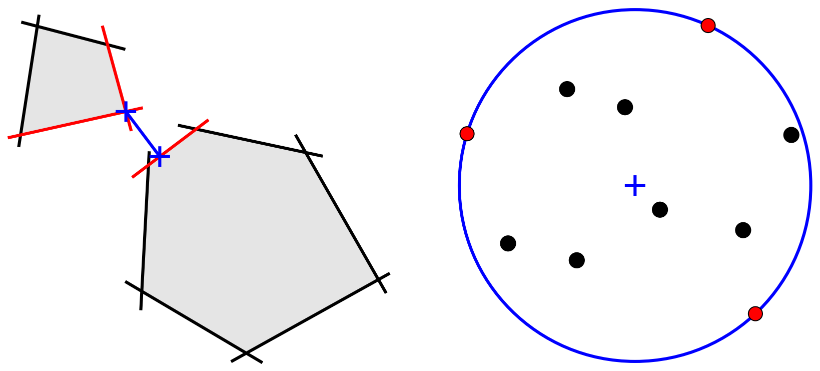
Diss. ETH No. 14738

Quadratic Programming in Geometric Optimization: Theory, Implementation, and Applications



Sven Schönherr

2002

Diss. ETH No. 14738

Quadratic Programming in Geometric Optimization: Theory, Implementation, and Applications

DISSERTATION

submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH, SWITZERLAND

for the degree of
DOCTOR OF TECHNICAL SCIENCES

presented by
SVEN SCHÖNHERR
Diplom-Mathematiker, Freie Universität Berlin, Germany
born on March 17th, 1968, in Berlin, Germany
citizen of Germany

accepted on the recommendation of
Prof. Dr. Emo Welzl, examiner
Prof. Dr. Thomas Liebling, co-examiner
Dr. Bernd Gärtner, co-examiner

2002

Für Marc-Daniel,
unseren Sonnenschein

Acknowledgments

I want to thank all people who have contributed by some means or other to the success of this thesis.

First and foremost among these is Bernd Gärtner, who supported and guided me during the whole time of my studies. He gave me valuable insights in the area of geometric optimization and always kept me on the right track in countless discussions. The subject of this thesis was proposed by him, and, without his ongoing encouragement and strong support, the completion of this work would not have been possible. My sincere thanks go to him.

I am deeply indebted to Emo Welzl for giving me the opportunity to work in his research groups in Berlin and Zurich. I learned a lot from him, and his elegant way of presenting mathematical problems and their solutions has always inspired me.

I am very grateful to Thomas Liebling for agreeing to be a co-examiner. His valuable comments helped improving the presentation.

The collegial atmosphere of the research group at ETH made my time in Zurich a big pleasure. Special thanks go to Christoph Ambühl, Udo Adamy, Falk Tschirschnitz, Zsuzsa Lipták, and Brigitte Kröll for the opportunities to balance work and leisure.

Let me express my best thanks to Helmut Alt, who enabled me to continue my scientific work in Berlin after Emo Welzl moved to Zurich. I owe many thanks to my colleagues at Freie Universität Berlin for the pleasurable atmosphere, especially to Christian Knauer, Hannah Heinrich, Alexander Wolff, Frank Hoffmann, Klaus Kriegel, and Carola Wenk.

One of the highlights during my studies was the participation in the development of CGAL. I am grateful to the whole CGAL team for the very interesting and successful cooperation. My special thanks go to Andreas Fabri, Geert-Jan Giezemann, Lutz Kettner, and Stefan Schirra for the intensive discussions on C++ design and software engineering.

I really enjoyed the cooperation with the partners of the two application projects. In particular, Hans-Jakob Lüthi, Artan Boriçi, Claude Diderich, Phillipe Schiltknecht, and Sébastien Barbey in the portfolio optimization project, and Udo Warschewske, Arno Schleich, and Wlad Oganowski in the stylus calibration project.

The constant support of my family has been fundamental for the success of this thesis. I thank them all, especially my wife, Martina.

Abstract

Many geometric optimization problems can be formulated as instances of linear or quadratic programming. Examples are the polytope distance problem, the smallest enclosing ball problem, the smallest enclosing annulus problem, and the optimal separating hyperplane problem. The resulting linear and quadratic programs share the property, that either the number of variables or the number of constraints is small. This is different from the usual setting in operations research, where both the number of variables and the number of constraints are large, while the matrix representing the constraints is sparse.

We present a solver for quadratic programming problems, which is tuned for applications in computational geometry. The solver implements a generalization of the simplex method to quadratic programs. Unlike existing solvers, it is efficient if the problem is dense and has a small number of variables or a small number of constraints.

A natural generalization of the smallest enclosing ball is the smallest enclosing ellipsoid. This problem is an instance of convex programming which cannot be formulated as a quadratic program. We derive explicit formulae for the primitive operations of a randomized method by Welzl in dimension $d = 2$. Compared to previous solutions, our formulae are simpler and faster to evaluate, and they only contain rational expressions, allowing for an exact solution.

In recent years the focus of interest in the field of computational geometry shifted from the design to practical aspects of the developed algorithms and data structures. One prominent exponent of the new focus is CGAL, the Computational Geometry Algorithms Library. It is a software library of geometric objects, data structures and algorithms written in C++. We implemented our new quadratic programming solver as part of CGAL. Based on it, we realized solutions for the first three geometric optimization problems mentioned above. The implementations are carefully designed to be efficient and easy to use at the same time. The distinguishing features of the solver are almost no overhead when used to solve linear programming problems, user-specifiable program representations, and a combination of exact and floating point

arithmetic. This results in a provably correct and usually very efficient implementation, which is underpinned by the experimental results.

The practical usefulness is shown by two applications in financial business and medicine, where our method solved 'real-world' problems. The quadratic programming solver was used to solve a large portfolio optimization problem as part of a joint project with a big Swiss bank (actually, we showed that the problem was infeasible). In another cooperation with a German company developing medical navigation systems, the smallest enclosing annulus implementation was used to calibrate the tip of medical probes.

We showed in this thesis, that quadratic programming can be used for efficient solutions of some geometric optimization problems and their applications. We developed a solver for linear and quadratic programs, which is tuned for the problems we were interested in. Its realization in CGAL gives always correct results and is very efficient in most practical cases.

Kurzfassung

Viele geometrische Optimierungsprobleme können als Instanzen von Linearem oder Quadratischem Programmieren formuliert werden. Beispiele sind das Problem des Abstands zweier Polytope, das Problem der kleinsten umschliessenden Kugel, das Problem des kleinsten umschliessenden Kugelrings und das Problem der optimal separierenden Hyperebene. Die resultierenden Linearen und Quadratischen Programme haben als gemeinsame Eigenschaft, dass entweder die Anzahl der Unbekannten oder die Anzahl der Nebenbedingungen klein ist. Dies unterscheidet sie vom üblichen Szenario im Bereich Operations Research, wo die Anzahl der Unbekannten und die Anzahl der Nebenbedingungen gross sind, während die Matrix der Nebenbedingungen dünn besetzt ist.

Wir präsentieren einen Löser für Quadratisches Programmieren, der auf Anwendungen in der Algorithmischen Geometrie zugeschnitten ist. Der Löser ist eine Verallgemeinerung der Simplex Methode auf Quadratische Programme. Im Gegensatz zu bereits existieren Lösern ist er effizient, wenn das Problem dicht besetzt ist und eine kleine Anzahl von Unbekannten oder eine kleine Anzahl von Nebenbedingungen hat.

Eine natürliche Verallgemeinerung der kleinsten umschliessenden Kugel ist das kleinste umschliessende Ellipsoid. Dieses Problem ist eine Instanz von Konvexem Programmieren, welche sich nicht als Quadratisches Programm formulieren lässt. Wir leiten explizite Formeln für die Basisoperationen eines randomisierten Algorithmus von Welzl in zwei Dimensionen her. Verglichen mit vorherigen Lösungen sind unsere Formeln einfacher und schneller auswertbar. Sie enthalten nur rationale Ausdrücke und ermöglichen so eine exakte Lösung.

In den letzten Jahren hat sich der Interessenschwerpunkt in der Algorithmischen Geometrie vom Design zu den praktischen Aspekten der entwickelten Algorithmen und Datenstrukturen verschoben. Ein prominenter Vertreter des neuen Schwerpunkts ist CGAL, die Computational Geometry Algorithms Library. Dies ist eine in C++ geschriebene Softwarebibliothek von geometrischen Objekten, Datenstrukturen und Algorithmen. Wir haben unseren neuen Löser für Quadratisches

Programmieren als Teil von CGAL implementiert. Darauf aufbauend haben wir Lösungen für die ersten drei der oben genannten geometrischen Optimierungsprobleme realisiert. Die Implementierungen sind sorgfältig entworfen, um zugleich effizient und einfach benutzbar zu sein. Die herausragenden Merkmale des Lösers sind fast keine Zusatzkosten beim Lösen von Linearen Programmen, eine durch den Benutzer spezifizierbare Repräsentierung des Problems und eine Kombination aus Fliesskomma- und exakter Arithmetik. Daraus ergibt sich eine beweisbar korrekte und in den meisten Fällen sehr effiziente Implementierung. Dies wird durch die experimentellen Ergebnisse unterstrichen.

Die Praxistauglichkeit wird anhand zweier Applikationen in den Bereichen Finanzen und Medizin gezeigt, wo unsere Methode Probleme aus der „realen Welt“ gelöst hat. Der Löser für Quadratische Programme wurde zum Lösen eines grossen Wertpapier-Optimierungsproblems verwendet (genauer gesagt haben wir gezeigt, dass das Problem keine Lösung besass). Im Rahmen einer Zusammenarbeit mit einer deutschen Firma, die ein medizinisches Navigationssystem entwickelt, haben wir die Lösung zur Berechnung des kleinsten umschliessenden Kugelrings zum Kalibrieren der Spitze von medizinischen Instrumenten eingesetzt.

Wir haben in dieser Arbeit gezeigt, dass Quadratisches Programmieren zur effizienten Lösung von einigen geometrischen Optimierungsproblemen und ihren Anwendungen verwendet werden kann. Wir haben einen Löser für Quadratische Programme entwickelt, der auf die betrachteten Probleme zugeschnitten ist. Seine Realisierung in CGAL liefert immer korrekte Ergebnisse und ist in den meisten praktischen Fällen effizient.

Contents

Chapter 0

Introduction

Computational Geometry is the subfield of algorithm design that deals with the design and analysis of solutions for problems involving geometric objects. Over the past two decades an enormous progress was made in the development of efficient geometric algorithms and data structures. However, many of these techniques have not found their way into practice yet. An important cause for this is the fact that the correct implementation of even the simplest of these algorithms can be a notoriously difficult task [66].

