as in the generic case, the total time is again dominated by the time spent within the intersection of the algebraic components with the other quadrics. This is even more significant as in the generic cases. However, the right plot in Figure 1.5 shows that the time spent within the curve/quadric intersection is dominated by the square free factorization of the resultant. This is due to the fact that for the degenerate instances the resultant is not always square free. In these cases the modular filter fails and Yun's algorithm has to call several very expensive gcd computations. Note that we abstained from introducing another family of degenerate instances with growing bits as the results for the existing family are already significant enough. In particular the costs for the gcd computations depend on the number of involved bits and will dominate all other costs within our approach.

Note that the used gcd is already based on modular methods. For more details on these methods and further benchmarks see Section 2.3.

Conclusions

Both approaches to the quadrics arrangement problem, the projection approach as well as our approach, have their own strengths and weaknesses.

- Though the parameterization approach in general introduces an algebraic extension of degree two and a considerable amount of extra bits the parameterization allows an easy application of MPFI arithmetic. In combination with the modular filter and the Bitstream Descartes Method this has the consequence that our approach is faster than the projection approach for the generic instances. However, in the degenerate cases it is not possible

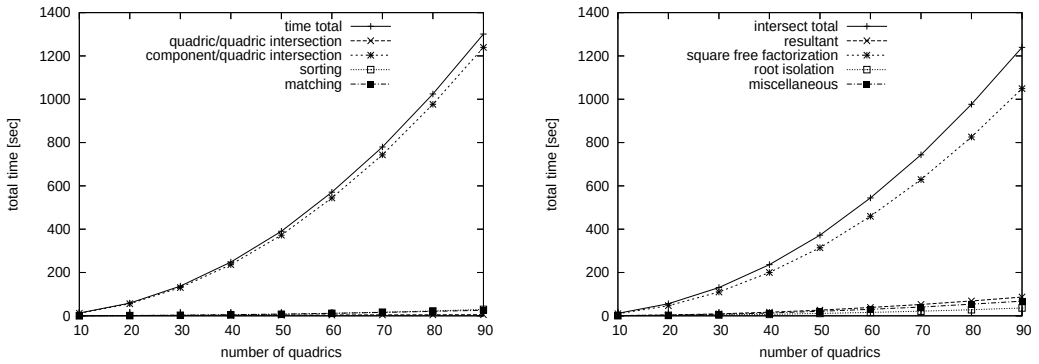


Figure 1.5: Detailed timings for the parameterization approach on the degenerate instances. The left plot shows the apportionment for the overall runtime. The right plot shows the apportionment for the curve/quadric intersection. In each plot the first curve is the sum of the others.

to avoid the very expensive gcd computations and we have to pay the bill for the extra bits introduced by the parameterization. Hence, we don't think that our approach will ever be faster than the projection approach for the degenerate situations. Even though the modular gcd methods may not be fully optimized it is just impossible that a gcd over algebraic extensions with more bits is faster than a gcd for polynomials with integer coefficients and fewer bits. However, in a setting in which one can expect non or only a few degeneracies the parameterization approach seems to be more appropriate than the projection approach.

- A clear disadvantage of our approach is that it is not extendable to more than just quadrics. This is due to the fact that the idea of ruled surfaces in the pencil is not applicable to higher degree surfaces. By contrast the projection approach is quite generic. In principal the main work is done by the curve pair analysis which is used to answer all predicates within the Bentley-Ottmann sweep-line algorithm. The other important step is to lift the projected arcs back to the correct part of the surface. Note that at least the first part is already available due to [27, 26] and the second part seems feasible using a similar ray shooting technique as it has already been applied for the projection approach.

1.10 Further Work

Though we have achieved a major milestone, namely the computation of the adjacency graph for a given set $\mathcal{S}$ of quadric surfaces, the computation of the 3D arrangement is still missing. However, the data structure representing the adjacency graph $\mathcal{G}(\mathcal{S})$ can be considered as a preliminary stage towards the data structure that represents the arrangement $\mathcal{A}(\mathcal{S})$. The idea is to store the arrangement $\mathcal{A}(\mathcal{S})$ in a variant of a structure used to represent *Nef-Polyhedra*.

Nef-Polyhedra in d-dimensional space are the closure of half-spaces under boolean set operations. As presented by Bieri and Nef [10], Nef-Polyhedra have the nice property that it is possible to represent them by modeling the local neighborhood of its vertices. This led to the idea of a vertex oriented structure, where each vertex stores its local neighborhood in a so called *sphere map*. The idea of the sphere map was introduced by Dobrindt *et al.* [20]. They proposed to compute the local neighborhood of a vertex by a symbolic intersection of the polyhedron with an ε-sphere around the vertex. The resulting surface is a planar Nef polyhedron embedded on the sphere. Each vertex in the sphere map corresponds to an incident edge of the 3D arrangement. In the same way each edge in the sphere map corresponds to a face in 3D. For each edge which is incident to a vertex, there is a corresponding vertex in the sphere map. Such a vertex is called an *svertex*. In the same way an incident face corresponds to a so called *sedge* in the sphere map. Figure 1.6 shows the data structure as it was presented by Granados *et al.* [35]. For a well-elaborated discussion see also [38, 37]. The approach is capable to support boolean operations. It has been implemented and published as a CGAL package by Peter Hachenberger as main author [39, 40].

The most powerful idea within this approach is to delay the construction of the final data structure and to perform all operations within the sphere maps of the vertices first. In the case of boolean set operations it is possible to first of all identify the relevant vertices, *i.e.* those which are part of the output, and to perform the set operations separately for each relevant sphere map. Thereafter, the vertices are sorted along common lines. This is used to link each svertex with the corresponding svertex in the sphere map of the next vertex. All other information such as the nesting of the shells can be deduced by ray shooting at a proper vertex.

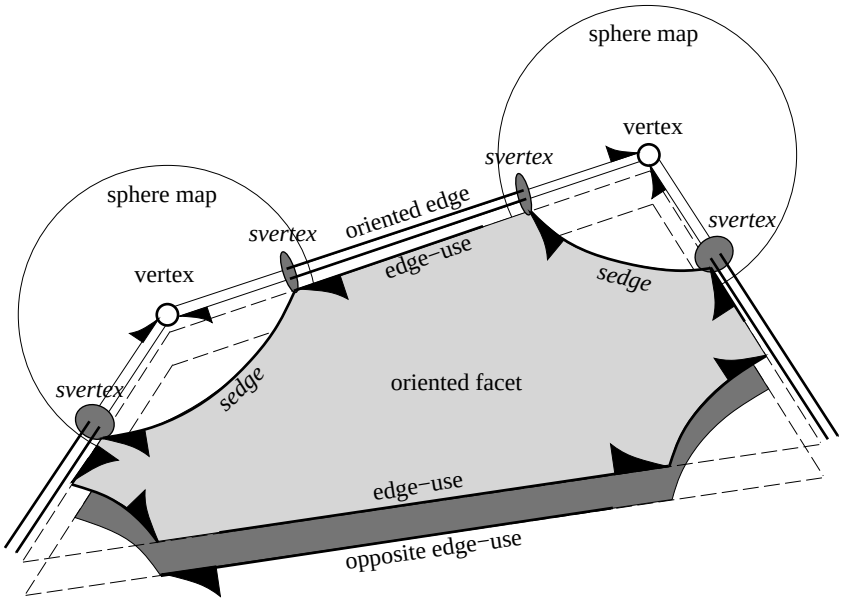


Figure 1.6: Data structure as it is used to represent Nef-Polyhedra in [35].

In case of quadrics, it should be possible to apply the same concepts. Our data structure representing $\mathcal{G}(\mathcal{S})$ is already organized in the same spirit, *i.e.* each vertex already stores all information needed to determine its local neighborhood. In particular, a sorting of the vertices is available due to the parameterization of the curves. However, a major missing part is the computation of the sphere map around each vertex. Fortunately, we are only conceptually interested in the sphere map, that is, the 2D arrangement representing the local neighborhood around the vertex may also be computed as the arrangement on a sufficiently small bounding box around the vertex. This has the advantage that the induced curves on the boundary of the box are just conics. For instance, the 2D arrangement on the box could be computed using the ConiX library [7], which is also part of the Exacus project. Since a bounding box of any desired precision can be computed using MPFI arithmetic the main difficulty is to detect that a bounding box is actually good enough. The bounding box of a vertex v should meet the following criteria:

- It should contain v in its interior. This can be guaranteed by computing a

more precise bounding box which is rounded afterwards.

- The arrangement on the bounding box should represent the neighborhood with respect to each quadric the vertex v lies on. If v lies on a smooth part of the quadric, the quadric should induce exactly one loop on the boundary of the box. If the vertex is part of the singular locus of the quadric this criterion is trivially met due to the fact that the quadric is point symmetric to v.
- The arrangement should represent the neighborhood with respect to each curve the vertex v lies on, that is, each intersection of the curve with the bounding box should be directly connected to v. This can be checked easily using the parameterization of the curves.
- The bounding box should not contain any other vertex of the arrangement. In particular no vertex which is induced by a quadric, a pair of quadrics or a triple of quadrics the vertex v lies on.

Note that this bounding box approach is only needed for degenerate situations, *i.e.* singular points or tangential intersection points. For a regular intersection point we can use the fact that the local neighborhood is isomorphic to the neighborhood induced by the tangential planes, *i.e.* it is enough to consider the tangential planes in order to compute a proper sphere map. Note that it is possible to use MPFI arithmetic for this case, since the planes are known to be in generic position.

1.11 Summary

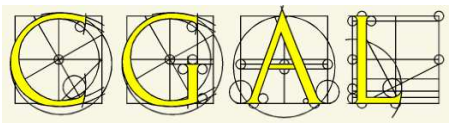
We have presented a major milestone towards the 3D arrangement, namely the computation of the adjacency graph connecting the vertices of the arrangement. Our prototype is implemented within the framework of the EXACUS project. It is *complete* in the sense that it can handle all kind of inputs including all degenerate ones, where intersection curves have singularities or pairs of curves intersect with high multiplicity. It is *exact* in the sense that it always computes the mathematical correct result. It is *efficient* measured in running times, *i.e.* it compares favorably to the only previous implementation, namely the projection approach by Berberich *et al.* [9].

Chapter 2

NumeriX: Algebraic Foundations for CGAL

2.1 Introduction

The focus in computational geometry was traditionally on linear objects. Even for them, achieving all the three goals: completeness, exactness, and efficiency (the first two imply robustness), is notoriously difficult for the well-known reasons, namely rounding errors of conventional floating-point machine arithmetic, handling of degeneracies, and inherent complexity of sophisticated algorithms.



CGAL, the Computational Geometry Algorithms Library [33, 54], is the state-of-the-art in implementing geometric algorithms completely, exactly, and efficiently. When CGAL was started, the research area of Computational Geometry already had existed for two decades. Every major problem was thoroughly discussed, many different algorithms existed and their theoretical complexity and practical runtime behavior were known. With this long history it was clear how to implement the algorithms for linear geometry robustly and efficiently. CGAL was initiated with the ambitious goal to "*make the large body of geometric algorithms developed in the field of computational geometry available for industrial application.*"

Recently there are efforts to extend CGAL to non-linear objects. Extension packages deal with arrangements of conic arcs, smallest enclosing ellipses, and Apollonius graphs (Voronoi diagrams of circles). And in fact, many geometric

algorithms apply also to non-linear objects. For example Bentley and Ottmann already observed that their sweep-line algorithm works for x-monotone curve segments [4]. But the devil is in the details. There are two main reasons. First, the number of degenerate cases grows heavily when going from straight-line to curved objects. Second, for linear objects almost all predicates, for example the sidedness test, can be realized over the field of rational numbers. For curved objects the coordinates of points are algebraic numbers and therefore comparisons are much more susceptible either to numerical errors or to a loss of efficiency.

As a consequence the situation for non-linear geometry is completely different than it has been for linear geometry. Although research areas like Solid Modeling and Algebraic Geometry consider curved objects for years, in the context of Computational Geometry they have a very short history. At the current state it is not clear which methods will be the best for efficient and robust implementations. Thus the final goal of a stable software library for curved objects is unlikely to be achieved in the first attempt. In particular the development of new and refined number types and concepts made up and will make up an essential part within the research process.

2.1.1 Exacus

For the reasons discussed above we decided to initially outsource our investigations for curved geometry from CGAL and conceived the EXACUS-project (*Efficient and Exact Algorithms for Curves and Surfaces*[1]). EXACUS was founded in April 2002 within the context of the ECG-project (*Effective Computational Geometry for Curves and Surfaces*[2]). We continued our work on EXACUS as part of the ACS-project (*Algorithms for Complex Shapes with certified numerics and topology*[3]). It is a cooperation of seven European research groups in Groningen, ETH Zürich, FU-Berlin, INRIA Sophia-Antipolis, Athens, Tel-Aviv, MPI Saarbrücken and GeometryFactory, the sole distributor of commercial licenses for CGAL.

[1] `http://www.mpi-sb.mpg.de/EXACUS/`
[2] `http://www-sop.inria.fr/prisme/ECG/`
[3] `http://acs.cs.rug.nl/`

The central goal of EXACUS is the development of a demonstrator of a reliable, correct, and efficient CAD geometry kernel. In our implementations, we address the fundamental problem of computing arrangements of algebraic curves and surfaces of small degree with applications to boolean operations on semi-algebraic sets. Although we call our library design prototypical, we spent nonetheless a great effort on completeness, exactness, efficiency, and documentation.

Since April 2002, EXACUS proved to be a good laboratory to improve our ideas for implementations for algebraic curves and surfaces. The theory and the implementations for the different applications behind EXACUS are described in the following series of papers: Based on the LEDA [62] implementation of the Bentley-Ottmann sweep-line algorithm [4] Berberich *et al.* [7] and Eigenwillig *et al.* [29] implemented arrangement computations for conic and cubic arcs in the plane, respectively. Berberich *et al.* [9] extended these techniques to quartic curves that are projections of spatial contour and intersection curves of quadrics. Kerber *et al.* [27, 26] presented an algorithm to compute the planar arrangement induced by segments of arbitrary algebraic curves. And recently Dupont *et al.* [22] presented an implementation to compute the adjacency graph of an arrangement of quadrics in 3-dimensional space. Our implementations are *complete* in the sense that they can handle all kinds of inputs including all degenerate ones, in particular singularities and tangential intersection points. They are *exact* in that they always compute the mathematically correct result. They are *efficient* measured in running times.

EXACUS is organized as a collection of C++ libraries, see Figure 2.1 for their layered architecture. The top layer illustrates the four applications pursued so far: CONIX, CUBIX, and ALCIX compute arrangements of curves in the plane. The QUADRIX library covers our efforts on quadrics. SWEEPX contains our generic sweep-line algorithm suitable for segments of curved arcs and boolean operations based on it. A very important layer is the NUMERIX library providing the algebraic foundations such as a generic number type support, symbolic algebra, and numerical algorithms.

The Library Support layer provides fundamentals, such as configuration and memory management. During configuration we detect a couple of other libraries. However, EXACUS does not depend on these libraries in various core parts of its functionality. For instance we use LEDA or CORE for its number types and

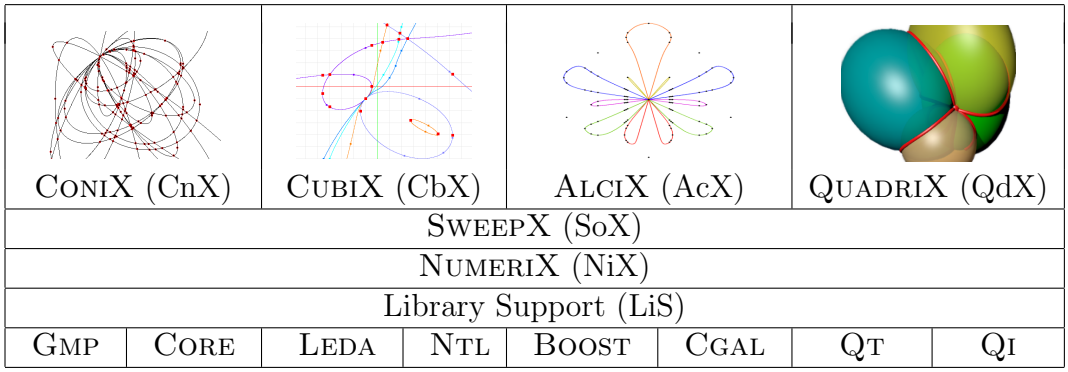


Figure 2.1: Library layers of the Exacus project.

Boost for the interval arithmetic. We optionally use Ntl for its fast implementation of a gcd for polynomials with integer coefficients. We use Qt for the GUI of the demo applications. And since we integrated a core part of the NumeriX library, namely the number type support, into the new Algebraic Foundations package of Cgal, we depend on Cgal for its number type support.

2.1.2 Integration of Exacus into Cgal

In order to profit from these experiences and from synergy effects, the acs partners decided to integrate core parts of Exacus into Cgal. We started the integration process right after the second public release of Exacus in August 2006. For the Cgal release 3.3, we in particular accomplished to improve and integrate the number type support of Exacus into the new Cgal package Algebraic Foundations [44]. This will allow us to integrate further parts of Exacus into Cgal. On the other hand Cgal will in turn serve as the main basis of further projects within Exacus.

The design of Exacus follows the generic programming paradigm with C++ templates. This is similar to the design principles incorporated in the Stl [2] and in Cgal [33, 13]. In particular we use concepts identical to those in the Stl (iterators, containers, functors) and Cgal (geometric traits class, functors).[4]

[4]For an introduction into generic programming with C++ templates see also Section 2.2.1.

Consequently, Exacus and Cgal are in principal compatible enough such that it is possible to integrate mature parts of Exacus into Cgal. However, there are some fundamental differences as well.

In the overall design of Cgal two major layers can be identified. The core part of Cgal is the geometric-kernel layer. The concept of a geometric kernel comprises all basic geometric data types and basic predicates [32, 16]. In particular, it abstracts from the choice of the coordinate system, *i.e.* homogeneous or not, and the used number types. The kernels which are provided by this layer are meant to be the main template argument to the more sophisticated data structures and algorithms in the second layer. This has the advantage that all applications in the second layer can easily choose their favorite kernel from the first layer. They just have to instantiate their code with the kernel class of choice. However, the concepts for the kernels in Cgal have been designed with linear geometry in mind, *i.e.* they are not capable to support curved geometry. As a consequence the Acs partners decided to aim for a *curved kernel* [30], which is on the same layer as the linear kernels, and an *algebraic kernel* in a layer underneath.

By contrast Exacus has no kernel at all, instead the provided tools are just collected within the namespaces of the various libraries. This was necessary in order to stay flexible enough for our ongoing research in this area. Instead we strived to identify the valuable algebraic tools and specified concepts for these tools. For instance the library NumeriX defines a concept for real root isolation and provides several models of this concept. As a consequence it is easy to interchange different methods. However, Exacus is not entirely generic, for instance we are restricted to one particular type representing polynomials. On the other hand we excel in a very sophisticated number type support and a very generic representations of algebraic points and segments in the SweepX layer.

With the second public release of Exacus in August 2006, the acs partners agreed that Exacus is now ready to provide substantial and mature contributions to Cgal. The integration process is organized by Eric Berberich and myself. At this stage I am in particular responsible for the integration of the library NumeriX. So far the integration of NumeriX has led to the following Cgal packages.

- **Algebraic Foundations:**
  This package is already part of Cgal release 3.3, it is the cornerstone for a further integration of Exacus into Cgal. Based on the experience

with the number type support in NumeriX it has lifted the number type concepts to a more abstract level supporting polynomials, finite fields, and algebraic extensions as well. The package is discussed in Section 2.2.

- **Modular Arithmetic:**
  Based on the introduced modular arithmetic the package is capable to support the Modular Filter as I presented in [47]. This package has been submitted for Cgal release 3.4. Moreover, it is planned to extend this generic framework such that it is capable to support, for instance, polynomial gcd computation based on modular methods, as indicated in Section 2.3.

- **Algebraic Kernel:**
  Together with our acs partners we finally agreed on a concept for an algebraic kernel. The specification splits into a univariate and a bivariate algebraic kernel. Beside the involvement into the design of the algebraic kernel concept my personal contribution are in particular the development of a concept for multivariate polynomials which is in fact a independent package. An detailed documentation of the concepts can be found in [8].

  Based on the algebraic tools from NumeriX we implemented a generic univariate kernel. Section 2.4 reports on this implementation. Note that we also plan to provide a bivariate algebraic kernel. However, a core part of the bivariate algebraic kernel is the so called two curve analysis which relates to the SweepX layer of Exacus. This is in the responsibility of Eric Berberich and not reported here.

2.2 Algebraic Foundations

CGAL is targeting towards exact computation with non-linear objects, in particular objects defined on algebraic curves and surfaces. As a consequence types representing polynomials, algebraic extensions and finite fields play a more important role in related implementations. The CGAL package Algebraic Foundations [44] has been introduced to stay abreast of these changes. We consider the package as the cornerstone for a further integration of EXACUS into CGAL.

Our interface extends the former number type support of CGAL [33, 13, 49, 54] which in principal only distinguished *Euclidean Rings* and *Fields*. Moreover, it improves the interface of NUMERIX [6] in the sense that it clearly separates the algebraic structure from other characteristics of a type. Therefore, the package avoids the term *number type*. Instead the package distinguishes between the *algebraic structure* of a type and whether a type is *real embeddable*, that is, whether it can be embedded on the real axis. In this way the framework is capable to maturely support not just ordinary (real embeddable) number types but also other types such as complex numbers, polynomials, algebraic extension fields, and finite fields. Another major contribution of the package is the introduction of the notion of *interoperable* types which allows an explicit handling of mixed operations.

The package strictly follows the generic programming paradigm. A short introduction of this paradigm is given in the next section. The package itself is discussed in the ensuing sections.

2.2.1 Generic Programming

Though our algorithms are designed to be as efficient as possible it is in general unfeasible to provide the optimal implementation for a certain algorithm. Usually the implementation of an algorithm is already antiquated before it is accomplished just due to the fact that there are some new techniques that could improve several subroutines of the algorithm. Therefore, we aim for an implementation that is flexible enough to integrate new methods easily and which is designed such that it can also be reused by other implementations. A way to

reach this goal is the *generic programming* paradigm. Here is one definition of this paradigm as it is given in [51]:

Definition 10. *Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:*

- *Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.*
- *Lifting of a concrete algorithm to a level which is as general as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.*
- *When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.*
- *Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.*

The C++ programming language supports the generic programming paradigm by *class templates* and *function templates*. Templates are incompletely specified components where some types are only identified by formal placeholders. These types are called the *template arguments* to a template. For each instantiation of a template argument the compiler generates a separate translation of the code.

Compared to object-oriented programming, the main advantage of generic programming is its flexibility. In particular polymorphism is available without the restrictions of inheritance which enforces a tight coupling through the inheritance relation ship. Moreover, inheritance implies some performance loss as well. First, each call to a virtual member function implies an additional indirection through

the *virtual function table* [61]. Second, inheritance implies a bit more memory usage, which is due to the so called virtual function table pointer. By contrast templates do not add any overhead due to the separate instantiation of the code. In particular the compiler can separately optimize the code with respect to each given template argument.

Concepts and Models

Of course it is not possible to use any type as template argument to a certain class or function template. Each generic algorithm imposes a set of minimal requirements that a type must fulfill to be correctly used as an argument in a call of this algorithm. Potential requirements are valid expressions, invariants, associated types, and even complexity guarantees.

- *Valid Expressions* are C++ expressions involving the template argument which must compile successfully.
- *Associated Types* are types that are related to the template argument in that they participate in one or more of the valid expressions, *e.g.*, the return type of some required member function.
- *Invariants* are run-time characteristics that must remain valid for all objects of the template argument. These invariants often take the form of pre-conditions and post-conditions.
- *Complexity Guarantees* are maximum limits on how long the execution of one of the valid expressions will take, or how much of various resources its computation will use.

The STL, the C++ Standard Template Library [65], groups these requirements into so called *Concepts* and established the de facto standard for documenting concepts. A type that meets these requirements is called a *Model* of the concept. A concept can extend the requirements of another concept, which is called a *Refinement*.

Note that concepts are not part of the C++ standard yet, that is, the language does not provide any feature to define requirements for template arguments. This is not even possible for the syntactical requirements. They are just implicitly checked during the instantiation and compilation process. In the same way the

runtime characteristics may be checked due to pre-conditions or post-conditions that are placed by hand, *i.e.* by the programmer but not by the compiler. However, there are substantial efforts to integrate *concepts* into the C++ programming language.[5]

Traits and Tags

While writing a class template or function it is often important to know additional information about a template argument, *e.g.* associated types or constants. A so called *Traits* is a class that provides generic access to this information at compile time. For example, the class template `std::iterator_traits<T>` looks something like this:

```
template <class Iterator>
struct iterator_traits {
   typedef ... iterator_category;
   typedef ... value_type;
   typedef ... difference_type;
   typedef ... pointer;
   typedef ... reference;
};
```

For instance, the traits' `value_type` specifies the type the iterator is 'pointing at'. The `iterator_category` specifies the category of the iterator, *e.g.* `forward_iterator`, `random_iterator`, *etc.*. It can be used to select the best algorithm for a certain iterator. This kind of associated type which ist used to indicated a property of a type is a so called *Tag*.

The most important feature of traits classes is that they allow us to associate information to types that are not under our control, *i.e.* build-in types and types defined by third-party libraries. In general this is done by (partially) specializing the traits class template for the particular type.

Adaptable Functor

A *Functor*, or also called *function object*, is an object that can be called as if it is a function. In general this is an object of a class defining the `operator()`. The

[5] `http://www.generic-programming.org/software/ConceptGCC/`

STL knows three basic functor concepts, namely Generator, Unary Function, and Binary Function describing objects that can be called as `f()`, `f(x)` and `f(x,y)`.[6] Functors returning a bool are called *Predicates*, *i.e.* Unary Predicate and Binary Predicate.

A major advantage of function objects is that each object can carry its own state. A simple example is that the function object has a counter counting the number of function calls. A more sophisticated one is, for instance, a functor comparing two multivariate polynomials where the comparison depends on the monomial order which is stored within the function object.

A further, very important feature of functors is that they can provide nested typedefs with additional information which can be indispensable within generic code. In particular it is often important to know the result type or the argument types of a function object. For this purpose the STL has introduced the so called *Adaptable Functor* concepts, which are refinements of the basic functor concepts.

- Adaptable Generator:
  This is a Generator providing its result type as a nested type `result_type`.
- Adaptable Unary Function:
  In the same way, this is a Unary Function providing the nested types `result_type` and `argument_type`.
- Adaptable Binary Function:
  This is a Binary Function providing the `result_type` as well as the types of the `first_argument_type` and `second_argument_type`.

The name *Adaptable* is due to the fact that these functors can be used by so called *Function Object Adaptors*. These are function objects that transform, manipulate or combine other function objects, see also [65].

2.2.2 Algebraic Structures

The algebraic structure concepts introduced within this section are motivated by their well known counterparts in traditional algebra, but we also had to pay

[6] This list can obviously be extended to ternary functions and beyond.

tribute to existing types and their restrictions. To keep the interface minimal, it was not desirable to cover all known algebraic structures, e.g., we did not introduce concepts for such basic structures as *groups* or exceptional structures as *skew fields*.

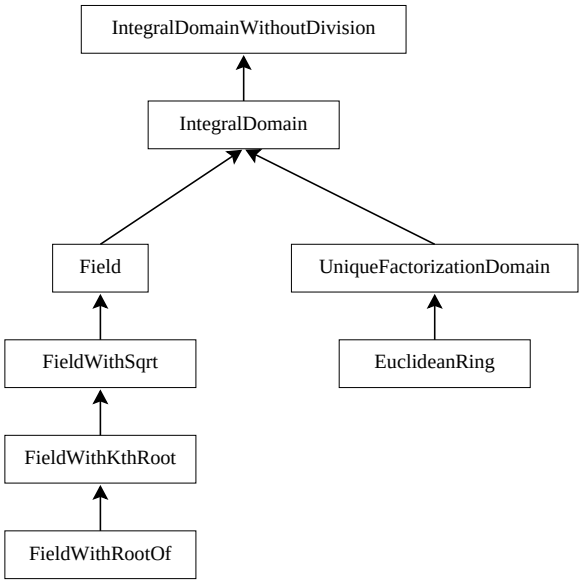


Figure 2.2: Concept Hierarchy of Algebraic Structures

An algebraic structure is at least Assignable, CopyConstructible, DefaultConstructible and EqualityComparable. Moreover, we require that any model of an algebraic structure concept is constructible from `int`.[7] The algebraic structure concepts form a concept hierarchy as it is shown in Figure 2.2. For an exact definition of all concepts we refer to the reference manual of [44].

- **IntegralDomainWithoutDivision:**

  This is the most basic algebraic structure concept introduced so far. A model of the concept IntegralDomainWithoutDivision represents a commutative ring with 0, 1, `+`, and `*`. The ring is required to be free of zero divisors.

[7] This is well defined due to the canonical homomorphism mapping 0 and 1 onto the zero and one of the ring, respectively.