There are two particular aspects that need to be dealt with to close the gap between the theoretical results of computational geometry and practical implementations [86], namely the *precision problem* and the *degeneracy problem*.

Theoretical papers assume exact arithmetic with real numbers. The correctness proofs of the algorithms rely on the exact computations, thus replacing exact arithmetic by imprecise built-in floating point arithmetic does not work in general. Geometric algorithms in particular are sensitive to rounding errors since numerical data and control flow decisions have usually strong interrelations. The numerical problems may destroy the combinatorial or geometric consistency of the algorithm, thus resulting in a program crash, in an infinite loop, or in unpredictable erroneous output.

Often, theoretical papers exclude degenerate configurations in the input. Typically, these degeneracies are specific to the problem and would involve the treatment of special cases in the algorithm. Simple examples of configurations considered as degenerate are duplicate points in a point set or three lines intersecting in one point. For some problems, it is not difficult to handle the degeneracies, but for others the special case treatment distracts from the solution of the general problem and it can amount to a considerable fraction of the coding effort.

In theory, this approach of excluding degeneracies from consideration is justified by the argument that degenerate cases are very rare in the set of all possible inputs over the real numbers, i.e. they are very unlikely if the input set is randomly chosen over the real numbers. Another argument is that it is first of all important to understand the general case before treating special cases.

In practice, however, degenerate inputs do occur quite frequently. For instance, the coordinates of the geometric objects may not be randomly chosen over the real numbers, but lie on a grid. For example, they may be created by clicking in a window in a graphical user interface. In some applications, what are called degeneracies are even high-valued design criteria. In architecture, for example, features of buildings do align on purpose. As a consequence, practical implementations usually must address the handling of degeneracies.

Besides the precision problem and the degeneracy problem, advanced algorithms bring about the additional difficulty that they are frequently hard to understand and hard to code.

In recent years the focus of interest in the field of computational geometry shifted from the design to practical aspects of the developed algorithms and data structures. One prominent exponent of the new focus is the CGAL[1] project [78]. The mission statement of the project (and its successors GALIA[2] and ECG[3]) is to

> *make the large body of geometric algorithms developed in the field of computational geometry available for industrial application.*

[1]Constructing a Geometric Algorithms Library
[2]Geometric Algorithms for Industrial Applications
[3]Effective Computational Geometry for Curves and Surfaces

The key tool in reaching this goal is CGAL, the *Computational Geometry Algorithms Library*. It was developed during the first two projects by several universities and research institutes in Europe and Israel. Since then, the library is (and will be in the future) actively maintained and extended.

I joined the CGAL team in January 1995. As one of the five members of the kernel design group [30], I influenced the design of the geometric kernel and the overall design of the library. My contributions to the basic library lie in the area of *geometric optimization*. In particular, I implemented two versions of Welzl's algorithm [105] for computing the smallest enclosing circle and the smallest enclosing ellipse, respectively, of a finite set of points in the plane. The latter implementation uses the exact primitives derived in Chapter 4. It is joint work with Bernd Gärtner, who wrote the underlying conic class [43, 44]. Further contributions are implementations of geometric optimization problems based on the new quadratic programming solver presented in this thesis as well as the implementation of the solver itself.

0.1 Geometric Optimization

Many geometric optimization problems can be formulated as instances of *linear programming* (LP) or *quadratic programming* (QP), where either the number of variables n or the number of constraints m is small. LP is concerned with minimizing a linear function subject to linear (in)equality constraints, while QP deals with convex quadratic objective functions of the form

$$c^T x + x^T D\, x,$$

with D being a positive semi-definite matrix. Examples are the following:

Polytope distance. Given two polytopes P and Q, defined by a total of k vertices (or k halfspaces) in d-dimensional space, find two points $p \in P$ and $q \in Q$ with smallest Euclidean distance. For fixed d, an $O(k)$ solution follows from the fact that this problem can be formulated as an *LP-type problem*[4] in the sense of [64]. It has explicitly been addressed by Wolfe for the special case where one polytope is defined by a single point [107] and by Sekitani and Yamamoto in the general case [94]. The latter algorithm applies the technique Welzl has used for the smallest enclosing ball problem [105] (see below), while the former works in a simplex-like fashion. In fact, when specialized to the situation in Wolfe's paper, our algorithm for solving QP problems is an implementation of his method. A related problem, namely that of finding the nearest point in a simplicial cone, has been studied by Murty and Fathi in [71]. The polytope distance problem – mostly occurring in collision detection – immediately fits into the QP framework.

Smallest enclosing ball. Given k points in d-dimensional space, find the ball of smallest volume containing all the points. This is a classical problem of computational geometry, with many applications, for example in bounding volume heuristics. For fixed dimension d, $O(k)$ algorithms are known [105], and efficient implementations exist [40, 19]. The smallest enclosing ball problem is a special case of a certain class of optimal containment problems [27, 35], which can be formulated as QP problems. A concrete formulation as a quadratic program together with a proof of its correctness is given in Chapter 3.

Smallest enclosing annulus. Given k points in d-dimensional space, find the annulus (region between concentric spheres with radii R and r, $R \geq r$) that contains the points and minimizes the difference $R^2 - r^2$. For $d = 2$, this is the annulus of smallest area, and in any dimension, the optimal annulus can be used to test whether the input points lie approximately on a sphere. Roundness tests in general have received some attention recently [60]. The problem is also of theoretical importance, because it can be used in an approximation algorithm for the minimum-width annulus, a much harder problem [2]. The smallest

[4]The abstract class of LP-type problems as introduced by Sharir and Welzl [64] is a generalization of the class of LP problems. An LP-type problem is a pair (H, w) with H being a set of 'constraints' and w being an 'objective function', where each subset $G \subseteq H$ gets assigned a value $w(G)$ with the following properties: Let $F \subseteq G$, then $w(F) \leq w(G)$ (*monotonicity*) and if $w(F) = W(G)$ and $w(G) < w(G \cup \{h\})$ for some $h \in H \backslash G$, then also $w(F) < w(F \cup \{h\})$ (*locality*). A basis of $G \subseteq H$ is an inclusion-minimal subset B of G such that $w(B) = w(G)$. To solve the problem means to find $w(H)$ and a basis of H. A linear program is a special case of an LP-type problem.

enclosing annulus problem is an LP problem, but it can of course also be considered as QP with a 'degenerate' objective function, having $D = 0$.

Optimal separating hyperplane. Given two point sets P and Q with a total of k points in d-dimensional space, test whether they can be separated by a hyperplane, and if so, find a separating hyperplane that maximizes the distance to the nearest point. The separation itself can be done by LP, but finding the optimal hyperplane is QP. The problem can be reduced to the polytope distance problem.

It is common to all four problems mentioned above, that d is usually small compared to k, often even constant ($d = 2, 3$). In order to solve them efficiently, this has to be taken into account. Like in case of LP, existing solvers for QP are not tuned for this scenario. They usually work in the *operations research* setting, where both the number of constraints m and the number of variables n are large, but the matrix representing the constraints is sparse. Of course, such solvers can be used to address problems in our scenario, but a performance penalty is unavoidable in this case.

The scenario for which we developed the QP solver is the one where $\min(n, m)$ is small. In the geometric optimization problems, this value is closely related to the dimension of the space the problem lives in. The value $\max(n, m)$, on the other hand, may be very large — it usually comes from the number of objects (points, halfspaces, etc.) that define the problem. Moreover, we assume no sparsity conditions, i.e. both matrices A and D may be dense. Namely, the geometric optimization problems we want to handle typically exhibit a dense structure; when we deal with point sets in Euclidean space, for example, the concept of sparsity can even be meaningless. Many properties of point sets are translation invariant, in which case coordinates with value zero play no special role at all.

Let us briefly discuss why 'standard' solvers (addressing the case where both m and n may be large and the problem is sparse) cannot efficiently be used in our scenario.

First of all, if n is large and the objective matrix D is dense, the problem cannot even be entered into solvers which require the $\Theta(n^2)$ nonzero entries of D to be explicitly given. This situation for example occurs in connection with the smallest enclosing ball problem. There is an alternative formulation in which D is sparse, but in return, the constraint matrix A will get larger. Our solver can handle implicitly given matrices D, and if m is small, only few entries will ever be evaluated.

Furthermore, the performance of standard solvers in practice crucially depends on the number of nonzero entries in the problem description, rather than on n and m (which is exactly what one wants for large, sparse problems). In our scenario, this is inappropriate, because in the dense case, no advantage is taken from the fact that one of the parameters is small.

It is well-known that the simplex method for LP can be generalized to deal with QP problems. For this one usually blows up the size of the constraint matrix in such a way that both the number of variables and the number of constraints become large [106]. In our scenario, this would mean to give away the crucial advantage of having one parameter small. But, if care is taken, the simplex method can smoothly be generalized to QP, where the blow-up is limited by the rank of D, rather than its size. This rank is usually closely related to $\min(n, m)$ in the problems we are interested in, even if $\max(n, m)$ is large.

A simplex-like method has another advantage: it generates *basic solutions*. In the smallest enclosing ball problem, for example, this means that not only the optimal radius, but also the input points that determine the optimal ball are computed. In contrast, QP problems are often solved using interior point methods[5] and to obtain basic solutions from that requires additional effort.

In practice, the simplex-like approach is polynomial in both parameters n and m of the QP problem. This means, we can solve the concrete geometric optimization problems mentioned above even in moderately high dimensions ($50 \leq d \leq 300$, depending on the problem). In particular, for the smallest enclosing ball problem this substantially extends the range of feasible dimensions that can be handled by computational geometry codes. Interior point codes (like CPLEX's QP solver) can go much higher, so if that is required, they are the ones to choose (with the disadvantages mentioned above).

[5]for example, CPLEX's quadratic programming solver does this[6]

[6]CPLEX is a trademark of CPLEX Optimization Inc.

A major issue any serious LP or QP solver has to address is numerical accuracy [39]. On the one hand the program must not crash due to numerical problems, on the other hand the computed result should be correct. What correctness means, and what the best way is to achieve it, depends on the application. The crucial point is that general purpose LP and QP solvers *must* be able to handle any problem but *may* take advantage of inputs they typically expect. Gärtner [39] classifies the existing solvers into two categories: Either the 'may' part is ignored, i.e. the solver is *expecting the worst*, or the 'must' part is neglected, i.e. the solver is *hoping for the best*.

Expecting the worst. This strategy avoids numerical errors completely by using an exact number type in all arithmetic operations, with the disadvantage of a big performance penalty. It is too pessimistic in the sense that floating point operations are assumed to give wrong results all the time, while in practice they work fine most of the time. This approach is implemented, for example, in the LP solvers that are part of vertex-enumeration codes [5, 36].