Note that IntegralDomainWithoutDivision in principal corresponds to integral domains in the algebraic sense, the distinction results from the fact that some implementations of integral domains lack the (algebraically always well defined) integral division. However, the concept has the advantage that is very easy to document that an algorithm evades any kind of division by referring to this concept.

- **IntegralDomain:**

  This concept refines IntegralDomain by requiring the missing integral division. Note that the integral division is provided through a separate function and not through the `operator /`, which is reserved for division in a *Field* only. The main advantage is a clear separation from the division with remainder as it is required by the concept EuclideanRing.

- **Field:**

  A *Field* is an *Integral Domain* in which every non-zero element has a multiplicative inverse, that is, an inverse element is defined for any element different from zero. The concept Field requires this division operation to be available through `operator/` and `operator /=`.

  This is further refined by the concepts FieldWithSqrt, FieldWithKthRoot and FieldWithRootOf, by requiring that a model is closed under the operations *'sqrt'*, *'kth root'* and *'extracting the real root of a polynomial'* respectively. Models of these concepts are for instance `leda::real` or `CORE::Expr`.

- **UniqueFactorizationDomain:**

  A UniqueFactorizationDomain is an IntegralDomain with the additional property that the ring it represents is a *unique factorization domain*, that is, any two elements, not both zero, possess a greatest common divisor (gcd).

- **EuclideanRing:**

  EuclideanRing refines UniqueFactorizationDomain. It is a UniqueFactorizationDomain that affords a suitable notion for remainders of divisions. In particular it is possible to use the Euclidean algorithm in order to compute the *gcd* of two elements.

For ease of use and since their semantic is sufficiently standard to presume their existence, the usual arithmetic and comparison operators are required to be realized via C++ operator overloading.

The remaining functionality is gathered within a traits class, namely the class template `Algebraic_structure_traits<AS>`. In particular, all required unary (e.g., sqrt) and binary functions (e.g., gcd, div) must be models of AdaptableUnaryFunction or AdaptableBinaryFunction and local to the traits class (e.g., `Algebraic_structure_traits<AS>::Sqrt()(x)`). This design allows us to profit from all parts in the STL and its programming style and avoids the name-lookup and two-pass template compilation problems experienced with the old design using overloaded functions.

For ease of use and backward compatibility all functionality is also accessible through global functions defined within namespace `CGAL`, e.g., `CGAL::sqrt(x)`. This is realized via function templates using the according functor of the traits class. Table 2.1 lists the supported functions in relation to the algebraic structure concepts.

Remarks

Unfortunately, the name IntegralDomainWithoutDivision is a bit awkward, since it indicates a restriction which is not the case. The alternative was to use IntegralDomain for IntegralDomainWithoutDivision and IntegralDomainWithDivision for IntegralDomain. However, this is out of the frying pan into the fire, since IntegralDomainWithDivision suggests some additional feature on top of the mathematical definition of an integral domain, which is not the case at all.

The concept Field refines IntegralDomain. This is because most ring-theoretic notions like greatest common divisors become trivial for Fields. Hence, we see Field as a refinement of IntegralDomain and not as a refinement of one of the more advanced ring concepts. If an algorithm wants to rely on gcd or remainder computation, it is trying to do things it should not do with a Field in the first place.

Table 2.1: Provided Functions in relation to the Algebraic Structure Concepts

Algebraic Structure Concept	Functions

IntegralDomainWithoutDivision	`operator +, -, *`
	`CGAL::is_zero`
	`CGAL::is_one`
	`CGAL::unit_part`
	`CGAL::simplify`
	`CGAL::square`

IntegralDomain	`CGAL::integral_division`
UniqueFactorizationDomain	`CGAL::gcd`

EuclideanRing	`CGAL::div_mod`
	`CGAL::mod`
	`CGAL::div`

Field	`operator /`
FieldWithSqrt	`CGAL::sqrt`
FieldWithKthRoot	`CGAL::kth_root`
FieldWithRootOf	`CGAL::root_of`

Tags in Algebraic Structure Traits

As mentioned in the previous section, tags are very useful to dispatch between alternative implementations within template code. In particular they can help to select the best algorithm for the associated type at compile time. The class `Algebraic_structure_traits<AS>` provides several tags for this purpose.

`Algebraic_structure_traits<AS>::Algebraic_category` is the most important tag. The tag indicates the most refined algebraic concept the template argument is a model of. The tag is one of:

- `CGAL::Integral_domain_without_division_tag`
- `CGAL::Integral_domain_tag`
- `CGAL::Field_tag`
- `CGAL::Field_with_sqrt_tag`
- `CGAL::Field_with_kth_root_tag`

- `CGAL::Field_with_root_of_tag`
- `CGAL::Unique_factorization_domain_tag`
- `CGAL::Euclidean_ring_tag`
- `CGAL::Null_tag`

`CGAL::Null_tag` is provided in case the type is not a model of an algebraic structure concept at all. The tags are derived from each other such that they reflect the hierarchy of the algebraic structure concept, e.g., `Field_with_sqrt_tag` is derived from `Field_tag`. The usage of this tag is illustrated by the example in Figure 2.3

Further tags are:

- `Algebraic_structure_traits<AS>::Is_exact`
- `Algebraic_structure_traits<AS>::Is_numerical_sensitive`

As the name indicates both are boolean tags, *i.e.*, they are either `CGAL::Tag_true` or `CGAL::Tag_false`. An algebraic structure is considered *exact*, if all operations required by its concept are computed such that a comparison of two algebraic expressions is always correct. An algebraic structure is considered as *numerically sensitive*, if the performance of the type is sensitive to the condition number of an algorithm.

Note that there is really a difference among these two notions, e.g., the fundamental type `int` is not numerical sensitive but considered inexact due to overflow. Conversely, types as `leda_real` or `CORE::Expr` are exact but sensitive to numerical issues due to the internal use of multiprecision floating point arithmetic. We expect that `Is_numerical_sensitive` is used for the dispatching of algorithms, while `Is_exact` is useful to enable assertions that can be check for exact types only.

2.2.3 Real Embeddable

Most number types represent some subset of the real numbers. From those types we expect functionality to compute the sign, absolute value or double approximations. In particular we can expect an order on such a type that reflects the order along the real axis. All these properties are gathered in the concept `RealEmbeddable`. The concept is orthogonal to the algebraic structure concepts,

```
#include <CGAL/basic.h>
#include <CGAL/IO/io.h>
#include <CGAL/Algebraic_structure_traits.h>

template< typename NT > NT unit_part(const NT& x);
template< typename NT >
NT unit_part_(const NT& x, CGAL::Field_tag);
template< typename NT >
NT unit_part_(const NT& x, CGAL::Integral_domain_without_division_tag);

template< typename NT >
NT unit_part(const NT& x){
    // the unit part of 0 is defined as 1.
    if (x == 0 ) return NT(1);

    typedef CGAL::Algebraic_structure_traits<NT> AST;
    typedef typename AST::Algebraic_category Algebraic_category;
    return unit_part_(x,Algebraic_category());
}

template< typename NT >
NT unit_part_(const NT& x, CGAL::Integral_domain_without_division_tag){
    // For many other types the only units are just -1 and +1.
    return NT(int(CGAL::sign(x)));
}

template< typename NT >
NT unit_part_(const NT& x, CGAL::Field_tag){
    // For Fields every x != 0 is a unit.
    // Therefore, every x != 0 is its own unit part.
    return x;
}

int main(){
    // Function call for a model of EuclideanRing, i.e. int.
    std::cout<< "int:    unit_part(-3  ): " << unit_part(-3  ) << std::endl;
    // Function call for a model of FieldWithSqrt, i.e. double
    std::cout<< "double: unit_part(-3.0): " << unit_part(-3.0) << std::endl;
    return 0;
}

// Note that this is just an example
// This implementation for unit part won't work for some types, e.g.,
// types that are not RealEmbeddable or types representing structures that have
// more units than just -1 and +1. (e.g. MP_Float representing Z[1/2])
// This is why Algebraic_structure_traits provides the functor Unit_part.
```

Figure 2.3: This example illustrates a dispatch for Fields using overloaded functions. Note that the example only needs two overloads since the algebraic category tags reflect the algebraic structure hierarchy.

i.e., it is possible that a type is a model of `RealEmbeddable` only, since the type may just represent values on the real axis but does not provide any arithmetic operations.

As for algebraic structures this concept is also traits class oriented. The main functionality, *e.g.* functors and tags required by the concept `RealEmbeddable`, is gathered in the class template `Real_embeddable_traits<T>`. In particular, it provides the boolean tag `Is_real_embeddable` indicating whether a type is a model of `RealEmbeddable`. The comparison operators are required to be realized via C++ operator overloading. All unary functions (e.g. *sign*, *abs*) and binary functions (e.g. *compare*) are models of the concepts `AdaptableUnaryFunction` and `AdaptableBinaryFunction` and are local to `Real_embeddable_traits`.

For ease of use and backward compatibility all functors are also accessible through global functions defined within namespace `CGAL`. This is symmetric to the design of the algebraic structure concepts. The supported functions are:

- `CGAL::is_zero`
- `CGAL::is_positive`
- `CGAL::is_negative`
- `CGAL::sign`
- `CGAL::abs`
- `CGAL::to_double`
- `CGAL::to_interval`[8]
- `CGAL::compare`

Remarks

In case a type is a model of `IntegralDomainWithoutDivision` and `RealEmbeddable` the number represented by an object of this type is the same for arithmetic and comparison. It follows that the ring represented by this type is a superset of the integers and a subset of the real numbers. Hence, the type has characteristic zero. In case the type is a model of `Field` and `RealEmbeddable` it is a superset of the rational numbers.

[8] Conversion to `GAL::Interval_nt`, an interval with boundaries of type `double`.

2.2.4 Interoperability of Types

A further major contribution of the Algebraic Foundations package is the introduction of a notion of *interoperable* types which allows an explicit handling of mixed operations. For this purpose the package introduces two concepts, namely `ImplicitInteroperable` and `ExplicitInteroperable`. We start with the more refined concept `ImplicitInteroperable`, since it is the more intuitive concept.

Implicit Interoperable

In general mixed operations are provided by overloaded operators and functions or just via implicit constructor calls. This level of interoperability is reflected by the concept `ImplicitInteroperable`. However, within template code the result type, or so called coercion type, of a mixed arithmetic operation may be unclear. Therefore, the package introduces the class template `Coercion_traits<A,B>`. First of all the traits indicates the implicit interoperability of `A` and `B` via the boolean tag `Are_implicit_interoperable`. If this is the case, `Coercion_traits<A,B>::Type` is the coercion type for two implicit interoperable types `A` and `B`.

Some trivial examples are `int` and `double` with coercion type double or the exact types `CGAL::Gmpz` and `CGAL::Gmpq` with coercion type `CGAL::Gmpq`. However, the coercion type is not necessarily one of the input types. For instance, the result of the multiplication of a polynomials with integer coefficients with a rational type is supposed to be a polynomial with rational coefficients.

Explicit Interoperable

The concept `ExplicitInteroperable` has been introduced to cover more complex cases for which it is hard or impossible to guarantee implicit interoperability. For two `ExplicitInteroperable` types the `Coercion_traits<A,B>` is required to provide a the functor `Coercion_traits<A,B>::Cast`, which converts an object of type `A` or `B` into the coercion type. Note that this functor is available for `ImplicitInteroperable` types as well since the concept `ImplicitInteroperable` refines `ExplicitInteroperable`. In this context the functor is in particular very useful in order to avoid repeated implicit conversion of the same object.

Remarks

In case two types `A` and `B` are `ExplicitInteroperable` with coercion type `C`, the types `A` and `B` are both valid argument types for all binary functors provided by traits classes `Algebraic_structure_traits<C>` and `Real_embeddable_traits<C>`. This holds for the according global functions as well.

The example in Figure 2.4 illustrates how two write code for explicit and implicit interoperable types. Note that the coercion type is particularly needed to determine the return type of the binary function.

Implementation Details

The default implementation of the class template `Coercion_traits<A,B>` declares `A` and `B` as not interoperable, that is, though two types are implicit interoperable in practice the default traits declares them as not interoperable, in particular it does not provide a valid coercion type. Hence, two types become (officially) interoperable only due to a valid specialization of the traits.

For ordinary types, *i.e.* non class templates, such as built-in types and number types of third party libraries their pairwise interoperability is declared by a whole bunch of total specializations. However, due to the aid of a few macros this is in general just a one-line-statement per pair.

The situation becomes more involved when it comes to compound types, *i.e.*, types which are class templates. Examples for such types are:

- `Polynomial<T>`, where `T` is the type which is used as the coefficient type.
- `Quotient<T>`, where `T` is the type used to represent the numerator and denominator.
- `Complex<T>`, where `T` is the type used to represent the real and the imaginary part, respectively.

Now consider the following example. Let `A` and `B` be two `ExplicitInteroperable` number types, with coercion type `C`. Hence, a user can expect that `Polynomial<A>` and the type `B` are `ExplicitInteroperable` as well, namely with coercion type `Polynomial<C>`. A first solution is to provide the according partial specialization of the traits:

```
#include <CGAL/basic.h>
#include <CGAL/Coercion_traits.h>
#include <CGAL/Quotient.h>
#include <CGAL/Sqrt_extension.h>
#include <CGAL/IO/io.h>

// this is the implementation for ExplicitInteroperable types
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
binary_function_(const A& a , const B& b, CGAL::Tag_false){
    std::cout << "Call for ExplicitInteroperable types: " << std::endl;
    typedef CGAL::Coercion_traits<A,B> CT;
    typename CT::Cast cast;
    return cast(a)*cast(b); // some arithmetic operation
}

// this is the implementation for ImplicitInteroperable types
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
binary_function_(const A& a , const B& b, CGAL::Tag_true){
    std::cout << "Call for ImpicitInteroperable types: " << std::endl;
    return a*b; // some arithmetic operation
}

// this function selects the correct implementation
template <typename A, typename B>
typename CGAL::Coercion_traits<A,B>::Type
binary_func(const A& a , const B& b){
    typedef CGAL::Coercion_traits<A,B> CT;
    typedef typename CT::Are_implicit_interoperable Are_implicit_interoperable;
    return binary_function_(a,b,Are_implicit_interoperable());
}

int main(){
    CGAL::set_pretty_mode(std::cout);

    // Function call for ImplicitInteroperable types
    std::cout<< binary_func(double(3), int(5)) << std::endl;

    // Function call for ExplicitInteroperable types
    CGAL::Quotient<int>           rational(1,3);    // == 1/3
    CGAL::Sqrt_extension<int,int> extension(1,2,3); // == 1+2*sqrt(3)
    CGAL::Sqrt_extension<CGAL::Quotient<int>,int> result;
    result = binary_func(rational, extension);
    std::cout<< result << std::endl;

    return 0;
}
```

Figure 2.4: The `binary_function` expects two `ExplicitInteroperable` arguments. For `ImplicitInteroperable` types a variant that avoids the explicit cast is selected.

```
template <class A,class B>
class Coercion_traits<Polynomial<A>,B >{...};
template <class A,class B>
class Coercion_traits<B,Polynomial<A> >{...};
```

Now consider a similar solution for `Quotient<T>`.

```
template <class A,class B>
class Coercion_traits<Quotient<A>,B >{...};
template <class A,class B>
class Coercion_traits<B,Quotient<A> >{...};
```

Given that `A` and `B` are explicit interoperable the user can expect that the types `Polynomial<A>` and `Quotient<B>` are explicit interoperable as well and that `Polynomial<Quotient<C> >` is the coercion type. But this is not true, instead the instantiation of `Coercion_traits<Polynomial<A>,Quotient<B> >` causes an error since the instantiation results in an ambiguity of the partial specializations:

```
template <class A,class B>
class Coercion_traits<Polynomial<A>,B >{...};
template <class A,class B>
class Coercion_traits<B,Quotient<A> >{...};
```

Of course, it is possible to provide further partial specializations such as:

```
template <class A,class B>
class Coercion_traits<Polynomial<A>,Quotient<B> >{...};
template <class A,class B>
class Coercion_traits<Quotient<A>,Polynomial<B> >{...};
```

But in the long term this naive design, as it was implemented in NumeriX, becomes unmanageable due to a quadratic number of partial specializations for the various pairs of compound types.