Hoping for the best. This strategy performs all numerical operations purely with floating point arithmetic. Although this is fast and will produce the correct result in most cases, it fails on some problems. This approach can be seen as too optimistic in the sense that all problems are assumed to be well-behaved, where in practice only most of them are. It is used, for example, in the state-of-the-art solver CPLEX.

Summarizing, *expecting the worst* is always correct but also always slow, while *hoping for the best* is always fast and usually correct. For our scenario, we use a mixed strategy *expecting the best and coping with the worst* by combining floating point and exact arithmetic as suggested in [39]. This approach has the advantage of being always correct and usually fast.

The following geometric optimization problem is a natural generalization of the smallest enclosing ball problem, but it cannot be formulated as a quadratic program.

Smallest enclosing ellipsoid. The problem of finding the smallest enclosing ellipsoid of a k-point set P in d-space is an instance of convex programming. Several algorithms for computing the minimal ellipsoid have been proposed. On the one hand, there are iterative methods which employ standard optimization techniques (such as gradient descent), adapted to the problem [102, 95]. These algorithms usually work on a dual problem, known as D-optimal design [101]. On the other hand, there are finite methods which find the desired ellipsoid within a bounded number of steps. For fixed d, the algorithm of Post [81] has complexity $O(k^2)$. An optimal deterministic $O(k)$ algorithm has been given by Dyer [26], randomized $O(k)$ methods are due to Adler and Shamir [1] and Welzl [105]. Since the problem is *LP-type* in the sense of [64], generic algorithms for this class of problems can be applied as well, see [46]. In any case, the runtime dependence on d is exponential. A method for the case $d=2$, without time analysis, has been developed by Silverman and Titterington [97].

All finite methods have the property that actual work is done only for problem instances whose size is bounded by a function in d. Assuming that d is constant, such instances can be solved in constant time. However, as far as explicit formulae for these *primitive operations* have been given — which is the case only for $d = 2$ — they are quite complicated and rely on solving third-degree polynomials [97, 80, 88].

In Chapter 4 we derive explicit formulae for the primitive operations of Welzl's randomized method [105] in dimension $d = 2$. Compared to previous ones, these formulae are simpler and faster to evaluate, and they only contain rational expressions, allowing for an exact solution.

The unique ellipsoid of smallest volume enclosing a compact set P in d-space (also known as the *Löwner-John ellipsoid* of P [56]) has appealing mathematical properties which make it theoretically interesting and practically useful. In typical applications, a complicated body needs to be covered by a simple one of similar shape and volume, in order to simplify certain tests. For convex bodies (e.g. the convex hull of a finite point set), the smallest enclosing ellipsoid – unlike the isothetic bounding box or the smallest enclosing sphere – guarantees a volume approximation ratio that is independent of the shape of the covered body. This is implied by the following property, first proved by John (who also established existence and uniqueness) [56]: if the smallest enclosing ellipsoid of a compact convex body K is scaled about its center with factor $1/d$, the resulting ellipsoid lies completely inside K.

0.2 Statement of Results

The main results of this thesis can be summarized as follows.

- We developed a solver for quadratic programming problems, which is tuned for applications in computational geometry. The solver implements a generalization of the simplex method to quadratic programs. Unlike existing solvers, it is efficient if the problem is dense and has few variables or few constraints. Our method is based on a preliminary version proposed by Gärtner [38], which we worked out to a mathematically correct and practically efficient algorithm.
- For the convex programming problem of computing the smallest enclosing ellipse of a planar point set, we describe (together with Gärtner [42]) primitives for a randomized algorithm of Welzl [105], leading to a rational arithmetic solution. An implementation of Welzl's method in combination with our primitives is realized in CGAL. It is, to our knowledge, the only available implementation for the smallest enclosing ellipse problem that is based on exact arithmetic, thus guaranteeing the correctness of the solution.
- We implemented our new quadratic programming solver in C++. The implementation is carefully designed to be efficient and easy to use at the same time. The distinguishing features are a combination of exact and floating point arithmetic, user-specifiable problem representations, and almost no overhead when used as an LP solver. We used generic programming with class templates, compile time tags, and iterators to reach our design goals. The flexibility for the pricing strategy was achieved through an object-oriented design with a base class and virtual functions. The hybrid arithmetic approach in the partial filtered pricing scheme leads to a considerable speed-up. The practical usefulness is underpinned by the experimental results.
- We successfully applied our method to solve 'real-world' problems. As part of a joint project with a big Swiss bank, our quadratic programming solver was used to solve a large portfolio optimization problem (actually, we showed that the problem was infeasible). In another cooperation with a German company developing medical navigation systems, the smallest enclosing annulus implementation was used to calibrate the tip of medical probes.

0.3 Outline of the Thesis

The thesis is subdivided into three parts. The first part contains the theoretical background for the implementation work described in the second part. The third part presents two applications.

Part I. We start in Chapter 1 with the definition of linear and quadratic programs and give a brief outline of the classical simplex method. Chapter 2 describes our new algorithm for solving quadratic programming problems. We derive optimality criteria for quadratic programs and explain the details of the extended pivot step. Furthermore, we show an efficient way of handling inequality constraints and how to cope with degeneracies. In Chapter 3 we present four geometric optimization problems that can be solved by quadratic programming. Explicit formulations of the corresponding quadratic or linear programs are given. One of these problems, namely the smallest enclosing ball problem, has a natural generalization to ellipsoids. Chapter 4 describes exact primitives for solving the smallest enclosing ellipsoid problem in two dimensions.

Part II. Cgal, the Computational Geometry Algorithms Library is introduced in Chapter 5. We review related work on geometric software and precursors of Cgal. A section on generic programming is followed by an overview of the library structure. More detailed descriptions are given of the design of the geometric kernel and the basic library. Chapter 6 describes a C++ implementation of our quadratic programming solver. We state our design goals and show how we achieved them. This is done by presenting the distinguishing implementation details and small chunks of the code. In Chapter 7 we describe the three geometric quadratic programming problems that are already available in Cgal. We define the problems, show how to solve them, and sketch some important implementation details. These implementations were tested on different input sets and compared to other mathematical programming implementations as well as to dedicated geometric algorithms. The experimental results are presented in Chapter 8.

Part III. Chapter 9 presents two applications. The first one is a financial application of our quadratic programming solver. It is used to solve portfolio optimization problems. The second one is a medical application, where the tip of a medical probe, a so called stylus, has to be calibrated. We use the solution of the smallest enclosing annulus problem in this application.

Part I

Theory

Chapter 1

Linear and Quadratic Programming

Linear programming (LP) is the problem of minimizing a linear function in n variables, subject to m linear (in)equality constraints over the variables. In addition, the variables may have to lie between prespecified bounds. In this general formulation, a *linear program* can be written as

$$\begin{array}{lll} \text{(LP)} & \text{minimize} & c^T x \\ & \text{subject to} & A\,x \lesseqqgtr b \\ & & \ell \leq x \leq u\,, \end{array} \tag{1.1}$$

where A is an $m \times n$-matrix, b an m-vector, c an n-vector, and ℓ, u are n-vectors of *bounds* (values $-\infty$ and ∞ may occur). The symbol '$\lesseqqgtr$' indicates that any of the m order relations it stands for can independently be '$\leq$', '$=$', or '$\geq$'.

Generalizing LP to *quadratic programming* (QP) means replacing the linear function in (1.1) by the convex quadratic function

$$c^T x + x^T D\, x\,,$$

resulting in the *quadratic program*

$$\begin{array}{llll} \text{(QP)} & \text{minimize} & c^T x + x^T D\, x & \\ & \text{subject to} & A\, x \lesseqqgtr b & (1.2) \\ & & \ell \le x \le u\,. & \end{array}$$

Here, D is a positive semi-definite $n \times n$-matrix[1]. If $D = 0$, we obtain a linear program as a special case of QP.

If a vector $x^* = (x_1^*, \ldots, x_n^*)^T$ exists that satisfies all constraints, the problem is called *feasible* and x^* is a *feasible solution*, otherwise the problem is called *infeasible.* If the *objective function* $f(x) = c^T x + x^T D x$ is bounded from below on the set of feasible solutions x^*, the problem is called *bounded*, otherwise *unbounded.* If the problem is both feasible and bounded, the objective function assumes its unique minimum value for some (not necessarily unique) optimal feasible solution x^*. Solving a quadratic program means either finding an *optimal solution* x^* if one exists, else proving that no feasible solution exists or that there is no bounded optimum.

1.1 Standard Form

For the description, we consider QP in *standard form*, given as

$$\begin{array}{llll} \text{(QP)} & \text{minimize} & c^T x + x^T D\, x & \\ & \text{subject to} & A\, x = b & (1.3) \\ & & x \ge 0\,, & \end{array}$$

where the number of variables n is at least as large as the number of equality constraints m. As in the simplex method for LP described in the next section, explicit bounds $\ell \le x \le u$ in (1.2) can smoothly be integrated, and if inequality constraints occur, they can be turned into equalities by introducing slack variables. This means, if m is small in (1.2), we may without loss of generality assume that the quadratic program is in standard form, having only few equality constraints.

If $m \gg n$ in (1.2), the problem contains many inequality constraints in all interesting cases; however, turning them into equalities generates a large number of slack variables, so that it is no longer true that

[1]i.e. $x^T D\, x \ge 0$ holds for all x.

the resulting standard form has few equality constraints. The crucial observation here is that the former inequalities can be treated implicitly; intuitively, we 'trade them in' for the new slack variables, moving the problem's complexity from the constraints to the variables, after which we are basically back to a quadratic program in standard form with $m \leq n$. The details are described in Section 2.4. In the sequel, we assume that we are given a problem in the form of (1.3).

1.2 Simplex Method

Before we describe our QP algorithm, let us briefly review the (revised) *simplex method* for LP [20]. It operates on linear programs, given as

$$\begin{array}{lll} \text{(LP)} & \text{minimize} & c^T x \\ & \text{subject to} & A\,x = b \\ & & x \geq 0\,, \end{array} \tag{1.4}$$

where the number of variables n is at least as large as the number of equality constraints m.

Consider pairs of index sets $B, N \subseteq [n]$ with $|B| = m$, $|N| = n{-}m$, and $B \,\dot{\cup}\, N = [n]$. Each such pair induces a partition of the variables x into x_B and x_N, a partition of c into c_B and c_N, and a partition of A into A_B and A_N, where A_B collects the columns of A with indices in B (analogously for A_N). If A_B^{-1} exists, the m-vector x_B of *basic variables* and the objective function value z are uniquely determined by the $(n{-}m)$-vector x_N of *nonbasic variables*, namely

$$x_B = A_B^{-1} b - A_B^{-1} A_N x_N\,, \tag{1.5}$$

$$z \;\;= c_B^T A_B^{-1} b + (c_N^T - c_B^T A_B^{-1} A_N)\, x_N\,. \tag{1.6}$$

Assigning nonnegative values to x_N such that the implied values of x_B are nonnegative as well yields a feasible solution x^* to (1.4). It is a *basic feasible solution* if the nonbasic variables are zero. A *basis* B (in the sense of LP) is the set of basic variables (identifying B and x_B), we will refer to it as *LP-basis* to avoid any confusion with the notion of a basis for quadratic programs (defined in Section 2.2).