The new solution for CGAL takes into account that some compound types are supposed to be more dominant than others. For instance a user can expect that the coercion of a `Polynomial<A>` and `Quotient<B>` results in a type that is representing a polynomial. Therefore, we have introduced a mechanism that is capable to assign a certain priority level to each partial specialization of the traits. First of all, we have introduced an enum to achieve a flexible naming scheme for the various levels.

```
enum COERCION_TRAITS_LEVEL {
    CTL_TOP          = 4,
    CTL_POLYNOMIAL   = 4,
    CTL_COMPLEX      = 3,
```

```
    CTL_INTERVAL      = 2,
    CTL_SQRT_EXT      = 1
};
```

The clou are the following few lines of meta template programming. The first part redirects an ordinary instantiation of `Coercion_traits<A,B>` to the top layer `Coercion_traits_for_level<A,B,CTL_TOP>`. The second part iterates through the layers by recursion. Finally the recursion stops at the lowest level, *i.e.* at `Coercion_traits_for_level<A,B,0>`, which contains the default implementation.

```
template<class A , class B>
struct Coercion_traits :
public Coercion_traits_for_level<A,B,CTL_TOP>{};

template <class A, class B, int i >
struct Coercion_traits_for_level:
public Coercion_traits_for_level<A,B,i-1>{};

// This is the default implementation:
template <class A, class B>
struct Coercion_traits_for_level<A,B,0> {...};
```

In this way, we can easily avoid ambiguity among the different partial specializations of the coercion traits. In the new layout the example for `Polynomial` and `Quotient` from above is defined as follows:

```
template <class A,class B>
class Coercion_traits_for_level<Polynomial<A>,B,CTL_POLYNOMIAL>{...};
template <class A,class B>
class Coercion_traits_for_level<B,Polynomial<A>,CTL_POLYNOMIAL>{...};
template <class A,class B>
class Coercion_traits_for_level<Quotient<A>,B,CTL_QUOTIENT>{...};
template <class A,class B>
class Coercion_traits_for_level<B,Quotient<A>,CTL_QUOTIENT>{...};
```

Note that it is quite easy to extend this scheme. For instance it is possible to add a new layer for matrices, that dominates polynomials by adding `CTL_MATRIX=5` and setting `CTL_TOP=5` as well.

For a short overview of some implicit and explicit interoperable types see Table 2.2 and Table 2.3, respectively. Note that this is really just a small subset of the possible combination. For instance we could have listed all pairs of build-in types or all possible combinations of the number types provided by LEDA, or other third party libraries. However, `CGAL::Coercion_traits` does not introduce

Table 2.2: A short overview of some ImplicitInteroperable types.

A	B		Coercion Type
int	int	↪	int
int	double	↪	double
int	CORE::BigInt	↪	CORE::BigInt
CORE::BigInt	CORE::BigRat	↪	CORE::BigRat
CGAL::Gmpz	CGAL::Gmpq	↪	CGAL::Gmpq
leda::rational	leda::real	↪	leda::real
Lazy_exact_nt<Gmpz>	Lazy_exact_nt<Gmpq>	↪	Lazy_exact_nt<Gmpq>

Table 2.3: A short overview of some ExplicitInteroperable types.

Explicit Interoperable Types A and B ⇒ Coercion Type
Polynomial<int> double ⇒ Polynomial<double>
Polynomial<Polynomial<leda::integer> > Quotient<leda::integer> ⇒ Polynomial<Polynomial<Quotient<leda::integer> > >
Sqrt_extension<Gmpz,Gmpz> > Gmpq ⇒ Sqrt_extension<Gmpq,Gmpz>
Polynomial<CORE::BigRat> Sqrt_extension<CORE::BigInt,CORE::BigInt> ⇒ Polynomial<Sqrt_extension<CORE::BigRat,CORE::BigInt> >

an explicit interoperability among types of two different third party libraries, *e.g.*, leda::rational is not interoperable with CORE::BigInt.

2.2.5 Summary

We have presented the new CGAL package Algebraic Foundations. This package has been successfully submitted to CGAL and is already contained in CGAL release 3.3. Based on the experience with the number type support in NUMERIX it has extended the former number type support of CGAL [33, 13, 49, 54], which in principal distinguished only real embeddable *Euclidean Rings* and *Fields*. The new concepts are more abstract allowing the additional support of polynomials, finite fields, and algebraic extensions. Moreover the package has introduced the notion of *interoperable* types which allows an explicit handling of mixed operations.

We consider this package as the cornerstone for a further integration of EXACUS into CGAL.

Acknowledgment: *I would like to mention Sebastian Limbach, his diligence and accuracy was of invaluable help throughout the entire reorganization of the number type support of* CGAL.

2.3 A Generic Modular GCD

The computation of the gcd of two polynomials is one of the major bottlenecks within all approaches investigated within Exacus so far. This is in particular true for the approach presented in Chapter 1 of this thesis due to the huge coefficient sizes and the fact that nearly all coefficients are defined over an algebraic extension of degree 2.

It is well known that polynomial gcd algorithms based on modular arithmetic are much more efficient than non-modular methods. In 1971 Brown [14] presented a modular GCD algorithm for polynomials in $\mathbb{Z}[x_1, ..., x_n]$. Langemyr and McCallum [57] developed an algorithm to compute the gcd of two univariate polynomials over an algebraic extension. A further improvement was presented by Encarnacion [31].

Regardless of these facts, by release 1.0 of Exacus in August 2006 we were still not using any modular methods for polynomial gcd computation. This section reports on a generic implementation of the above methods for univariate polynomials. Moreover, we have introduced a hybrid approach that incorporates the ideas of Langemyr-McCallum and Encarnacion, which performs slightly better for our purposes.

To the best of our knowledge, there is no other generic open source code available, that supports a modular gcd for polynomials over algebraic extensions. The presented prototypes are implemented within NumeriX and rely heavily on the Cgal package Modular Arithmetic [45], which I submitted for Cgal release 3.4. We see this implementation as a proof of concept for this package as well as for the newly introduced traits classes and functors.

This is joint work with Dominik Hülse an undergraduate student at the Johannes Gutenberg Universität Mainz. In the context of his diploma thesis Dominik has partially implemented and evaluated the algorithms presented in this section.

Outline: The investigated algorithms based on modular methods for polynomials over integers and algebraic extensions are presented in Section 2.3.1 and 2.3.2, respectively. Section 2.3.3 provides details on the implementation and presents the introduced traits classes. In Section 2.3.4 we compare our modular implementations with the naive approach as it was implemented in Exacus so far. Section 2.3.5 will draw conclusions and will outline further work.

2.3.1 Modular GCD in $\mathbb{Z}[x]$

The principal idea is to compute the gcd with respect to several primes and to recover the original gcd in $\mathbb{Z}[x]$ using the Chinese remainder theorem, *e.g.* see Knuth [55]. The major disadvantage of traditional (non-modular) methods is that they can not avoid the exponential coefficient growth in the intermediate steps of the Euclidean Algorithm, whereas the coefficient of the final gcd are in general smaller than the input coefficients. By contrast, the modular methods do not suffer from this effect at all, since the coefficients in the intermediate steps are bounded by the size of the current prime, that is, their size is constant. However, it is important that the modular methods are output sensitive and do not rely on some worst case analysis for the coefficient size in the final gcd, which is exponential [14].

For a given prime $p \in \mathbb{Z}$, let $\mathbb{F}_p$ be the Galois field with p elements and $\phi_p : \mathbb{Z} \to \mathbb{F}_p$ the field homomorphism defined by $\phi_p : x \mapsto (x \mod p)$. The homomorphism from $\mathbb{Z}[x]$ to $\mathbb{F}_p[x]$ induced by ϕ_p will also be denoted as ϕ_p. The homomorphic image of ϕ_p will also be denoted as the *modular image* of some entity.

Let F be some polynomial in $Z[x]$, we will use the following notation:

- $deg(F)$: The degree of F.
- $lc(F)$: The leading coefficient of F.
- $monic(F)$: $F/lc(F) \in \mathbb{Q}[x]$, the monic associate to F.
- $cont(F)$: The content, *i.e.*, the unit normal *gcd* of the coefficients of F
- $pp(F)$: $F/cont(F) \in Z[x]$, the primitive part of F.

We will now outline the univariate variant of the modular GCD algorithm for polynomials in $\mathbb{Z}[x]$ as presented in Brown [14], see Algorithm 5.

In principal the algorithm is a while loop (step 5-13) that computes the gcd with respect to several primes until it is possible to recover the correct gcd using the Chinese Remainder Theorem. However, there are also two essential details that should be kept in mind.

- The Chinese Remainder is only capable to recover coefficients in $\mathbb{Z}$. Hence, it is very important that $\tilde{G} \in \mathbb{F}_p[x]$ represents a polynomial in $\mathbb{Z}[x]$. Therefore, the $gcd(\tilde{F_1}, \tilde{F_2})$, which is computed in step 7, is multiplied by $\tilde{g}$, which is the modular image of $\overline{g} = gcd(lc(F_1), lc(F_2))$. By Gauss's lemma [74],

Algorithm 5 Given the polynomials F_1', $F_2' \in \mathbb{Z}[x]$ compute $G' \in \mathbb{Z}[x]$, the greatest common divisor of F_1' and F_2'.

(1) Set $c_1 = cont(F_1')$, $c_2 = cont(F_2'), c = gcd(c_1, c_2)$
(2) Set $F_1 = F_1'/c_1$, $F_2 = F_2'/c_2$
(3) Set $f_1 = lc(F_1)$, $f_2 = lc(F_2)$, $\overline{g} = gcd(f_1, f_2)$.
(4) Set n = 0, e = min(deg(F_1),deg(F_2))
(5) Select p, a new odd prime not dividing f_1 or f_2
(6) Set $\tilde{g} = \phi(\overline{g})$, $\tilde{F_1} = \phi(F_1)$, $\tilde{F_2} = \phi(F_2)$
(7) Compute $\tilde{G} = \tilde{g} \cdot gcd(\tilde{F_1}, \tilde{F_2}) \in \mathbb{F}_p[x]$.
(8) If deg($\tilde{G}$) = 0, set G = 1, and skip to step (15).
If deg($\tilde{G}$) > e, p is an unlucky prime, so go back to step (5).
If deg($\tilde{G}$) < e, a former prime was unlucky, set n = 0, e = deg($\tilde{G}$)
(9) Set n = n+1
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*).
$(q, G^*) := chinese\_remainder((q, G^*), (p, \tilde{G}))$
(12) If the coefficients of G^* have changed go back to step (5).
(13) If $G^* \nmid \overline{g} \cdot F_1$ or $G^* \nmid \overline{g} \cdot F_2$ go back to step (5)
(14) Set G = pp(G^*)
(15) Set G' = cG, and return.

this is a multiplicative upper bound for $lc(G)$ and guarantees that the recovered coefficients are in $\mathbb{Z}$. Otherwise, $\tilde{G}$ would represent $monic(G)$ which is (in general) a polynomial in $\mathbb{Q}[x]$.

- For some (unlucky) primes it happens that the gcd loses some factors, which implies that the prime divides at least $lc(F_1)$ or $lc(F_2)$. The algorithm discards such primes in step 5.

- For some other (unlucky) primes it may happen that the gcd in $\mathbb{F}_p[x]$ contains additional factors. Unfortunately, it is not possible to discard these primes beforehand. Therefore, the algorithm keeps track of $deg(\tilde{G})$, that is, it incorporates only those primes for which $deg(\tilde{G})$ is minimal. This is done in step 8.

- Algorithm 5 is a variant of the algorithm proposed by Brown [14] in the

sense that it is output sensitive using an idea by Langemyr and McCallum [57]. Instead of computing as many primes as needed to guarantee a correct recovery of the gcd, it checks whether the recovered polynomial G^* becomes stable, that is, if two successive Chinese Remainder steps yield the same polynomial it is in all probability the desired polynomial. Of course this is verified by dividing the input polynomials by the assumed gcd, see step 13. In the paper of Encarnacion [31] this method is called the *trial-division* method.

The remaining steps are just to compute the correct constant factor for G'.

2.3.2 Modular GCD in $\mathbb{Z}(\alpha)[x]$

We shall now consider modular methods for polynomials defined over algebraic extensions. Though our main interest is focused on algebraic extension of degree 2, we discuss the algorithm by Langemyr and McCallum [57] and the improved version by Encarnacion [31] for any degree.

Denominator for Algebraic Integer

Though the algorithms are based on Brown's algorithm for polynomials defined over the integers the crux is that due to the algebraic extension it is not as easy to obtain a multiplicative upper bound for the denominators in $monic(G)$, which is needed for a successful application of the Chinese Remainder.

Lowercase Greek letters will in general denote algebraic numbers, whereas lowercase Italic letters will denote rational numbers. For each α there is a unique polynomial $M \in \mathbb{Z}[t]$ having α as a root, such that M is a primitive, irreducible polynomial with positive leading coefficient. M is called the *minimal polynomial* of α. Given such a polynomial M the field $K = \mathbb{Q}(\alpha)$ is obtained by extending the rationals by any root α of M. The degree of M, that is, the degree of the algebraic extension, will be denoted by $n = deg(M) \geq 2$. Write $res(A, B)$ for the resultant of $A,\ B \in \mathbb{Z}[t]$. The *discriminant* of M is

$$disc(M) = (-1)^{n(n-1)/2} lc(M)^{-1} res(M, M'), \tag{2.1}$$

where M' denotes the derivative of M. In the sequel we will use D to denote $disc(M)$, and d the largest integer whose square divides D.