In each iteration, the simplex method replaces a basic variable by a nonbasic variable such that the resulting basis with corresponding basic feasible solution y^* satisfies $c^T y^* \leq c^T x^*$. An iteration – called *pivot step* – consists of three parts: the *pricing*, which either finds a nonbasic variable to *enter* the basis or certifies that the current solution is optimal; the *ratio test*, which either determines the basic variable that *leaves* the basis or states that the problem is unbounded; and the *update*, which computes the new basis B' and the corresponding basic feasible solution.

Pricing. Consider the vector $\gamma := c_N^T - c_B^T A_B^{-1} A_N$ of *reduced costs*. The current basic feasible solution is optimal if $\gamma \geq 0$, because then $x_N \geq 0$ implies $\gamma^T x_N \geq 0$, i.e. the objective function value z in (1.6) cannot be improved by increasing one of the nonbasic variables. In case $\gamma^T x_N \ngeq 0$, the pricing step returns some index j with $\gamma_j < 0$. The corresponding x_j is the *entering variable*.

Ratio Test. Increasing x_j by $t > 0$ decreases the objective function value by $\gamma_j t$. The variables become linear functions in t, i.e. $x_j^*(t) = t$ and $x_B^*(t) = x_B^* - t\, A_B^{-1} A_j$ using (1.5). The task of the ratio test is to find the largest value of t such that $x^*(t)$ stays feasible, equivalently $x_B^*(t) \geq 0$. If such a value t_0 exists, there is an index $i \in B$ with $x_i^*(t_0) = 0$ and x_i is the *leaving variable*. Otherwise the problem is unbounded.

Update. The basis B is replaced by the new basis $B' := B \cup \{j\} \setminus \{i\}$. Furthermore, some representation of the basis matrix A_B has to be maintained, suitable to compute $c_B^T A_B^{-1}$ and $A_B^{-1} A_j$ efficiently. The representation must also be easy to update, i.e. the update should be substantially cheaper than computing the representation from scratch when A_B changes to $A_{B'}$. As we will see in the implementation of the QP solver, the inverse of the basis matrix is a suitable representation in our scenario (see Part II, Section 6.3).

Further details and proofs can be found in the book of Chvátal [20].

Chapter 2

A Simplex-Like Quadratic Programming Algorithm

In this chapter, we describe our generalization of the simplex method to QP, based on a preliminary version of Gärtner [38]. We point out the main differences to the usual simplex method for LP (referred to as *LP simplex* in the following) and describe the generalized pivot step in detail. Furthermore, we show how inequality constraints can be handled efficiently. For the presentation of our method, we make the following nondegeneracy assumption.

Assumption 2.1 (nondegeneracy of quadratic program)
Given a quadratic program in standard form (1.3), we assume the following:

(i) The rows of A are linearly independent, i.e. A has full (row) rank.

(ii) The subsystem $A_G x_G = b$ has only solutions for sets $G \subseteq [n]$ with $|G| \geq m$.

Later in this chapter, we describe how to deal with these degeneracies. Section 2.5 gives the details.

2.1 Karush-Kuhn-Tucker Conditions

In the LP simplex, the linear objective function is kept in a way that allows to decide directly if the current solution is optimal. In case of QP things are more complicated, since the objective function is no longer linear. We will use the Karush-Kuhn-Tucker (KKT) conditions for convex programming [79, 11] to decide optimality. In the most general setting, these conditions hold for convex programs, i.e. minimizing a convex objective function subject to convex equality and inequality constraints. The following two theorems are specialized versions, they provide necessary and sufficient optimality criteria for quadratic programs [11, Section 3.4].

Theorem 2.2 (KKT conditions for QP)
A feasible solution $x^* \in \mathbb{R}^n$ to the quadratic program

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & c^T x + x^T D\, x \\ & \text{subject to} & A\, x = b \\ & & x \geq 0\,, \end{array}$$

is optimal if and only if there exists an m-vector λ and an n-vector $\mu \geq 0$ such that

$$\begin{aligned} c^T + 2{x^*}^T D &= -\lambda^T A + \mu^T\,, \\ \mu^T x^* &= 0\,. \end{aligned}$$

Our QP algorithm described below will solve subproblems without nonnegativity constraints. The following version of the KKT conditions is known as the method of Lagrange multipliers.

Theorem 2.3 (KKT conditions for unconstrained QP)
A feasible solution $x^* \in \mathbb{R}^n$ to the unconstrained quadratic program

$$\begin{array}{lll} \text{(UQP)} & \text{minimize} & c^T x + x^T D\, x \\ & \text{subject to} & A\, x = b\,, \end{array}$$

is optimal if and only if there exists an m-vector λ such that

$$c^T + 2{x^*}^T D = -\lambda^T A\,.$$

Note, if the rows of A are linearly independent, then λ and μ are unique in Theorem 2.2. The same holds for λ in Theorem 2.3.

Corollary 2.4 *Any vector $x^* > 0$ satisfying $A\,x = b$ is an optimal solution to (QP) if and only if it is an optimal solution to (UQP).*

Proof. For $x^* > 0$ the second condition of Theorem 2.2 implies $\mu^T = 0$. Thus, the first condition of Theorem 2.2 and the (only) condition of Theorem 2.3 are equivalent. □

2.2 Basic Solutions

As the LP simplex, the QP simplex iterates through a sequence of basic feasible solutions, always improving the objective function value. Interpreted geometrically, the set of feasible solutions is a polyhedron $\mathcal{P}$. In case of LP, each basic feasible solution corresponds to a vertex of $\mathcal{P}$, hence there is always an optimal solution which is a vertex of $\mathcal{P}$. In case of QP, this is not true in general, as the following trivial example shows: Minimize x^2 subject to $-1 \leq x \leq 1$. Here $\mathcal{P}$ is a polytope in 1-space with vertices -1 and 1, but the unique optimal solution is assumed at point 0, in the interior of $\mathcal{P}$. Thus we have to consider more than only the basic feasible solutions in the sense of LP to find the optimum of a quadratic program.

A (QP) *basic feasible solution* is characterized by a subset B of the variables (a *QP-basis*, defined below) and numerical values x^*_B for the variables in B; variables not in B will always have value zero. A variable is called *basic* if it is in B, *nonbasic* otherwise. Unlike in the LP simplex, the basis B may contain more than m variables, and in this case their values are not uniquely determined by only the constraints of the quadratic program. Instead, x^*_B will be the optimal solution to an *unconstrained* subproblem as follows.

Definition 2.5 (QP-Basis)
A subset B of the variables of a quadratic program in standard form defines a QP-basis if and only if

(i) the unconstrained subproblem

$$\begin{array}{lll} (UQP(B)) & \textit{minimize} & c_B^T x_B + x_B^T D_B x_B \\ & \textit{subject to} & A_B x_B = b \end{array} \tag{2.1}$$

has a unique optimal solution $x_B^* > 0$, and

(ii) A_B has full (row) rank, i.e. $\operatorname{rank}(A_B) = m$,[1]

where c_B, D_B and A_B are the entries of c, D and A relevant for the variables in B, respectively.

Obviously, x^* with $x_i^* = 0$ for all nonbasic variables x_i is a feasible solution to the original problem (1.3). In case of LP (i.e. $D = 0$), the definition of a QP-basis specializes to the usual notion of basic feasible solutions (in the nondegenerate case).

In the sequel we identify each basic variable with the corresponding index, i.e. $B \subseteq [n]$ and $i \in B$ if and only if x_i is basic. The nonbasic variables are collected in $N := [n] \setminus B$.

The following theorem gives an upper bound on the maximum size of a QP-basis. The idea of the proof is due to Gärtner [38].

Theorem 2.6 *Every QP-basis B satisfies*

$$|B| \leq m + \operatorname{rank}(D).$$

Proof. Suppose there exists a basis B with corresponding optimal solution x_B^* and $|B| > m + \operatorname{rank}(D)$. We will show that x_B^* is not the *unique* solution to (UQP(B)), thus contradicting the definition of a QP-basis.

By the KKT conditions (Theorem 2.3), there exists a vector λ such that

$$c_B^T + 2{x_B^*}^T D_B = -\lambda^T A_B \, ,$$

[1]Note that A_B has at least m columns by Assumption 2.1 (ii).

if and only if x^*_B is optimal for (UQP(B)). It follows that any $y \geq 0$ in the intersection of the affine spaces

$$U := \{\, y \mid y^T D_B = {x^*}^T D_B \,\}\,, \qquad V := \{\, y \mid A_B\, y = b \,\}$$

is also an optimal solution to (UQP(B)). Since U and V are solution spaces of linear equation systems, their dimension is determined by the difference of the number of variables and the rank of the respective matrices D and A, see e.g. [32]. With $k := |B|$ and $r := \text{rank}(D)$, we get

$$\begin{aligned} \dim(U \cap V) &= \dim(U) + \dim(V) - \dim(U{+}V) \\ &\geq (k{-}r) + (k{-}m) - k \\ &= k - (m{+}r)\,. \end{aligned}$$

Let $W := \{\, y \mid y \geq 0 \,\}$, then the polytope $\mathcal{P} := U \cap V \cap W$ has dimension at least $k{-}(m{+}r)$ and is contained in the set of optimal solutions of (UQP(B)). The assumption $k > m{+}r$ implies $\dim(\mathcal{P}) > 0$, thus x^*_B is not unique. □

Again, if $D = 0$, we recover the bound for LP-bases. The theorem is a crucial ingredient of our method. Well-known methods to generalize the simplex algorithm to QP integrate D explicitly into the constraint matrix [106]. This is appropriate for large sparse problems and corresponding solvers, because it does not increase the number of nonzero entries in the problem description. In our scenario, however, this method is inappropriate, because it results in a problem with many variables *and* many constraints. The lemma limits the influence of D by considering its rank rather than its size. As we see in the next chapter, the rank of D is small in our applications.

2.3 Pivot Step

The process of going from one basic solution to the next is called a *pivot step.* In the LP simplex, the *pricing* is the part of the pivot step that decides whether the current solution is optimal, and which variable *enters* the basis if an improvement is possible.