A number is an *algebraic integer* if it is the root of some monic polynomial with coefficients in $\mathbb{Z}$. Though Encarnacion considers arbitrary extensions, we will restrict ourselves to α being an algebraic integer as discussed in Langemyr and McCallum [57], that is, we assume that $lc(M) = 1$. For α being an algebraic integer we define the integral domain

$$\mathbb{Z}[\alpha] := \{a_0 + a_1\alpha^1 + \cdots + a_{n-1}\alpha^{n-1} \mid a_i \in \mathbb{Z}\}, \tag{2.2}$$

and

$$(1/a)\mathbb{Z}[\alpha] := \{(1/a)\delta \mid \delta \in \mathbb{Z}[\alpha]\}. \tag{2.3}$$

For any $\beta \in K = \mathbb{Q}(\alpha)$, there is a $b > 0$ in $\mathbb{Z}$ such that $\beta \in (1/b)\mathbb{Z}[\alpha]$. For b being as small as possible, b is called the *denominator* of β. For a polynomial $F \in (1/r)\mathbb{Z}[\alpha][x]$, r is called a *multiplicative bound* for the denominators of the coefficients of F.

For $\beta = B(\alpha)$ with $B \in \mathbb{Z}[t]$ we define

$$res(M, \beta) := res(M, B). \tag{2.4}$$

Theorem 11 (Weinberger-Rothschild [83]). *Let α be an algebraic integer and $F \in \mathbb{Q}(\alpha)[x]$. If $G \in \mathbb{Q}(\alpha)[x]$ is a divisor of F over $\mathbb{Q}(\alpha)[x]$ then $monic(G)$ is in $(1/(fd))\mathbb{Z}[\alpha][x]$, where $f = res(M, lc(F))$.*

Corollary 12. *Let α be an algebraic integer and $F_1, F_2 \in \mathbb{Q}(\alpha)[x]$. If G is a common divisor of F_1 and F_2 over $\mathbb{Q}(\alpha)[x]$ then $monic(G)$ is in $(1/bd)\mathbb{Z}[\alpha][x]$, where $b = gcd(f_1, f_2)$ with $f_1 = res(M, lc(F_1))$ and $f_2 = res(M, lc(F_2))$.*

According to [31] it is even possible to strengthen Theorem 11 and Corollary 12 by replacing d by the *defect* of α in case α is an algebraic integer. However, it is known [67] that both entities are not easy to compute. Hence, in practice it is common to use D instead, which is obviously a multiplicative upper bound for d.

Algorithm by Langemyr and McCallum

Let α be an algebraic integer and M its minimal polynomial. For a given prime $p \in \mathbb{Z}$, define $M_p = (M \mod p)$ and $\mathbf{R}_p = F_p[t]/M_p$. Let $\phi_p : \mathbb{Z}[\alpha] \to \mathbf{R}_p$ be the ring homomorphism defined by $\phi_p : B(\alpha) \mapsto (B(t) \mod p) \mod M_p$. The homomorphism from $\mathbb{Z}[\alpha][x]$ to $\mathbf{R}_p[x]$ induced by ϕ_p will also be denoted as ϕ_p.

Algorithm 6 is based on Langemyr and McCallum and it is just a variant of Algorithm 5. In order to remove superfluous constant factors contained in F_1' and F_2' we multiply both polynomials with a normalization factor, such that the leading coefficient is in $\mathbb{Z}$. Thereafter, it is guaranteed that all constant factors are in $\mathbb{Z}$. Step 2 computes and removes these scalar factors for each polynomial. Moreover, this has the side effect that we can set $f_i = lc(F_i) \in \mathbb{Z}$. It remains to compute $D = disc(M)$, the denominator bound for algebraic integers, which depends on the particular algebraic extension (step 3a).

Though the remaining steps look similar to those in Brown's algorithm there is one more matter to adhere to. The gcd which is computed in step 7 is computed in $\mathbf{R}_p[x]$, which may fail since $\mathbf{R}_p$ will in general not be a field, that is, we may need to invert a zero-divisor. However, in this case p is also considered as an unlucky prime and discarded.

Encarnacion's Algorithm

Recall the role of $\overline{g}$ in Algorithm 6. Without this factor the Chinese Remainder tries to recover $monic(G) \in \mathbb{Q}(\alpha)[x]$, which is not possible. Instead the computed polynomial G^* is in $\mathbb{Z}[\alpha][x]$, where each coefficient is just in the same residue class as the corresponding coefficient of $monic(G)$. Having this in mind, Encarnacion [31] proposed to recover the rational coefficients of $monic(G) \in \mathbb{Q}(\alpha)[x]$ using the rational reconstruction algorithm.

Let $n, d \in \mathbb{Z}$ with $d > 0$ and $gcd(n, d) = 1$. Let m be a positive integer satisfying $gcd(m, d) = 1$. Let $u = n/d \mod m$. The rational reconstruction problem is: given u and m find n and d. A solution to this problem without proof was first given by Wang [79], where Wang *et al.* [80] contains the proof. In principal Wang's algorithm is a variant of the extended Euclidean algorithm and outputs n/d provided $|n|$ and d are both less than $\sqrt{m/2}$, that is, $m > 2max(|n|, d)^2$. Though the algorithm may fail in some cases the algorithm succeeds with high probability if m is sufficiently large.

Algorithm 6 (Langemyr - McCallum) Given F_1, $F_2 \in \mathbb{Z}[\alpha][x]$ compute $G \in \mathbb{Z}[\alpha][x]$, the (up to constant factor) greatest common divisor of F_1 and F_2.

(1) Set $F_i = normalization\_factor(lc(F_i)) * F_i$.
(2) Set $F_i = F_i / scalar\_factor(F_i)$.
(3) Set $f_i = lc(F_i) \in \mathbb{Z}$.
(3a) Set $D = denominator\_for\_algebraic\_integers(\mathbb{Q}(\alpha))$.
(3b) Set $\overline{g} = gcd(f_1, f_2)D$.
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$.
(5) Select p, a new odd prime not dividing f_1, f_2 or D.
(6) Set $\tilde{g} = \phi_p(\overline{g}) \in \mathbf{R}_p$, $\tilde{F_i} = \phi_p(F_i) \in \mathbf{R}_p[x]$.
(7) Try to compute $\tilde{G} = \tilde{g} \cdot gcd(\tilde{F_1}, \tilde{F_2}) \in \mathbf{R}_p[x]$.
If this fails, discarded p and go back to step (5).
(8) If $deg(\tilde{G}) = 0$, set $G^* = 1$, and skip to step (14).
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, a former prime was unlucky, set $n = 0$, $e = deg(\tilde{G})$.
(9) Set $n = n + 1$.
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*):
$(q, G^*) := chinese\_remainder((q, G^*), (p, \tilde{G}))$.
(12) If the coefficients of G^* have changed go back to step (5).
(13) If $G^* \nmid F_1$ or $G^* \nmid F_2$ go back to step (5).
(14) Return G^*.

Hence, Encarnacion's algorithm, Algorithm 7 is a naive version of Algorithm 6 that terminates due to an additional Wang step which is applied to G^*. However, the check that a new prime p does not divide D is still required due to the rational reconstruction algorithm.

Though the algorithm by Langemyr and McCallum is output sensitive due to the *trial-division* method, the algorithm of Encarnacion is even 'more' output sensitive. This is due to the fact that $\overline{g}$, as it is used in Algorithm 6, tends to be a very loose upper bound, that is, Algorithm 6 tends to add a lot of superfluous bits to the output. On the other hand Encarnacion's algorithm suffers from the additional costs due to the rational reconstruction, which is quadratic in the

Algorithm 7 (Encarnacion) Given the polynomials F_1, $F_2 \in \mathbb{Z}[\alpha][x]$ compute $G \in \mathbb{Z}[\alpha][x]$, the (up to constant factor) greatest common divisor of F_1 and F_2.

(1) Set $F_i = normalization\_factor(lc(F_i)) * F_i$.
(2) Set $F_i = F_i / scalar\_factor(F_i)$.
(3) Set $f_i = lc(F_i) \in \mathbb{Z}$.
(3a) Set $D = denominator\_for\_algebraic\_integers(\mathbb{Q}(\alpha))$.
(3b) `dispensed` // computation of $\overline{g}$
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$.
(5) Select p, a new odd prime not dividing f_1, f_2 or D.
(6) Set $\tilde{F}_i = \phi_p(F_i) \in \mathbf{R}_p[x]$.
(7) Try to compute $\tilde{G} = gcd(\tilde{F}_1, \tilde{F}_2) \in \mathbf{R}_p[x]$
If this fails, discarded p and go back to step (5)
(8) If $deg(\tilde{G}) = 0$, set $G^\star = 1$, and skip to step (14)
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5)
If $deg(\tilde{G}) < e$, a former prime was unlucky, set $n = 0$, $e = deg(\tilde{G})$
(9) Set $n = n + 1$
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5)
(11) Use Chinese remainder to update the tuple (q, G^*)
$(q, G^*) := chinese\_remainder((q, G^*), (p, \tilde{G}))$
(11a) Try to use Wang's algorithm to compute $(d, G^\star) := wang(q, G^*)$, where $G^\star \in \mathbb{Z}[\alpha][x]$ and $d \in \mathbb{Z}$ the found denominator.
(12) If Wang fails or $G^\star$ has changed go back to step (5)
(13) If $G^\star \nmid F_1$ or $G^\star \nmid F_2$ go back to step (5) // '$\nmid$' denotes pseudo division
(14) return $G^\star$

number of recovered bits [79].

Following the lead of Encarnacion [31] it is not expedient to examine both algorithms using a strict worst case complexity analysis, as this includes the assumption that the algorithm first of all uses all possible unlucky primes, whereas in general almost non of the used primes is unlucky. And since both algorithms are output sensitive, it is more realistic to incorporate the output size in an analysis as well.

The subsequent bounds, which are take from Encarnacion [31], will be expressed in the following parameters:

m $= \max\{deg(F_1), deg(F_2)\}$
n $= deg(M)$, degree of the algebraic extension.
$\mathcal{D}$ $= gcd(f_1, f_2)D$
$\mathcal{M}$ The maximum of the absolute values of the coefficients of the minimal polynomial $M \in \mathbb{Z}[x]$.
$\mathcal{G}$ The maximum of the absolute values of the numerators and denominator in $monic(G) \in Q(\alpha)[x]$ and the cofactors $monic(F_i/G) \in \mathbb{Q}(\alpha)[x]$.

Complexity Langemyr and McCallum:
If we are lucky the algorithm by Langemyr and McCallum (Algorithm 6) computes the gcd of F_1 and F_2 in

$$O(mn\log^2(\mathcal{D}+\mathcal{G}) + m^2n^2(n\log^2(\mathcal{M}) + \log^2(m\mathcal{G}) + \log(\mathcal{D}+\mathcal{G}))) \tag{2.5}$$

word operations. This is assembled by $O(\log(\mathcal{D}+\mathcal{G}))$ modular gcd computations at a cost of $O(m^2n^2)$ and $O(\log(\mathcal{D}+\mathcal{G}))$ Chinese Remainder computations at a cost of $O(mn\log(\mathcal{D}+\mathcal{G}))$. The remaining costs are due to the trial divisor at a cost of $O(m^2n^2(n\log^2(\mathcal{M}) + \log^2(m\mathcal{G})))$, which is assumed to be performed only once.

Complexity Encarnacion:
If we are lucky the algorithm by Encarnacion (Algorithm 7) computes the gcd of F_1 and F_2 in

$$O(m^2n^2(n\log^2(\mathcal{M}) + \log^2(m\mathcal{G})) + mn\log^3(\mathcal{G})) \tag{2.6}$$

word operations. This first term is assembled by $O(\log(\mathcal{G}))$ modular gcd computations at a cost of $O(m^2n^2)$, which is combined with the costs for the trial division. The second term is due to $O(\log(\mathcal{G}))$ applications of the rational reconstruction algorithm to an $O(mn)$ coefficients of size $O(\log(\mathcal{G}))$. This yields a total of $O(mn\log^3(\mathcal{G}))$ since Wang's algorithm [79] is quadratic in the coefficient size.

Note that the costs of Encarnacion's algorithm do not depend on $\mathcal{D}$ at all. However, this is not necessarily advantageous due to the cubic dependency on the coefficient size in G, which is caused by the rational reconstruction algorithm. Hence, we can expect that Encarnacion's algorithm is superior only if $\mathcal{G}$ is small

relative to $\mathcal{D}$. In the special context of this thesis, we can expect that, due to the low degree of the polynomials ($m = 8$) and the low degree of the algebraic extension ($n = 2$), the costs for both algorithms are dominated by the coefficient size. Hence, the algorithm by Langemyr and McCallum is expected to perform better than Encarnacion's algorithm, in particular due to the fact that $\log(\mathcal{G})$ is about $2\log(D)$.

A hybrid approach

The complexity analysis of both algorithms in the previous section shows that both algorithms have a weak point. For Encarnacion's algorithm it is the overhead due to the additional rational reconstruction step which has to be performed in each round. For Langemyr and McCallum the weak point is that $gcd(f_1, f_2)D$ is a very loose upper bound for the denominator of the gcd which causes the use of additional superfluous primes due to the additional bits that have to be recovered. However, during our benchmarks, see also Section 2.3.4, we observed that $gcd(f_1, f_2)$ is a good denominator bound in practice. And indeed, within all our examples, the generated examples as well as those appearing in the benchmarks for the approach presented in Chapter 1, the additional factor D was needed only once.

Our hybrid approach, Algorithm 8, incorporates these observations in the sense that it modifies the algorithm by Langemyr and McCallum by using $gcd(f_1, f_2)$ as the denominator bound. This has the effect that it in general needs $O(log(D))$ fewer rounds to compute the gcd. However, for the unlucky case that D is indeed needed the algorithm would not terminate. Therefore, it uses the rational reconstruction algorithm as well, but only as a fallback. That is, it calls Wang's algorithm only if the fiftieth part of the accumulated time spent within the Chinese remainder exceeds the time spent in the last call of Wang's algorithm. Hence, in practice our hybrid is as output sensitive as Encarnacion's algorithm but, de facto, without the extra costs of the rational reconstruction.

2.3.3 Implementation

We introduced several traits classes to provide the functionality needed by the algorithms presented in section 2.3.1 and 2.3.2. A detailed documentation of the

Algorithm 8 (Hybrid) Given the polynomials F_1, $F_2 \in \mathbb{Z}[\alpha][x]$ compute $G \in \mathbb{Z}[\alpha][x]$, the (up to constant factor) greatest common divisor of F_1 and F_2.