2.3.1 Pricing

Testing whether a nonbasic variable x_j can improve the solution by entering the current QP-basis B is done as follows. Let $\hat{B} := B \cup \{j\}$ and consider the subproblem

$$\begin{array}{lll} (\mathrm{QP}(\hat{B})) & \text{minimize} & c_{\hat{B}}^T x_{\hat{B}} + x_{\hat{B}}^T D_{\hat{B}} x_{\hat{B}} \\ & \text{subject to} & A_{\hat{B}} x_{\hat{B}} = b \\ & & x_{\hat{B}} \geq 0\,. \end{array} \tag{2.2}$$

By the KKT conditions for QP (Theorem 2.2), $x_{\hat{B}}^*$ (with $x_j^* = 0$) is an optimal solution to $(\mathrm{QP}(\hat{B}))$ if and only if there exist vectors λ and $\mu \geq 0$ such that

$$c_{\hat{B}}^T + 2{x_{\hat{B}}^*}^T D_{\hat{B}} = -\lambda^T A_{\hat{B}} + \mu \tag{2.3}$$

$${x_{\hat{B}}^*}^T \mu = 0\,. \tag{2.4}$$

Since $x_B^* > 0$, $\mu_B = 0$ holds using (2.4). Extracting x_j^* from (2.3) gives

$$c_B^T + 2{x_B^*}^T D_B \quad + 2x_j^*\, D_{B,j}^T = -\lambda^T A_B \tag{2.3a}$$

$$c_j + 2{x_B^*}^T D_{B,j} + 2x_j^*\, D_{j,j} \ = -\lambda^T A_j + \mu_j\,, \tag{2.3b}$$

where $D_{B,j}$ is the j-th column of D_B. Equations (2.3a) together with the constraints of (2.1) determine x_B^* (with $x_j^* = 0$) and λ by the linear equation system

$$M_B \begin{pmatrix} \lambda \\ \hline x_B^* \end{pmatrix} = \begin{pmatrix} b \\ \hline -c_B \end{pmatrix}, \qquad M_B := \left(\begin{array}{c|c} 0 & A_B \\ \hline A_B^T & 2D_B \end{array}\right). \tag{2.5}$$

We call M_B the *basis matrix* (of B). By the definition of a QP-basis, x_B^* is the unique optimal solution to $(\mathrm{UQP}(B))$ and A_B has full row rank. Thus, also λ is unique and M_B is regular, i.e. M_B^{-1} exists.

Using the values of x_B^* and λ, (2.3b) uniquely determines μ_j. In case μ_j is negative, $x_{\hat{B}}^*$ (with $x_j^* = 0$) cannot be an optimal solution to $(\mathrm{QP}(\hat{B}))$ because $\mu_j < 0$ contradicts $\mu \geq 0$ in Theorem 2.2. On the other hand, if $\mu_j \geq 0$ holds for all nonbasic variables x_j, we found vectors λ and $\mu \geq 0$ fulfilling the KKT conditions for the original problem, i.e. x^* with $x_N^* = 0$ is an optimal solution to (QP). Hence, the current solution can be improved by entering variable x_j into the basis if and only if $\mu_j < 0$.

Different strategies for choosing the entering variable are described in Part II, Section 6.4.

2.3.2 Ratio Test

The second part of the LP simplex's pivot step is the *ratio test*. It decides which variable has to *leave* the basis if another variable enters it. Unless the problem is degenerate, there is a unique choice for the leaving variable. In the QP simplex, it can happen that a variable enters the basis, but there is no leaving variable, so that the basis gets larger. This is the case if the objective function reaches a local minimum (while the entering variable is increased), before some other (basic) variable goes down to zero. Moreover, it can happen that even if some leaving variable is found, the solution at that point is not basic. In this case, the pivot step continues, and more variables may leave the basis, until another basic solution is discovered. The QP ratio test consists of three steps which are described below.

Step 1. Starting with a QP-basis B and an entering variable x_j, we want to find a new basis $B' \subseteq B \cup \{j\}$ with better objective function value. Define $\hat{B} := B \cup \{j\}$, then $x^*_{\hat{B}}$ (with $x^*_j = 0$) is the optimal (and unique) solution to

$$\begin{array}{lll} (\mathrm{UQP}^t_j(\hat{B})) & \text{minimize} & c^T_{\hat{B}} x_{\hat{B}} + x^T_{\hat{B}} D_{\hat{B}} x_{\hat{B}} \\ & \text{subject to} & A_{\hat{B}} x_{\hat{B}} = b \\ & & x_j = t \end{array} \tag{2.6}$$

for $t = 0$. Furthermore, $(\mathrm{UQP}^t_j(\hat{B}))$ has a unique optimal solution $x^*_{\hat{B}}(t)$ for each value of t, which is determined by

$$M_B \begin{pmatrix} \lambda(t) \\ \hline x^*_B(t) \end{pmatrix} = \begin{pmatrix} b \\ \hline -c_B \end{pmatrix} - t \begin{pmatrix} A_j \\ \hline 2D_{B,j} \end{pmatrix} \tag{2.7}$$

and $x^*_j(t) = t$. This follows from the KKT conditions for (2.6) (Theorem 2.3), the regularity of M_B, and some elementary transformations.

In the sequel, we use the following equivalent formulation

$$\begin{pmatrix} \lambda(t) \\ \hline x^*_B(t) \end{pmatrix} = \begin{pmatrix} \lambda \\ \hline x^*_B \end{pmatrix} - t \begin{pmatrix} q_\lambda \\ \hline q_x \end{pmatrix}, \qquad \begin{pmatrix} q_\lambda \\ \hline q_x \end{pmatrix} := M_B^{-1} \begin{pmatrix} A_j \\ \hline 2D_{B,j} \end{pmatrix}. \tag{2.8}$$

While increasing t starting from zero (which improves the current solution), two things may happen. Either one of the basic variables becomes zero, or a local minimum of the objective function is reached. The first event can be tested by solving $x^*_i(t) = 0$ for all $i \in B$. To check the second event, we look at the linear function $\mu_j(t)$ derived from (2.3b) and (2.8)

$$\begin{aligned} \mu_j(t) &= c_j + A_j^T \lambda(t) + 2D_{B,j}^T x^*_B(t) + 2D_{j,j} x^*_j(t) \\ &= \mu_j + t\,(2D_{j,j} - A_j^T q_\lambda - 2D_{B,j}^T q_x)\,. \end{aligned} \tag{2.9}$$

As soon as $\mu_j(t)$ becomes zero for some $t > 0$, $\lambda(t)$ and $\mu = 0$ fulfill the KKT conditions of Theorem 2.2, thus $x^*_{\hat{B}}$ with $x^*_j(t) = t$ is an optimal solution to (QP($\hat{B}$)). The following lemma shows that $\hat{B}$ is the new basis in case of the second event.

Lemma 2.7 *If $\mu_j(t)$ in (2.9) becomes zero for some $t > 0$ before any of the basic variables vanishes, then $\hat{B}$ is the new basis.*

Proof. We already know that $x^*_{\hat{B}}(t)$ is an optimal solution to (QP($\hat{B}$)), if $\mu_j(t) = 0$. We will show that $M_{\hat{B}}$ is regular, which directly implies that $x^*_{\hat{B}}(t)$ is unique and that A_B has full row rank.

Consider the matrices M_B and $M_{\hat{B}}$. The latter is just the basis matrix M_B plus one additional row and one additional column for index j, which can be assumed to be the largest index in B, w.l.o.g. It is easy to verify that

$$M_{\hat{B}} = \left(\begin{array}{cc|c} 1 & & 0 \\ & \ddots & \vdots \\ & 1 & 0 \\ \hline q_\lambda^T \,|\, & q_x^T & 1 \end{array}\right) \left(\begin{array}{c|c} & 0 \\ M_B & \vdots \\ & 0 \\ \hline 0 \;\dots\; 0 & \nu \end{array}\right) \left(\begin{array}{cc|c} 1 & & q_\lambda \\ & \ddots & \overline{} \\ & 1 & q_x \\ \hline 0 \;\dots & 0 & 1 \end{array}\right) \tag{2.10}$$

holds, where

$$\nu := 2D_{j,j} - A_j^T q_\lambda - 2D_{B,j}^T q_x.$$

The determinant of $M_{\hat{B}}$ is

$$\det(M_{\hat{B}}) = \det(M_B)\,\nu\,,$$

i.e. $M_{\hat{B}}$ is regular if $\nu \neq 0$, since $\det(M_B) \neq 0$ follows from the fact that M_B is a basis matrix. Suppose $\nu = 0$, then by (2.9)

$$\mu_j(t) = \mu_j + t\,\nu = \mu_j < 0\,,$$

a contradiction to $\mu_j(t) = 0$. □

In case the first event happens, i.e. some basic variable x_i becomes zero, we implicitly add the constraint $x_i = 0$ to ($\mathrm{UQP}_j^t(\hat{B})$) by removing index i from B. If $M_{B\setminus\{i\}}$ is regular, we still have a unique optimal solution to ($\mathrm{UQP}_j^t(\hat{B}\setminus\{i\})$) for each value of t and Step 1 is iterated. Otherwise we proceed to Step 2.

Step 2. Let B be the set of basic variables after the last iteration of Step 1. Since M_B has become singular, (2.7) does no longer determine unique solutions to ($\mathrm{UQP}_j^t(\hat{B})$) for arbitrary t (with $\hat{B} = B \cup \{j\}$). Reconsidering the KKT conditions for ($\mathrm{QP}(\hat{B})$), i.e. Equations (2.3) and (2.4), yields

$$M_{\hat{B}} \begin{pmatrix} \lambda \\ \hline x_B^* \\ \hline x_j^* \end{pmatrix} = \begin{pmatrix} b \\ \hline -c_B \\ \hline -c_j \end{pmatrix} + \mu_j \begin{pmatrix} 0 \\ \hline 0 \\ \hline 1 \end{pmatrix}. \tag{2.11}$$

In case $M_{\hat{B}}$ is singular, we proceed directly to Step 3. Otherwise, the system of linear equations (2.11) has a unique solution for each value of μ_j. The solutions are determined by a linear function in μ_j, which can be written as

$$\begin{pmatrix} \lambda(\mu_j) \\ \hline x_{\hat{B}}^*(\mu_j) \end{pmatrix} = \begin{pmatrix} \lambda \\ \hline x_{\hat{B}}^* \end{pmatrix} + \mu_j \begin{pmatrix} p_\lambda \\ \hline p_{x_{\hat{B}}} \end{pmatrix},$$

with

$$\begin{pmatrix} p_\lambda \\ \hline p_{x_B} \\ \hline p_{x_j} \end{pmatrix} := M_{\hat{B}}^{-1} \begin{pmatrix} 0 \\ \hline 0 \\ \hline 1 \end{pmatrix}. \tag{2.12}$$

Any solution $x^*_{\hat{B}}(\mu_j)$ is feasible for $(\mathrm{UQP}(\hat{B}))$, and it is optimal if $\mu_j = 0$. Let t_1 be the value of t for which the first event occured in the last iteration of Step 1, then $x^*_{\hat{B}}(\mu_j(t_1))$ is the current feasible solution at the beginning of Step 2.