(1) Set $F_i = normalization\_factor(lc(F_i)) * F_i$.
(2) Set $F_i = F_i / scalar\_factor(F_i)$.
(3) Set $f_i = lc(F_i) \in \mathbb{Z}$.
(3a) Set $D = denominator\_for\_algebraic\_integers(\mathbb{Q}(\alpha))$.
(3b) Set $\overline{g} = gcd(f_1, f_2)$.
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$.
(5) Select p, a new odd prime not dividing f_1, f_2 or D.
(6) Set $\tilde{g} = \phi_p(\overline{g}) \in \mathbf{R}_p$, $\tilde{F_i} = \phi_p(F_i) \in \mathbf{R}_p[x]$.
(7) Try to compute $\tilde{G} = \tilde{g} \cdot gcd(\tilde{F_1}, \tilde{F_2}) \in \mathbf{R}_p[x]$.
If this fails, discarded p and go back to step (5).
(8) If $deg(\tilde{G}) = 0$, set $G^* = 1$, and skip to step (14).
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, a former prime was unlucky, set $n = 0$, $e = deg(\tilde{G})$.
(9) Set $n = n + 1$
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5).

(11) Start $timer_{CR}$.
$(q, G^*) := chinese\_remainder((q, G^*), (p, \tilde{G}))$
Stop $timer_{CR}$.
(12) If the coefficients of G^* have changed skip to step (15).
(13) If $G^* \nmid F_1$ or $G^* \nmid F_2$ skip to step (15).
(14) return G^*.
(15) If $timer_{CR} > 50 \cdot timer_W$ go back to step (5).
Reset $timer_{CR}$ and $timer_W$.
(16) Start $timer_W$.
Try $(d, G^\star) := wang(q, G^*)$,
Stop $timer_W$.
(17) If Wang fails or $G^\star$ has changed go back to step (5).
(18) If $G^\star \nmid F_1$ or $G^\star \nmid F_2$ go back to step (5).
(19) return $G^\star$.

traits classes is provided in the appendix of [46].

- **Scalar Factor Traits:**
  The traits class, `Scalar_factor_traits`, is of general interest, since a recurring problem in handling compound types, as polynomial, homogeneous vectors or matrices, is how to remove superfluous factors even though the coefficient type does not provide a *gcd*. Instead, the traits provides access to the *gcd* of the scalar coefficients of a compound type. For instance the scalar factor of a coefficient $a + b\sqrt{c} \in \mathbb{Z}[\sqrt{c}]$ is $gcd(a, b)$.

  Note that the scalar factor traits is only applicable in case the innermost coefficient type, that is, the scalar type, is a unique factorization domain.

- **Chinese Remainder Traits:**
  The traits class, `Chinese_remainder_traits`, provides access to the Chinese remainder algorithm. The traits class is designed such that it is applicable to compound types that can be decomposed into scalar coefficients in the sense of the `Scalar_factor_traits`, i.e. the actual algorithm is applied to each scalar coefficient. For example, a Chinese remainder for a polynomial calls Chinese remainder for each coefficient and returns the resulting polynomial.

  Note that the scalar factor traits is only applicable in case the innermost coefficient type, that is, the scalar type, is an Euclidean ring.

- **Wang Traits:**
  The traits class, `Wang_traits`, provides access to Wang's algorithm. The design of the traits class is symmetric to the `Chinese_remainder_traits`.

- **Algebraic Extension Traits:**
  The traits class, `Algebraic_extension_traits`, provides the functionality related to algebraic extensions. In particular it provides functors for $normalization\_factor$ and $denominator\_for\_algebraic\_integers$.

2.3.4 Benchmarks

To analyse the impact of the bitsize of the coefficients, we generated two families with growing bitsize. One family with integer coefficients (INT_B) and one family with algebraic extensions of degree 2 (EXT_B). Each instance in the family

contains 50 pairs. Each pair is composed of three factors, the gcd of degree 1 and the two co-factors of degree 7. The factors are random polynomials in the sense that their diced scalar coefficients have the desired bitsize. It is guaranteed that the cofactors are coprime.

Since the degree of the gcd has an impact on the length of the polynomial remainder sequence and thus on the runtime of the algorithms, we generated two more families with a growing degree of the gcd. One family for integer coefficients (INT_D) and a second with algebraic extensions of degree 2 (EXT_D). Again, each instance in the family contains 50 pairs which are generated as discussed in the previous paragraph. Each pair is composed of three factors, the gcd with the desired degree and the two co-factors. The bitsize of the coefficients is fixed to 1000 bits for each factor. Note that for these instance $\mathcal{D}$ is about as large as $\mathcal{G}$, that is, we can expect that the algorithm by Langemyr and McCallum performs better than Encarnacion's algorithm. However, the instances reflect the situation as it appears in the quadric intersection approach, see Chapter 1.

The benchmarks were measured on a Pentium(R) M processor 1.7 GHz with 512 KB cache under Linux and the GNU C++ compiler v3.4.6 with optimizations (`-O3`) and disabled assertions (`-DNDEBUG`).

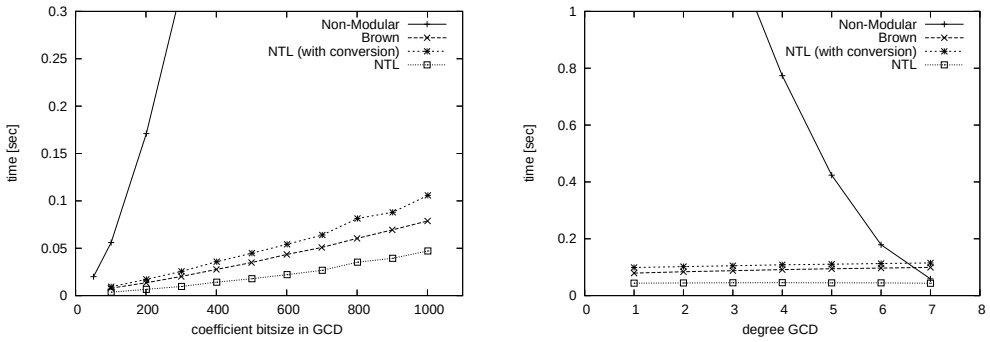


Figure 2.5: Brown's algorithm compared to the non modular method in NumeriX and the gcd provided by the NTL. The left and the right plot show timings for family (INT_B) and (INT_D), respectively.

Figure 2.5 shows the comparison of our implementation of Brown's algorithm with the old non-modular implementation of NumeriX. Moreover we have compared our gcd to the algorithm provided by the well known library for number

theory NTL [73] by Victor Shoup.

- The left plot compares the algorithms on the random instances with growing bits (INT_B). Brown's algorithm performs far better than the old, non-modular implementation, whereas the NTL implementation performs only twice as good as our generic approach. The curve shown in the plot is the pure runtime of the gcd computation within the NTL. However, we also have to pay additional costs for the transformation to NTL number types. This has the consequence that our own implementation seems preferable, at least for our small univariate polynomials with large coefficients.

- The right plot shows that the modular methods are almost independent from the degree of the gcd, whereas the non-modular method profits from a short polynomial remainder sequence (PRS) while the degree of the gcd becomes higher. This is due to the fact that for a short PRS the non-modular method does not suffer from coefficient growth whereas the modular methods use the same number of primes and have to pay about the same cost for the modular image and the Chinese remainder.

As a consequence, we modified our modular implementation such that it switches to the traditional non-modular method in case the length of the PRS is smaller than 2.

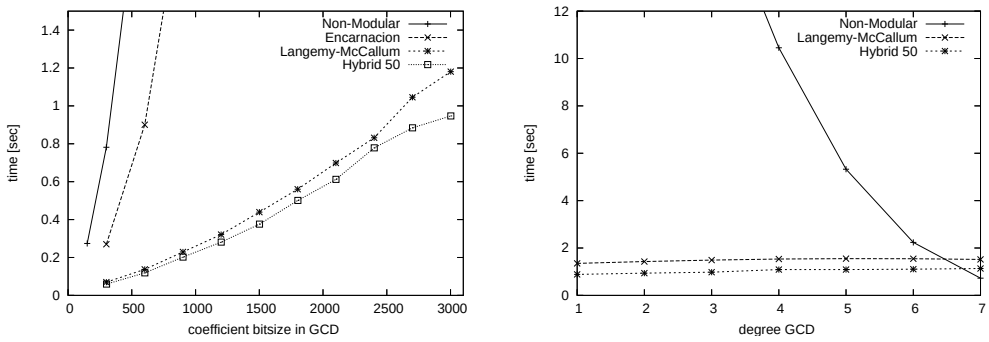


Figure 2.6: The Hybrid approach compared to our implementations of the algorithms by Langemyr/McCallum and Encarnacion and the old non-modular implementation of NumeriX. The left and the right plot shows timings for family (EXT_B) and (EXT_D), respectively.

Figure 2.6 shows the comparison of our hybrid approach compared to our implementations of the algorithms by Langemyr/McCallum and Encarnacion and the old non-modular implementation of NumeriX. As expected the left plot (EXT_B) shows that the modular methods perform far better than the old non-modular implementation. Moreover, our hybrid approach outperforms the algorithms of Langemyr-McCallum and Encarnacion. Again, the right plot (EXT_D) shows that it is better to use the non-modular method for short polynomial remainder sequences.

Note that we expect some further improvement for Algorithm 7 and Algorithm 8, since the current implementation of Wang's algorithm does not take advantage of the known multiplicative denominator bound, as it is indicated in [63].

2.3.5 Summary

We presented a generic implementation computing the gcd of univariate polynomials. In case of integer polynomials our generic implementation is competitive to the one provided by the NTL [73]. In case of polynomials over algebraic extensions of degree 2 we presented a hybrid approach which comprise the vantages of Langemyr-McCallum [57] and Encarnacion [31]. To the best of our knowledge, there is no other generic open source code available, that supports a modular gcd for polynomials over algebraic extensions.

All algorithms have been implemented within EXACUS, *i.e.* NUMERIX. However, due to the integration of the most fundamental parts of EXACUS, namely the package on Algebraic Foundations [44] and the package on Modular Arithmetic [45], it should be straight forward to integrate this code into CGAL as well. In particular, we propose to integrate the new traits classes into the current framework of CGAL. However, there are some redundancies in the design, e.g. the traits classes related to scalar factor, Chinese remainder and Wang's algorithm could be combined into one traits class related to scalar coefficients.

It is obvious that we should aim for a multivariate implementation as it is presented in [14], since the coefficient growth, *i.e.* the degree of the polynomials, in the multivariate polynomial remainder sequence has to be avoided as well. The principal idea is to evaluate the multivariate polynomial in all but one variable at several supporting points and to compute the univariate gcd at these points.

Thereafter, the multivariate gcd is reconstructed using these results. Hence, the implementation of the multivariate gcd should be rather independent from the used coefficient type once the univariate gcd is provided.

Acknowledgment: *We thank Arno Eigenwillig, who had made insightful comments on the treatment of ring extensions in EXACUS, they helped to get this work started.*

2.4 Algebraic Kernel

The CGAL algebraic kernel specification splits into a univariate and a bivariate algebraic kernel. We provide a generic implementation of an algebraic kernel within CGAL, being a model of the concept AlgebraicKernel_1, see Berberich *et al.* [8] for the documentation of the concept.

For a univariate algebraic kernel the most important components are the root isolation method and the implementation of the number type representing real algebraic numbers, that is, real roots of univariate polynomials. Following the generic programming paradigm we have implemented a generic univariate algebraic kernel. The implementation is generic, in the sense that it is possible to exchange the root isolation method and the representation of the algebraic real numbers. Note, that the kernel can be considered as generic in the coefficient type as well, since this type is deduced from the template arguments.

The class template `CGAL::Algebraic_kernel_1` takes two template arguments. The first is supposed to be a model of the concept Isolator and providing the root isolation method. The second is supposed to be a model of the concept AlgebraicRealRep. This allows to substitute the internal representation of the class `Algebraic_real_1` provided by the kernel. Both concepts are documented in the appendix of Hemmer-Limbach [48].

In the sequel we will in particular discuss different implementation variants of the number type representing the real algebraic numbers. Thereafter, we give a brief overview on the currently available models of the concept Isolator. Finally we report on benchmarks obtained using the different instantiations of this kernel.

2.4.1 Algebraic Reals

Each model of the concept AlgebraicKernel_1 is required to provide a public type `Algebraic_real_1`. This type is supposed to represent coordinates of algebraic points, as they, e.g., appear within the arrangement computation of algebraic curves. `Algebraic_real_1` is required to be a model of the concept `RealComparable`, whereas arithmetic operations are not required.

Our kernel has implemented the type `Algebraic_real_1` as a handle class pointing to an internal representation class. This class is a template argument to our kernel and must be a model of the concept AlgebraicRealRep. The representation class stores a univariate square-free polynomial P and an open interval $I = (lower, upper)$ isolating exactly on root, namely the represented algebraic real. It is guaranteed that the polynomial is not zero at the endpoints of the interval. The most important functionalities as refinement and comparison are taken from this class.

- *refine:* A call of this function refines the isolating interval in the sense that it halves the size of the interval leastwise. This may be done by evaluating the sign of the polynomial P at the mid point of the interval. If we observe a sign change with respect to the sign of P at $lower$ the root is in the lower half of the interval. Otherwise, the represented root is in the upper half of the interval. Note that this approach is possible due to the fact that P is square-free. In case the polynomial evaluates to zero an explicit representation of the root and a linear factor of P is found.
- *compare:* In general two algebraic reals are compared using their isolating intervals. However, if the intervals overlap the algebraic reals may be equal. This is the case if the gcd of the two defining polynomials has a root within the intersection of the two defining intervals. Otherwise, both numbers are refined until the intervals do not overlap anymore. Note that the gcd computation may lead to a non trivial factor for both polynomials.

As discussed in Section 2.3 an gcd computation may be very expensive. Therefore, we have introduces several optimizations trying to avoid unnecessary gcd computations.

- **Modular filter:** Even though the two intervals of two algebraic reals may overlap it is often the case that they are not equal. Hence, the gcd computation in this case is unnecessary. To avoid this computation we compute the resultant of both polynomials using modular arithmetic, that is, with respect to one prime. If the result is not zero, we can guarantee that the gcd is constant. Otherwise, we compute the gcd. This filter is base on the Cgal package on Modular Arithmetic [45].
- **Common factor propagation:** Whenever we compute a non-trivial gcd we use this factor to reduce the degree of the defining polynomial, that is,

we replace the polynomial by the gcd or the corresponding cofactor. In this way we avoid or at least reduce the cost of further gcd computations for a specific algebraic real.
Of course it is vantages that all algebraic reals defined by one polynomial profit from this information. Therefore, we have implemented the class `Algebraic_real_1` such that it keeps a list of all number defined on the same polynomial, which is used to propagate a known factor. This list is initialized within the `Solve_1`[9] functor of the algebraic kernel.
This feature is in particular useful for algebraic reals used within the arrangement computation since all real roots of a resultant are needed, see also [7, 29, 9] for more details.