While growing μ_j from $\mu_j(t_1)$ towards zero (similar to t in Step 1), two events may occur. Again, either one of the remaining basic variables becomes zero, or a local minimum of the objective function is reached. If the latter happens, i.e. μ_j equals zero, we found an optimal solution $x^*_{\hat{B}}(0) > 0$ to $(\mathrm{UQP}(\hat{B}))$, which is at the same time an optimal solution to the constrained problem $(\mathrm{QP}(\hat{B}))$ by Corollary 2.4. The uniqueness of the solution follows from the regularity of $M_{\hat{B}}$, which also implies that $\hat{B}$ is the new basis in that case.

On the other hand, if some basic variable x_k becomes zero (first event), we implicitly add the constraint $x_k = 0$ to $(\mathrm{UQP}(\hat{B}))$ by removing k from $\hat{B}$. If $M_{\hat{B}\setminus\{k\}}$ stays regular, we still get unique solutions of (2.11) for arbitrary values of μ_j. In this case Step 2 is iterated, otherwise we continue with Step 3.

Step 3. Let $\hat{B}$ be the set of basic variables (including x_j) after Step 2. Then, by Lemma 2.8 below, the linear equation system (2.11) has only solutions if $\mu_j = 0$, hence any solution of (2.11) is already an optimal solution to $(\mathrm{UQP}(\hat{B}))$. But, since $M_{\hat{B}}$ is singular, $\hat{B}$ is not a QP-basis yet. Using the polytope $\mathcal{P}$ from the proof of Theorem 2.6 (with $\hat{B}$ instead of B), we find a basis $B' \subsetneq \hat{B}$ with the same objective function value as follows.

We initialize B' with $\hat{B}$. Starting from $x^*_{\hat{B}}$ (which obviously belongs to $\mathcal{P}$), we decrease the value of some variable while maintaining the invariant $x_{B'} \in \mathcal{P}$, until the first variable, say x_i, becomes zero. Note that x_i can be the decreased variable itself or any of the other variables. Fixing x_i to zero, equivalently replacing $\mathcal{P}$ with the intersection of $\mathcal{P}$ and the hyperplane $\{\, x \mid x_i = 0 \,\}$, reduces the dimension of $\mathcal{P}$ by one. We remove i from B' and iterate this step until the dimension of $\mathcal{P}$ becomes zero. Then $\mathcal{P}$ consists of the single point $x^*_{B'}$, which is the unique solution to $(\mathrm{UQP}(B'))$.

Lemma 2.8 *In case $M_{\hat{B}}$ is singular after Step 2, then the linear equation system (2.11) has only solutions if $\mu_j = 0$.*

Proof. The right-hand-side of (2.11) is a vector-valued linear function in μ_j. Using Gaussian elimination, $M_{\hat{B}}$ can be made triangular while the right-hand-side stays linear in μ_j (see e.g. [32, Section 0.4]). Thus, any solution to (2.11) can be written as a linear polynomial in μ_j, i.e.

$$\begin{pmatrix} \lambda \\ \hline x_B^* \\ \hline x_j^* \end{pmatrix} =: \begin{pmatrix} y_\lambda \\ \hline y_B \\ \hline y_j \end{pmatrix} + \mu_j \begin{pmatrix} z_\lambda \\ \hline z_B \\ \hline z_j \end{pmatrix}.$$

If $\mu_j \neq 0$, the vector z has to fulfill

$$M_{\hat{B}} \begin{pmatrix} z_\lambda \\ \hline z_B \\ \hline z_j \end{pmatrix} = \begin{pmatrix} 0 \\ \hline 0 \\ \hline 1 \end{pmatrix}. \tag{2.13}$$

Showing that (2.13) has no solution proves the lemma.

There exists a vector $r \neq 0$ with $r^T M_{\hat{B}} = 0$ because $M_{\hat{B}}$ is singular. Assuming that (2.13) has a solution z^* gives

$$r_j = r^T \begin{pmatrix} 0 \\ \hline 0 \\ \hline 1 \end{pmatrix} = r^T M_{\hat{B}} z^* = 0\,. \tag{2.14}$$

On the other hand, we get $r_j \neq 0$ for all $r \neq 0$ with $r^T M_{\hat{B}} = 0$ by the following case distinction, contradicting the assumption that (2.13) has a solution.

If there has been no iteration in Step 2, i.e. both M_B and $M_{\hat{B}}$ are singular after Step 1, let x_i with $x_i(t) = 0$ be the last removed variable. Then $M_{B \cup \{i\}}$ is regular and (2.8) reads as

$$M_{B \cup \{i\}} \begin{pmatrix} q_\lambda \\ \hline q_x \\ \hline q_{x_i} \end{pmatrix} = \begin{pmatrix} A_j \\ \hline 2D_{B,j} \\ \hline 2D_{i,j} \end{pmatrix},$$

which gives

$$M_B \begin{pmatrix} q_\lambda \\ \hline q_x \end{pmatrix} + \begin{pmatrix} A_i \\ \hline 2D_{B,i} \end{pmatrix} q_{x_i} = \begin{pmatrix} A_j \\ \hline 2D_{B,j} \end{pmatrix}$$

if the last row is ignored. Now, for all $r' \neq 0$ with ${r'}^T M_B = 0$ holds

$$ {r'}^T \left(\frac{A_j}{2D_{B,j}} \right) \neq 0, \tag{2.15} $$

since $(A_i^T | 2D_{B,i}^T) r' \neq 0$ follows from the regularity of $M_{B\cup\{i\}}$ and $q_{x_i} \neq 0$ is implied by $x_i(t) = 0$. Suppose there exists a vector $\tilde{r} \neq 0$ with $\tilde{r}_j = 0$ and $\tilde{r}^T M_{\hat{B}} = 0$. Then

$$ \tilde{r}^T M_{\hat{B}} = \left(\frac{\tilde{r}_B}{0} \right)^T M_{\hat{B}} = \left(\tilde{r}_B^T M_B \;\middle|\; \tilde{r}_B^T \left(\frac{A_j}{2D_{B,j}} \right) \right) $$

shows

$$ \tilde{r}_B^T \left(\frac{A_j}{2D_{B,j}} \right) = 0, $$

contradicting (2.15), which completes the first case.

The second case occurs when there has been at least one iteration in Step 2. Let x_k with $x_k(\mu_j) = 0$ be the last removed variable, then $M_{\hat{B}\cup\{k\}}$ is regular and (2.12) gives

$$ M_{\hat{B}\cup\{k\}} \begin{pmatrix} p_\lambda \\ \hline p_{x_B} \\ \hline p_{x_k} \\ \hline p_{x_j} \end{pmatrix} = \begin{pmatrix} 0 \\ \hline 0 \\ \hline 0 \\ \hline 1 \end{pmatrix}. $$

By ignoring the 'k'-th row, we obtain

$$ M_{\hat{B}} \begin{pmatrix} p_\lambda \\ \hline p_{x_B} \\ \hline p_{x_j} \end{pmatrix} + \begin{pmatrix} A_k \\ \hline 2D_{B,k} \\ \hline 2D_{j,k} \end{pmatrix} p_{x_k} = \begin{pmatrix} 0 \\ \hline 0 \\ \hline 1 \end{pmatrix}. $$

Finally, using the regularity of $M_{\hat{B}\cup\{k\}}$ and $p_{x_k} \neq 0$ (which is implied by $x_k(\mu_j) = 0$)), it follows that

$$ r_j = r^T \begin{pmatrix} 0 \\ \hline 0 \\ \hline 1 \end{pmatrix} = r^T \begin{pmatrix} A_k \\ \hline 2D_{B,k} \\ \hline 2D_{j,k} \end{pmatrix} z_k \neq 0 $$

holds for all $r \neq 0$ with $r^T M_{\hat{B}} = 0$. This is a contradiction to (2.14) and completes the proof. □

2.3.3 Update

The old basis B gets replaced by the new basis $B' \subseteq B \cup \{j\}$ which was determined in the ratio test. Furthermore, some representation of the basis matrix M_B has to be maintained, which allows to compute x^*_B and λ of (2.5), q_λ and q_x of (2.8), and p_λ and $p_{x_{\hat{B}}}$ of (2.12) efficiently. In addition, the update should be substantially cheaper than computing the representation from scratch when M_B changes to $M_{B'}$. As already suggested in the previous chapter, we will see that the inverse of the basis matrix is a suitable representation in our scenario (cf. Part II, Section 6.3).

2.4 Inequality Constraints

The efficiency of our method results from a special property of the problems we consider. Usually, the number of constraints m, which is a lower bound on the basis size (in the nondegenerate case), is very small. Various problems even have m constant and in some cases we are able to give an upper bound on the basis size that does not depend on the possibly large number of variables n, (cf. Chapter 3). On the other hand, we also want to solve problems with only few variables but many (in)equality constraints, i.e. we have $m \gg n$. By introducing slack variables in the usual way [20], we end up with a problem with up to $n{+}m$ variables and m constraints. Hence both numbers depend on the possibly large m and we lost the special property of one very small parameter of our original problem. The consequences are large bases and a basis matrix of size at least $(2m)\times(2m)$. To overcome this undesirable situation, we exploit the fact that only constraints holding with equality determine the values of the original variables. These are the equality constraints and those inequality constraints for which the corresponding slack variable is nonbasic.

If the given quadratic program contains inequality constraints, they are removed by introducing *slack variables.* Let E and S be the sets of indices of the equality and inequality constraints, resp. For each $i \in S$, we replace the inequality

$$\sum_{j=1}^{m} a_{ij} x_j \lesseqgtr b_i,$$

by the equation

$$\sum_{j=1}^{m} a_{ij} x_j \pm x_{s_i} = b_i, \tag{2.16}$$

where x_{s_i} is added, if the relation is $\leq$, and subtracted otherwise. The matrix A is enlarged by $|S|$ *slack columns*

$$A_{s_i} := \pm e_i, \quad i \in S, \tag{2.17}$$

where e_i is the i-th unit vector, resulting in a quadratic program with $n{+}|S|$ variables and m equality constraints. Given a basis B, we want to compute the current solution x^*_B. By the KKT conditions (Theorem 2.2), we have to solve the system

$$A_B x_B = b, \tag{2.18}$$
$$A_B^T \lambda + 2 D_B x_B = -c_B. \tag{2.19}$$

Define B_O and B_S to be the sets of indices of the original resp. slack variables in the current basis, i.e. $B = B_O \,\dot{\cup}\, B_S$. Furthermore, partition S into two sets, such that S_B and S_N contain the indices of those inequality constraints for which the corresponding slack variable is currently basic resp. nonbasic. Now, Equation (2.18) can be written as

$$\left(\begin{array}{c|c} A_{E\ ,B_O} & A_{E\ ,B_S} \\ \hline A_{S_N,B_O} & A_{S_N,B_S} \\ \hline A_{S_B,B_O} & A_{S_B,B_S} \end{array}\right) \left(\begin{array}{c} x_{B_O} \\ \hline x_{B_S} \end{array}\right) = \left(\begin{array}{c} b_E \\ \hline b_{S_N} \\ \hline b_{S_B} \end{array}\right),$$

equivalently

$$A_{E\ ,B_O} x_{B_O} + A_{E\ ,B_S} x_{B_S} = b_E, \tag{2.20}$$
$$A_{S_N,B_O} x_{B_O} + A_{S_N,B_S} x_{B_S} = b_{S_N}, \tag{2.21}$$
$$A_{S_B,B_O} x_{B_O} + A_{S_B,B_S} x_{B_S} = b_{S_B}. \tag{2.22}$$

Two things follow from (2.17). Firstly, A_{E,B_S} and A_{S_N,B_S} have only zero entries, thus the second terms in (2.20) and (2.21) vanish. Secondly, the values of the basic slack variables are determined by (2.22), i.e.

$$\pm x_{s_i} = b_i - A_{i,B_O} x_{B_O}, \quad s_i \in B_S,$$

where the sign of x_{s_i} is the same as in (2.16). Splitting up equation (2.19) in a similar manner yields

$$\left(\begin{array}{c|c|c} A^T_{E,B_O} & A^T_{S_N,B_O} & A^T_{S_B,B_O} \\ \hline A^T_{E,B_S} & A^T_{S_N,B_S} & A^T_{S_B,B_S} \end{array}\right)\left(\begin{array}{c} \lambda_E \\ \hline \lambda_{S_N} \\ \hline \lambda_{S_B} \end{array}\right)$$
$$+\left(\begin{array}{c|c} 2D_{B_O,B_O} & 2D_{B_O,B_S} \\ \hline 2D_{B_S,B_S} & 2D_{B_S,B_S} \end{array}\right)\left(\begin{array}{c} x_{B_O} \\ \hline x_{B_S} \end{array}\right) = \left(\begin{array}{c} -c_{B_O} \\ \hline -c_{B_S} \end{array}\right),$$

respectively

$$\begin{aligned} A^T_{E,B_O}\lambda_E + A^T_{S_N,B_O}\lambda_{S_N} + A^T_{S_B,B_O}\lambda_{S_B} \\ + 2D_{B_O,B_O}x_{B_O} + 2D_{B_O,B_S}x_{B_S} &= -c_{B_O}, && (2.23) \\ A^T_{E,B_S}\lambda_E + A^T_{S_N,B_S}\lambda_{S_N} + A^T_{S_B,B_S}\lambda_{S_B} \\ + 2D_{B_S,B_O}x_{B_O} + 2D_{B_S,B_S}x_{B_S} &= -c_{B_S}. && (2.24) \end{aligned}$$

Here, only D_{B_O,B_O} has nonzero entries while all other parts of D have just zero entries, and so has $-c_{B_S}$. Together with the known charateristics of A, equation (2.24) boils down to

$$\pm\lambda_i = 0, \quad i \in S_B.$$

Plugging this into (2.23) gives

$$A^T_{E,B_O}\lambda_E + A^T_{S_N,B_O}\lambda_{S_N} + 2D_{B_O,B_O}x_{B_O} = -c_{B_O}.$$

Finally, we end up with the equation system

$$\begin{aligned} A_{E\dot\cup S_N,B_O}x_{B_O} &= b_{E\dot\cup S_N} && (2.25) \\ A^T_{E\dot\cup S_N,B_O}\lambda_{E\dot\cup S_N} + 2D_{B_O,B_O}x_{B_O} &= -c_{B_O} && (2.26) \end{aligned}$$

where the number of variables and the number of constraints is bounded by

$$|E| + |S_N| + |B_O| \leq \min\{n, m\} + n.$$

Here, we estimated $|E|+|S_N|$ in two ways: $|E|+|S_N| \leq |E|+|S| = m$ and $|E|+|S_N| = |E|+|S|-|S_B| = m-|B_S| = m-|B|+|B_O| \leq n$.

Summarizing, we found a small representation of the equation system given in (2.18) and (2.19). The size of the new representation is at most $(\min\{n,m\}+n) \times (\min\{n,m\}+n)$, which is $(2n) \times (2n)$ if $m > n$ — a tremendous improvement for $m \gg n$ over the original size of at least $(2m) \times (2m)$. Solving the new equation system gives the values of $x^*_{B_O}$ and $\lambda_{E \dot{\cup} S_N}$, the entries of $x^*_{B_S}$ are computed directly from $x^*_{B_O}$, and λ_{S_B} is the zero vector.

2.5 Degeneracies

According to Assumption 2.1 (i), we have to cope with linearly dependent rows of A. We address the problem by either transforming $Ax = b$ into an equivalent system $A'x = b'$ with $\text{rank}(A') = m$ or by showing that there is no solution at all, meaning that the optimization problem is infeasible.

To test the rank condition of a given equation system $Ax = b$, we bring matrix A into triangular form using some standard method, e.g. Gaussian elimination. Let $A'x' = b'$ be the resulting system with

$$A' = \begin{pmatrix} a'_{1,1} & a'_{1,2} & \dots & a'_{1,m} & \dots & a'_{1,n} \\ 0 & a'_{2,2} & \dots & a'_{2,m} & \dots & a'_{2,n} \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \dots & 0 & a'_{m,m} & \dots & a'_{m,n} \end{pmatrix}, \tag{2.27}$$

where $a'_{j,j} \neq 0$ holds if at least one nonzero entry exists in row j. This can be always achieved by swapping the columns of A' appropriately.[2]

In case we end up with all entries of the diagonal $a'_{1,1}$ to $a'_{m,m}$ being nonzero, then A' and A have full (row) rank m and we are done. Otherwise some $a'_{j,j} = 0$, i.e. row j of A' has only zero entries, thus we found a linearly dependent row of A. If the corresponding b'_j is zero too, we just remove constraint j from the equation system. This is done for all 'zero' rows, resulting in an equivalent system where the constraint

[2]The vector x' contains the entries of x in possibly different order. A pair of entries (x_i, x_j) is swapped in x' if the corresponding columns of A were swapped in A' during the transformation.

matrix has full row rank. Only if $b'_j \neq 0$ holds, constraint j cannot be fulfilled and the original equation system $Ax = b$ has no solution.

If there exists a solution of $Ax = b$ with less than m nonzero entries, then we guarantee the second condition of Assumption 2.1 to hold using symbolic perturbation. The right-hand-side of the equation system is perturbed by the vector $\varepsilon := (\epsilon, \epsilon^2, \ldots, \epsilon^m)^T$ for some $0 < \epsilon < 1$. Any solution of the resulting equation system

$$Ax = b + \varepsilon \tag{2.28}$$

is a vector-valued polynomial in ϵ, i.e.

$$x = \begin{pmatrix} x_1(\epsilon) \\ x_2(\epsilon) \\ \vdots \\ x_n(\epsilon) \end{pmatrix} = \begin{pmatrix} x_1^0 + x_1^1\epsilon + x_1^2\epsilon^2 + \cdots + x_1^m\epsilon^m \\ x_2^0 + x_2^1\epsilon + x_2^2\epsilon^2 + \cdots + x_2^m\epsilon^m \\ \vdots \\ x_n^0 + x_n^1\epsilon + x_n^2\epsilon^2 + \cdots + x_n^m\epsilon^m \end{pmatrix}. \tag{2.29}$$

Choosing ϵ appropriately ensures that no solution has less than m nonzero entries. However, this choice has never to be done explicitly as we will see in the implementation of the QP solver (cf. Part II, Section 6.2.4).

2.6 Conclusion

We presented an algorithm for solving quadratic programming problems. The solver is a generalization of the simplex method to quadratic programs. Unlike existing solvers, it is efficient if the problem is dense and has either few variables or few constraints. Inequality constraints are handled implicitly and degenerate problems are treated using symbolic perturbation.

Chapter 3

Geometric Optimization Problems

In this chapter we present four geometric optimization problems that can be formulated as instances of LP or QP.

3.1 Polytope Distance

Consider two point sets $P = \{ p_1, \dots, p_r \}$ and $Q = \{ q_1, \dots, q_s \}$ with $r+s = n$ in d-dimensional space. The *polytope distance* problem deals with minimizing $\|p-q\|$ such that $p = \sum_{i=1}^{r} \lambda_i p_i$ and $q = \sum_{i=1}^{s} \mu_i q_i$, see Figure 3.1 (primal version). The λ_i and μ_i are in $[0,1]$ and sum up to one. This can be written as a quadratic program in n variables and two constraints, namely

$$\begin{array}{lll} \text{(PD)} & \text{minimize} & x^T C^T C\, x \\ & \text{subject to} & \sum_{i=1}^{r} \quad x_i = 1 \\ & & \sum_{i=r+1}^{n} x_i = 1 \\ & & x \geq 0\,, \end{array} \tag{3.1}$$

where $C = (p_1, \dots, p_r, -q_1, \dots, -q_s)$. Here, $D = C^T C$ is an $n \times n$-matrix, but its rank is only d. Hence, by Theorem 2.6, the QP simplex will trace

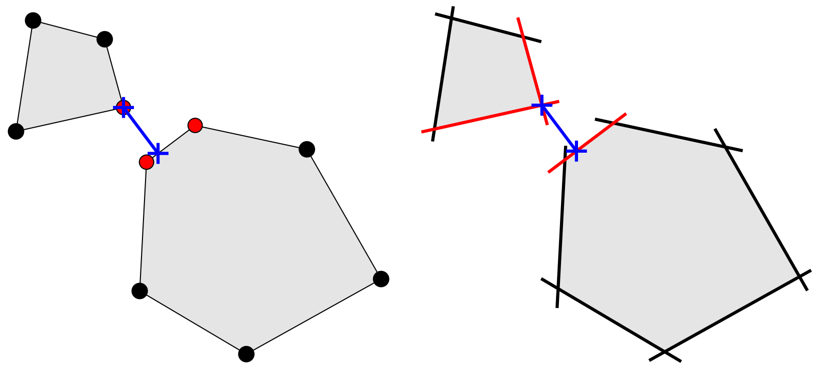


Figure 3.1: *Polytope distance (primal and dual version)*

only bases of size $d+2$ at most. This proves that the closest features of two d-polytopes are always determined by at most $d+2$ points. (If the polytopes have positive distance, $d+1$ points suffice.)