We have implemented two models of AlgebraicRealRep, namely the classes `Algebraic_real_rep` and `Algebraic_real_rep_bfi`. In principal, both classes follow the design as discussed above.

The only difference is that the class `Algebraic_real_rep_bfi` uses multiprecision floating point interval arithmetic to evaluate the sign of the defining polynomial. We use an adaptive approach. An `Algebraic_real_rep_bfi` keeps an approximation of the defining polynomial of a certain precision. This approximation is used to evaluate the sign of the polynomial at some value, *e.g.* the mid point of the interval. If this is not sufficient the precision of the approximation is doubled. If this is not enough the polynomial is evaluated using exact methods. Note that the used precision at most doubles within each sign evaluation.

2.4.2 Root Isolators

Isolating the real roots of a univariate polynomial is the core of the `Solve_1` functor provided by the algebraic kernel. Therefore, we have introduced the Isolator concept allowing a generic access to miscellaneous root isolation approaches. So far we have integrated two isolators into Cgal.

- `CGAL::Descartes`
  This class has been integrated from the NumeriX library of Exacus and implements the Descartes Method as presented in Collins-Akritas [18].

[9] This functor constructs all `Algebraic_real_1` defined by one univariate polynomial.

- `CGAL::Bitstream_descartes`
  This class has been integrated from the NumeriX library of Exacus. Coefficients are converted to (potentially infinite) bit-streams. Thereafter, a variant of Descartes algorithm is used to determine the isolating intervals for the roots. The major advantage of this approach is that it is adaptive, *i.e.*, it uses only as many bits as needed to isolate the roots. Details can be found in Eigenwillig *et al.* [28].

2.4.3 Benchmarks

Due to the modular design of the algebraic kernel it was very easy to obtain the benchmarks for the different models of Isolator in combination with the different models of AlgebraicRealRep.

To analyze the dependency on the bit-size, we generated two families of random univariate polynomials with integer coefficients (INT) and coefficients of algebraic extensions of degree two (EXT). Both families consist of ten instances, where each instance contains 50 polynomials of degree 12. The instances differ in the increasing bit size of their polynomials. In order to complicate the isolation and refining process, we constructed the polynomials such that they have several roots within the interval $[-10, 10]$, that is, we computed the product of six parabolas, where each parabola has at least one root within the interval $[-10, 10]$. Hence, all polynomials have 12 real roots.

We benchmarked three important functionalities of AlgebraicKernel_1:

- `Solve_1`:
  This is the core functor of `CGAL::Algebraic_kernel_1` and provides several operators to construct algebraic reals from a univariate polynomial. For this benchmark we used the most general operator, that also returns the multiplicity of the roots. Hence, the presented benchmarks include the time for:

  - detecting that the polynomial is square free using the modular filter
  - isolating the roots using the entrusted Isolator.
  - constructing, *i.e.* initializing, the algebraic reals

- *compare*:
  All obtained roots were sorted using `std::sort`. Note that the `operator<` is available due to the fact that the concept AlgebraicKernel_1, requires `Algebraic_real_1` to be a model of `RealComparable`.

- *refine*:
  In order to benchmark the refinement method we converted each algebraic real to `double`, which triggers a refinement to a precision of at least 53 bits.

The benchmarks were measured on a Pentium(R) M processor 1.6 GHz with 512 KB cache under Linux and the GNU C++ compiler v3.4.6 with optimizations (`-O3`) and disabled assertions (`-DNDEBUG`).

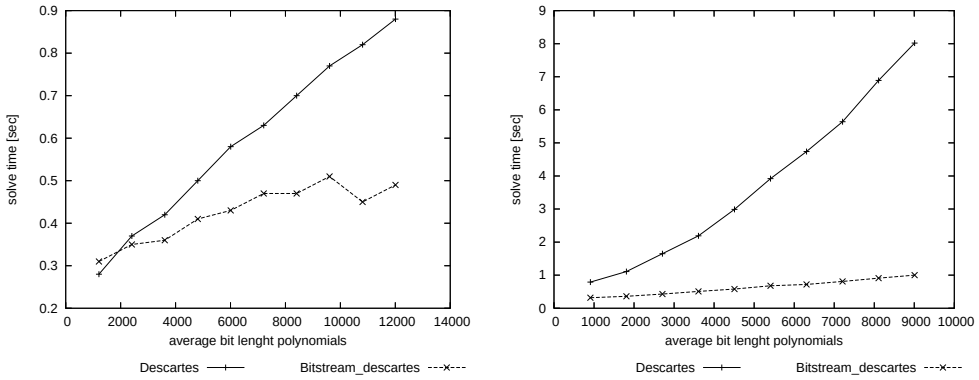


Figure 2.7: Benchmarks of solve method based on the different Isolator models. The left plot and the right plot show the benchmarks for the families (INT) and (EXT), respectively.

Figure 2.7 shows, that bitstream approach performs better than the traditional Descartes method. However, it shows that `CGAL::Bitstream_descartes` has a constant overhead with the consequence that the `CGAL::Descartes` performs better for small coefficients. In case of algebraic extensions of degree 2, bitstream approach is significantly better than the traditional Descartes Method. It turned out that the solve is independent from the two given AlgebraicRealRep models. Hence, we only show the curves obtained by using `Algebraic_real_rep`.

Figure 2.8, shows that the sort also depends on the used Isolator. It seems that the current implementation of `CGAL::Descartes`, refines the roots a bit

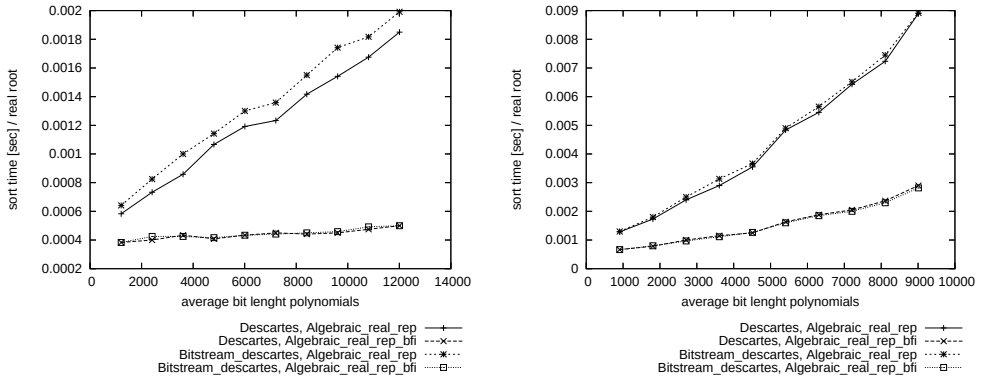


Figure 2.8: Benchmarks of the compare function for the different instantiations of the algebraic kernel. The left plot and the right plot show the benchmarks for the families (INT) and (EXT), respectively.

more than it is supposed to. Therefore, it gives a certain advantage to its roots within the sort. Apart from that it is obvious that the `Algebraic_real_rep_bfi` class performs far better than the class `Algebraic_real_rep`. This is due to the fact that the refinement method of `Algebraic_real_rep_bfi` is widely decoupled from the bit size of the coefficients. This is affirmed by Figure 2.9, since it shows that the refine method of `Algebraic_real_rep_bfi` can be considered as independent from the bit size of the coefficients .

Figure 2.10 shows the total time of all measured operation and for all combinations. As expected, the `CGAL::Bitstream_descartes` in combination with `Algebraic_real_rep_bfi` is clearly ahead of the others.

2.4.4 Summary

We presented an implementation of a generic univariate algebraic kernel within CGAL. Our design clearly separates the major tools, namley the root isolation method and the representation of the algebraic real numbers. As a consequence it is very easy to integrate other approaches, *i.e.* new root isolation methods. However, it is clear that this is not the end of our investigations, for instance the current refine method for the algebraic reals should be improved further since

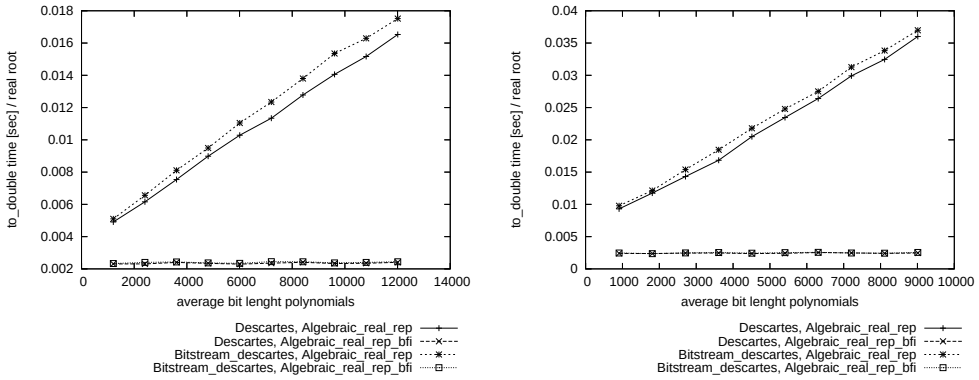


Figure 2.9: Benchmark of the refine method. The left plot and the right plot show the benchmarks for the families (INT) and (EXT), respectively.

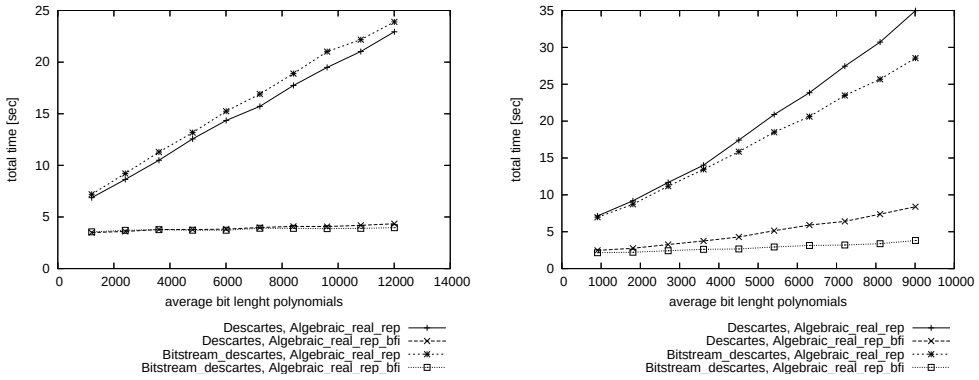


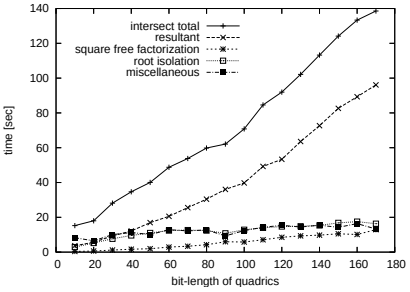
Figure 2.10: Total time for all benchmarked operations. The left plot and the right plot show the benchmarks for the families (INT) and (EXT), respectively.

we currently just half the interval within each refinement step. It is planned to integrate a quadratic refinement as indicated by Abbott [1]. So far this has been postponed, since this is not a bottleneck of the approach presented in Chapter 1.

2.5 Summary and Further Work

So far the integration of Exacus into Cgal has made substantial progress. With the new Cgal packages Algebraic Foundations [44] and Modular Arithmetic [45] we have laid the cornerstone for a further integration of Exacus into Cgal. This is in particular proved by the easy creation the new AlgebraicKernel_1 using existing tools from NumeriX. Of course this is not the end of our efforts and we are confident that the integration of Exacus into Cgal will result in further useful packages. Beside the obvious ones, such as a package on multivariate polynomials or a two dimensional algebraic kernel we would like to point out another idea based on modular arithmetic.

2.5.1 Modular Aided Lazy Evaluation



The plot to the right shows the detailed benchmark results for the intersection of a smooth quartic with a third quadric on the random instances with growing bits, see also Section 1.9.3. For these instances it is very likely that all appearing resultant polynomials are square free. This is checked by the modular filter within the square free factorization. The root isolation of univariate polynomials, which uses the Bitstream Descartes Method by Eigenwillig *et al.* [28], is almost independent of the bit-size of the coefficients of the polynomial. This also holds for the new `Algebraic_real` classes, which are filtered by MPFI arithmetic, see also [48]. Therefore, the only computation that still depends on the bit-size is the computation of the resultant polynomial and the conversion of the coefficients to modular arithmetic within the modular filter. Hence, as the bit size increases the time spent within the resultant computation finally dominates the overall computation time. However, this is in fact a complete waste of time. The only information needed is that the resultant is square free and the leading bits used within the Bitstream Descartes.

In order to detect that the resultant is square free, we could compute its modular image using modular arithmetic in the first place. Thereafter, we could use MPFI arithmetic to generate the bits for the Bitstream Descartes on demand. However,

in order to be effective we would have to apply this parallel scheme to each level of the algorithm, *e.g.* within the class representing the algebraic numbers. Hence, we consider this approach as too laborious and error-prone.

Therefore, it seems appropriate to integrate all these effective filtering techniques into one number type. The principal idea is not new, in order to avoid or delay exact computations the number type is implemented such that an object remembers the way it was constructed by storing the history of operations in a *directed acyclic graph* (DAG) [3]. Note that the literature may also refer to this graph as the *expression tree* of the number [62, 52]. When a sign evaluation is performed the expression is first of all evaluated using interval arithmetic. If the interval contains zero the DAG is used to evaluate the value using some exact type. CGAL provides a class template implementing such a lazy number type called `Lazy_exact_nt<NT>`. It is parameterized by an exact type used to perform the exact computations when needed. The current implementation of `Lazy_exact_nt<NT>` uses intervals based on double arithmetic but this could be easily replaced by MPFI arithmetic. However, this alone would not gain any additional benefit since it is not clear how many bits one should use for the MPFI arithmetic.

The new idea is to initially evaluated the DAG using modular arithmetic, that is, the DAG is evaluate with respect to one prime. If the result is not zero, we can conclude that the exact value is not zero as well. In this case we use MPFI arithmetic with increasing precision until the sign of the interval is unambiguous. Otherwise, it is very likely that the exact value is zero as well. This is due to the fact that we use a fairly large prime of about 26 bits[10]. Hence, in this case we immediately evaluate the DAG using the exact number type.

Within the context discussed above this should be exactly what we need. Due to the fact that each number would know its modular image the modular filter could immediately check that the polynomial is square free. Hence, we would save all the time for computing the exact coefficients of the resultant and the conversion of these coefficients to modular arithmetic. Thereafter, the MPFI arithmetic could be used to generated the bits for the Bitstream Descartes on demand. Note that the class `Algebraic_real` would profit in the same way.

However, there are some open problems as well.

[10]The number of bits is due to the internal use of double arithmetic, for more details see [12].

- If this scheme is applied to an exact integer type the modular arithmetic can support addition, subtraction, multiplication and integral division only. In particular the modular arithmetic is not able to support a division with remainder. In this case the division would trigger an exact evaluation of the DAG. For instance the computation of the content of a polynomial may become a very costly operation. Hence, we may have to redesign some parts of the code. In particular those parts that try to remove superfluous constant factors.
- It may be even more appropriate to apply this scheme to rational numbers, since the modular arithmetic supports exactly the same arithmetic operations, *i.e.* both types model a field. This could be an advantage for the generic situations but may cause additional costs in the degenerate ones.
- It is not possible to extend this idea to other arithmetic operations as `sqrt, sin, cos,` *etc.*. However, it should be possible to integrate types such as `Sqrt_extension` into this scheme.

Depending on the experience with this scheme it could be worthwhile to think of further optimizations. In principal it is better to decouple and regroup the exact and the filtered computations. For instance the best way to compute some entity using interval arithmetic may differ from the way it is computed using the exact number type. Moreover, a regrouping would decrease the number of nodes in the DAG, which in itself would gain a better performance.

In the case of interval arithmetic and the `Lazy_exact_nt` CGAL has already lifted the scheme from the number type level to the geometric level and introduced the *Lazy Exact Kernel*, as presented by Pion *et al.* [66]. The kernel incorporates two kernels, an exact kernel and an inexact kernel which is based on interval arithmetic. Each predicate (or construction) is implemented such that it first of all uses the predicate from the inexact kernel. If this fails it calls the exact one. It should be possible to integrate the idea of a modular filter within this scheme as well. As an example take the orientation predicate for three points in the plane. First of all the modular arithmetic checks that the points are not collinear. In this case the predicate is answered using only MPFI arithmetic. Otherwise, the exact version of the predicate is called.

Bibliography

[1] J. Abbott. Quadratic interval refinement for real roots. Poster presented at the 2006 Int. Symp. on Symb. and Alg. Comp. (ISSAC 2006).

[2] M. H. Austern. *Generic Programming and the STL.* Addison-Wesley, 1998.

[3] M. O. Benouamer, P. Jaillon, D. Michelucci, and J. M. Moreau. A "lazy" solution to imprecision in computational geometry. In *Proc. Fifth Canadian Conf. Computational Geometry*, pages 73–78, 1993.

[4] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.

[5] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS, Efficient and Exact Algorithms for Curves and Surfaces.

[6] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS: Efficient and exact algorithms for curves and surfaces. In *13th Annual European Symposium on Algorithms*, pages 155–166. Springer, 2005.

[7] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *ESA 2002, LNCS 2461*, pages 174–186, 2002.

[8] E. Berberich, M. Hemmer, M. Karavelas, and M. Teillaud. Revision of interface specification of algebraic kernel. Technical Report ACS-TR-243301-01, ACS, Saarbrücken, Germany, 2007.

[9] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, and N. Wolpert. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *Proc. 21. Annual Symposium on Computational Geometry*, pages 99–106, New York, NY, USA, 2005. ACM Press.

[10] H. Bieri and W. Nef. A sweep-plane algorithm for computing the volume of polyhedra represented in boolean form. *Linear Algebra and its Applications*, 53:69–97, 1983.

[11] T. J. Bromwich. *Quadric forms and their classification by means of invariant factors.* Cambrige Tracs in Mathematics and Mathematical Physics, 1906.

[12] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[13] H. Brönnimann, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming—Proceedings of a Dagstuhl Seminar*, LNCS 1766, pages 206–217. Springer-Verlag, 2000.

[14] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18:478–504, 1971.

[15] W. S. Brown. The subresultant PRS algorithm. *ACM Trans. Math. Softw.*, 4(3):237–249, 1978.

[16] H. Brönnimann, A. Fabri, G. J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Schirra, and S. Pion. 2D and 3D kernel. In CGAL Editorial Board, editor, *CGAL-3.2 User and Reference Manual*. CGAL, 2006.

[17] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. 2nd GI Conf. on Automata Theory and Formal Languages*, volume 6, pages 134–183. LNCS, Springer, Berlin, 1975. Reprinted with corrections in: B. F. Caviness and J. R. Johnson (eds.), Quantifier Elimination and Cylindrical Algebraic Decomposition, 85–121. Springer, 1998.

[18] G. E. Collins and A.-G. Akritas. Polynomial real root isolation using Descartes' rule of sign. In *SYMSAC*, pages 272–275, 1976.

[19] T. Culver, J. Keyser, M. Foskey, S. Krishnan, and D. Manocha. ESOLID - A system for exact boundary evaluation. *Computer-Aided Design (Special Issue on Solid Modeling)*, 36, 2003.

[20] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes Comput. Sci.*, pages 314–324, 1993.

[21] L. Dupont. *Paramétrage quasi-optimal de l'intersection de deux quadriques : théorie, algorithme et implantation.* Thèse d'université, Université Nancy II, October 2004.

[22] L. Dupont, M. Hemmer, S. Petitjean, and E. Schömer. Complete, exact and efficient implementation for computing the adjacency graph of an arrangment of quadrics. In *Proc. 15th Annual European Symposium on Algorithms*, Eilat, Israel, October 2007.

[23] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics I: The generic algorithm. accepted, 2007.

[24] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics II: A classification of pencils. accepted, 2007.

[25] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics III: Parameterizing singular intersections. accepted, 2007.

[26] A. Eigenwillig and M. Kerber. Exact and efficient 2D-arrangements of arbitrary algebraic curves. In *Proc. 18th ACM-SIAM Sympos. Discrete Algorithms*, 2007. To appear.

[27] A. Eigenwillig, M. Kerber, and N. Wolpert. Fast and exact geometric analysis of real algebraic plane curves. In Christopher W. Brown, editor, *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, pages 151–158, Waterloo, Ontario, Canada, July 2007. ACM, ACM.

[28] A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, and N. Wolpert. A descartes algorithm for polynomials with bit-stream coefficients. In Viktor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing: 8th International Workshop, CASC 2005*, volume 3718 of *Lecture Notes in Computer Science*, pages 138–149, Kalamata, Greece, 2005. Springer.

[29] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Exact, efficient, and complete arrangement computation for cubic curves. *Computational Geometry 35(1-2)*, pages 36–73, 2006.

[30] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards and open curved kernel. In *Proc. 20th Annu. Sympos. Comput. Geom.*, pages 438–446, 2004.

[31] M. J. Encarnacion. Computing GCDs of polynomials over algebraic number fields. *J. Symbolic Computation*, 20:299–313, 1995.

[32] A. Fabri, G. J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[33] A. Fabri, G. J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. *Softw. – Pract. and Exp.*, 30(11):1167–1202, 2000.

[34] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The Design and Implementation of Planar Maps in CGAL. *The ACM Journal of Experimental Algorithmics*, 5, 2000. Also in LNCS Vol. 1668 (WAE '99), pages 154–168.

[35] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, and implementation. In *ESA 2003, LNCS 2832*, pages 654–666. Springer, 2003.

[36] M. Greenberg and J. Harper. *Algebraic Topology: A First Course*. Benjamin-Cummings, Reading, MA, 1981.

[37] P. Hachenberger. *Boolean Operations on 3D Selevctive Nef Complexes: Data Structure, Algorithms Optimized Implementation, Experiments and Applications*. PhD thesis, Universität des Saarlandes, December 2006.

[38] P. Hachenberger and L. Kettner. Boolean operations on 3D selective Nef complexes: Optimized implementation and experiments. In Leif Kobbelt and Vadim Shapiro, editors, *ACM Symposium on Solid and Physical Modeling (SPM 2005)*, pages 163–174, Cambridge, MA, USA, 2005. ACM.

[39] P. Hachenberger and L. Kettner. 2D boolean operations on nef polygons embedded on the sphere. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. CGAL, 3.3 edition, 2007.

[40] P. Hachenberger and L. Kettner. 3D boolean operations on nef polyhedra. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. CGAL, 3.3 edition, 2007.

[41] D. Halperin. Arrangements. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 21, pages 389–412. CRC Press LLC, Boca Raton, FL, 1997.

[42] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford Science Publications, 1979.

[43] H. Hasse. *Vorlesungen über Zahlentheorie*. Springer, 2nd edition, 1964.

[44] M. Hemmer. Algebraic foundations. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. CGAL, 3.3 edition, 2007.

[45] M. Hemmer. CGAL package for modular arithmetic operations. Technical Report ACS-TR-243406-01, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, GERMANY, 2007.

[46] M. Hemmer and D. Hülse. Traits classes for polynomial gcd computation over algebraic extensions. Technical Report ACS-TR-241405-03, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, GERMANY, 2007.

[47] M. Hemmer, L. Kettner, and E. Schömer. Effects of a modular filter on geometric applications. Technical Report ECG-TR-363111-01, MPI für Informatik, Saarbrücken, Germany, 2004.

[48] M. Hemmer and S. Limbach. Benchmakrs on a generic univariate algebraic kernel. Technical Report ACS-TR-243306-03, MPI für Informatik, Saarbrücken, Germany, 2007.

[49] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th Workshop on Algorithm Engineering (WAE'01)*, LNCS 2141, pages 76–91, Arhus, Denmark, August 2001. Springer-Verlag.

[50] W. Hodge and D. Pedoe. *Methods of Algebraic Geometry*. Cambridge University Press, 1953. Volumes I and II.

[51] M. Jazayeri, R. Loos, and D. R. Musser, editors. *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1 1998*, volume 1766 of Lecture Notes in Computer Science. Springer, 2000.

[52] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. Sympos. Comput. Geom.*, pages 351–359, 1999.

[53] L. Kettner. Reference counting in library design—optionally and with union-find optimization. In A. Lumsdaine and S. Schupp, editors, *Library-Centric Software Design (LCSD'05)*, pages 1–10, San Diego, CA, USA, October 2005. Department of Computer Science, Texas A&M University.

[54] L. Kettner and S. Näher. Two computational geometry libraries: LEDA and CGAL. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Disc. and Comput. Geom.*, pages 1435–1463. CRC Press, second edition, 2004.

[55] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

[56] T. Y. Lam. *The Algebraic Theory of Quadratic Forms*. W. A. Benjamin, MA, 1973.

[57] L. Langemyr and S. McCallum. The computation of polynomial GCD's over an algebraic number field. *J. Symbolic Computation*, 8:429–448, 1989.

[58] S. Lazard, L. M. Peñaranda, and Sylvain Petitjean. Intersecting quadrics : An efficient and exact implementation. In *Proc. 20th ACM Symposium on Computational Geometry*, Brooklyn, NY, June 2004.

[59] J. Levin. Algorithm for drawing pictures of solid objects composed of quadratic surfaces. *Commun. ACM*, 19(10):555–563, October 1976.

[60] J. Levin. Mathematical models for determining the intersections of quadric surfaces. *Comput. Graph. Image Process.*, 11:73–87, 1979.

[61] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, May 1996.

[62] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[63] M. Monagan. Maximal quotient rational reconstruction: an almost optimal algorithm for rational reconstruction. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 243–249. ACM Press, 2004.

[64] B. Mourrain, J. P. Técourt, and M. Teillaud. Sweeping of an arrangement of quadrics in 3D. *Computational Geometry: Theory and Applications*, 30:145–164, 2005. Special issue, 19th European Workshop on Computational Geometry.

[65] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide, second edition: C++ programming with the standard template library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[66] S. Pion and A. Fabri. A generic lazy evaluation scheme for exact geometric computations. In *In Proc. 2nd Library-Centric Software Design*, pages 75–84, 2006.

[67] M. E. Pohst. *Computational Algebraic Number Theory, DMV Seminar Band 21.* Springer-Verlag, 1993.

[68] A. Requicha and H. Voelcker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(1):9–24, 1982.

[69] F. Rouillier. Solving zero-dimensional systems through the rational univariate representation. *Journal of Applicable Algebra in Engineering, Commuinication and Computing 9(5)*, pages 433–461, 1999.

[70] E. Schömer, J. Reichel, T. Warken, and C. Lennerz. Efficient collision detection for curved solid objects. In Kunwoo Lee and Nicholas M. Patrikalakis, editors, *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, pages 321–328, Saarbrücken, Germany, 2002. Association of Computing Machinery (ACM), ACM.

[71] E. Schömer and N. Wolpert. An exact and efficient approach for computing a cell in an arrangement of quadrics. *Computational Geometry: Theory and Applications*, 33:65–97, 2006. Special Issue on Robust Geometric Applications and their Implementations.

[72] C. Segre. Studio sulle quadriche in uno spazio lineare ad un numero qualunque di dimensioni. *Mem. della R. Acc. delle Scienze di Torino*, 36(2):3–86, 1883.

[73] V. Shoup. NTL: A library for doing number theory.
http://www.shoup.net/ntl/.

[74] V. Shoup. *A computational introduction to number theory and algebra.* Cambridge Univ. Press, 2005.

[75] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations.* Academic Press, New York, NY, 1991.

[76] F. Uhlig. Simultaneous block diagonalization of two real symmetric matrices. *Linear Algebra and Its Applications*, 7:281–289, 1973.

[77] F. Uhlig. A canonical form for a pair of real symmetric matrices that generate a nonsingular pencil. *Linear Algebra and Its Applications*, 14:189–209, 1976.

[78] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[79] P. S. Wang. A p-adic algorithm for univariate partial fractions. *Proceedings of SYMSAC '81, ACM Press*, pages 212–217, 1981.

[80] P. S. Wang, M. J. T. Guy, and J. H. Davenport. p-adic reconstruction of rational numbers. *SIGSAM Bulletin, 16, No 2*, 1982.

[81] K. Weierstrass. Zur Theorie der bilinearen und quadratischen Formen. *Monatshefte Akademie der Wissenschaften, Berlin*, pages 310–338, 1868.

[82] R. Wein. High level filtering for arrangements of conic arcs. In *ESA 2002, LNCS 2461*, pages 884–895, 2002.

[83] L. P. Rothschild P. J. Weinberger. Factoring polynomials over algebraic number fields. *ACM Transactions on Mathematical Software*, 2:335–350, December 1976.

[84] N. Wolpert. *An Exact and Efficient Algorithm for Computing a Cell in an Arrangement of Quadrics*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2002.

[85] D. Y. Y. Yun. On square-free decomposition algorithms. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 26–35, New York, NY, USA, 1976. ACM Press.