Note that the $n \times n$-matrix D in (3.1) is dense. An equivalent formulation of (PD) can be obtained by introducing a d-vector y of additional variables with $y = Cx$, resulting in

$$\begin{array}{lll} (\mathrm{PD}') & \text{minimize} & y^T y \\ & \text{subject to} & y = Cx \\ & & \sum_{i=1}^{r} x_i = 1 \\ & & \sum_{i=r+1}^{n} x_i = 1 \\ & & x \geq 0\,. \end{array} \tag{3.2}$$

This quadratic program has $n+d$ variables and $d+2$ constraints, but now the objective function has a sparse representation.

The dual version of the polytope distance problem is the following. Given two sets of halfspaces $G = \{\, g_1^-, \ldots, g_r^- \}$ and $H = \{\, h_1^-, \ldots, h_s^- \}$ in dimension d with $r+s = n$, find points $g \in \bigcap_{i=1}^{r} g_i^-$ and $h \in \bigcap_{j=1}^{s} h_j^-$ such that $\|g-h\|$ is minimized, see Figure 3.1 (dual version). A point x is contained in halfspace g_i^- if the function $g_i(x)$ defining the corresponding hyperplane is non-positive (the same holds for h_j^-). We obtain the quadratic program

$$\begin{array}{llll} (\mathrm{PD}'') & \text{minimize} & (x-y)^T(x-y) & \\ & \text{subject to} & g_i\,(x) \leq 0 & i = 1 \ldots r \\ & & h_j(y) \leq 0 & j = 1 \ldots s\,, \end{array} \tag{3.3}$$

where the d-vectors x and y are the $2d$ variables. The n constraints are linear, since the expressions $g_i(x) \leq 0$ and $h_j(y) \leq 0$ boil down to inner product computations.

3.2 Smallest Enclosing Ball

Let $P = \{\, p_1, \ldots, p_n \,\}$ be an n-point set. The *smallest enclosing ball* problem is to find a point p^* such that $\max_{i=1}^{n} \|p_i - p^*\|$ is minimized. The point p^* is then the center of the smallest enclosing ball of P, see Figure 3.2. The following theorem shows that this problem can be written as a quadratic program, thus generalizing a result of Rajan [83] on the smallest enclosing sphere of $d+1$ points in $\mathbb{R}^d$.

Theorem 3.1 *Given an n-point set $P = \{p_1, \ldots, p_n\}$, define the $d \times n$-matrix $C := (p_1, \ldots, p_n)$, consider the quadratic program*

$$\begin{array}{lll} (\mathrm{MB}) & \text{minimize} & x^T C^T C\, x - \sum_{i=1}^{n} p_i^T p_i\, x_i \\ & \text{subject to} & \sum_{i=1}^{n} x_i = 1 \\ & & x \geq 0\,, \end{array} \tag{3.4}$$

and let $(x_1^*, \ldots, x_n^*)$ be some optimal solution. Then the point

$$p^* = \sum_{i=1}^{n} p_i x_i^*,$$

is the center of the smallest enclosing ball of P and the negative value of the objective function at x^* is the squared radius of this ball.

Proof. By the Karush-Kuhn-Tucker conditions for convex quadratic optimization problems (Theorem 2.2), there exists a 1-vector (i.e. a number) λ and an n-vector $\mu \geq 0$, such that

$$2\, {x^*}^T C^T p_k - p_k^T p_k = -\lambda + \mu_k \qquad k = 1 \ldots n\,, \tag{3.5}$$

$${x^*}^T \mu = 0, \tag{3.6}$$

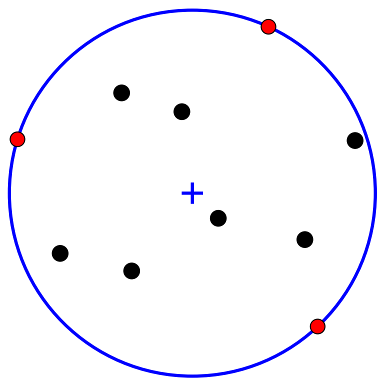


Figure 3.2: *Smallest enclosing ball*

if and only if x^* is optimal for (MB). Using $Cx^* = \sum_{i=1}^n p_i x_i^*$ gives

$$\| p_k - p^* \|^2 = p_k^T p_k - 2(Cx^*)^T p_k + {p^*}^T p^* = \lambda - \mu_k + {p^*}^T p^*.$$

Hence, p^* is the center of a ball B containing P with squared radius at most $\lambda + {p^*}^T p^*$. Moreover, since either $x_k^* = 0$ or $\mu_k = 0$ for all $k \in \{1, \dots, n\}$ by (3.6), there exists some $\mu_j = 0$ and the squared radius of B is exactly $r^2 := \lambda + {p^*}^T p^*$. Let S be the set of points p_j with positive coefficients, i.e. those points of P that lie on the boundary of B (because $x_j^* > 0$ implies $\mu_j = 0$). Thus, $p^* \in \text{conv}(S)$ and B is the smallest enclosing ball of P by Lemma 3.2.

Summing up (3.5) multiplied by x_k^* gives

$$2\, {x^*}^T C^T C\, x^* - \sum_{k=1}^n p_k^T p_k\, x_k^* = \sum_{k=1}^n (-\lambda + \mu_k)\, x_k^* = -\lambda.$$

The second equation holds because $\sum_{k=1}^n x_k^* = 1$ and either $x_k^* = 0$ or $\mu_k = 0$. Subtracting ${p^*}^T p^*$ from both sides completes the proof. □

The following lemma is well known. We use a proof of Seidel [93].

Lemma 3.2 *Let S be a set of points on the boundary of some ball B with center c. Then B is the smallest enclosing ball of S, if and only if c lies in the convex hull of S.*

Proof. Let u be some unit vector and $\mu_u := \min_{p \in S}\{\,(p-c)^T u\,\}$. For $\lambda \geq 0$, define $x_u(\lambda) := c + \lambda u$ and consider the smallest ball $B_u(\lambda)$ containing S with center $x_u(\lambda)$. For all $p \in S$,

$$\|x_u(\lambda)-p\|^2 = (c-p)^T(c-p) + \lambda^2 u^T u - 2\lambda(c-p)^T u$$

is maximized when the third term is minimized, i.e. if $(c-p)^T u = \mu_u$. Thus, the squared radius of $B_u(\lambda)$ is

$$r_u(\lambda) := R^2 - \mu_u^2 + (\lambda - \mu_u)^2.$$

Observe, every point x can be uniquely represented in the form $x_u(\lambda)$ with $\lambda \geq 0$ (except for $x = c$, where the representation is not unique). The squared radius $r_u(\lambda)$ is minimized for $\lambda = \mu_u$ if $\mu_u \geq 0$, and for $\lambda = 0$ otherwise. Furthermore, $\mu_u < 0$ holds if and only if there exists no hyperplane orthogonal to u which separates c and S.

If c lies in the convex hull of S then c is not separable from S. Thus, $\mu_u < 0$ implies $\lambda = 0$ and $B = B_u(0)$ is the smallest enclosing ball of S. On the other hand, if $c \notin \mathrm{conv}(S)$, there exists a unit vector u orthogonal to some hyperplane separating c and S. Here, $B_u(\mu_u)$ contains S but has smaller radius than B. □

As before in the polytope distance problem, the matrix $D = C^T C$ has rank at most d, so Theorem 2.6 shows that the optimal basis has size at most $d+1$, proving that $d+1$ points suffice to determine the smallest enclosing ball.

Again, D is a dense $n \times n$-matrix. At the cost of adding d additional constraints and d additional variables, the following equivalent formulation is obtained.

$$\begin{array}{lll} (\mathrm{MB}') & \text{minimize} & y^T y - \sum_{i=1}^n p_i^T p_i\, x_i \\ & \text{subject to} & y = Cx \\ & & \sum_{i=1}^n x_i = 1 \\ & & x \geq 0\,. \end{array} \tag{3.7}$$

3.3 Smallest Enclosing Annulus

An *annulus* is the region between two concentric spheres. The *smallest enclosing annulus* is the annulus of minimal difference between its squared radii that covers a given point set, see Figure 3.3. In two dimensions, this is the annulus of smallest area. Compared to the minimum-width annulus [2] having minimal difference between its radii, our optimization criterion can be seen as minimizing that difference scaled by the 'size' of the annulus, namely by the sum of the two radii.

For an n-point set $P = \{ p_1, \dots, p_n \}$ in d-space, let r and R denote the small respectively large radius of the annulus covering P, and let c be the annulus' center. The objective is to minimize $R^2 - r^2$ subject to the constraints

$$r \leq \|p_i - c\| \leq R \qquad i = 1 \dots n\,,$$

equivalently

$$r^2 \leq (p^i_1 - c_1)^2 + \dots + (p^i_d - c_d)^2 \leq R^2 \qquad i = 1 \dots n\,,$$

using $p_i = (p^i_1, \dots, p^i_d)$ and $c = (c_1, \dots, c_d)$. Defining

$$\alpha := r^2 - (c_1^2 + \dots + c_d^2)\,, \qquad \beta := R^2 - (c_1^2 + \dots + c_d^2)\,,$$

yields a linear program in $d+2$ variables and $2n$ constraints,

$$\begin{array}{lll} \text{(MA)} & \text{minimize} & \beta - \alpha \\ & \text{subject to} & \sum_{j=1}^{d} 2p^i_j\, c_j \leq p_i{}^T p_i - \alpha \qquad i = 1 \dots n \\ & & \sum_{j=1}^{d} 2p^i_j\, c_j \geq p_i{}^T p_i - \beta \qquad i = 1 \dots n\,, \end{array} \tag{3.8}$$

where α, β, and $c_1, \dots, c_d$ are unknown. From an optimal solution $(\alpha^*, \beta^*, c^*_1, \dots, c^*_d)$ to (MA) the squared radii r^2 and R^2 can be computed as

$$r^2 = \alpha^* + \|\, c^* \|^2\,, \qquad R^2 = \beta^* + \|\, c^* \|^2\,.$$

There is also a dual version of (3.8), namely the following linear program in $2n$ variables $\lambda := (\lambda_1, \dots, \lambda_n)$ and $\mu := (\mu_1, \dots, \mu_n)$ with $d+2$ constraints

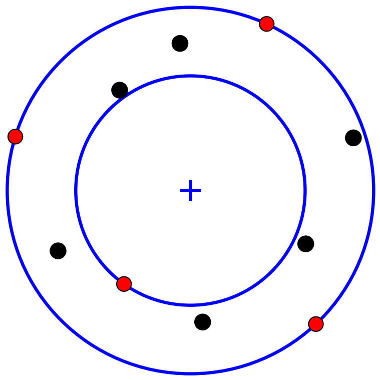


Figure 3.3: *Smallest enclosing annulus*

$$\begin{array}{lll} \text{(MA')} & \text{minimize} & \sum_{i=1}^n {p^i}^T p^i \, \mu_i - \sum_{i=1}^n {p^i}^T p^i \, \lambda_i \\ & \text{subject to} & \sum_{i=1}^n 2p^i_j \, \lambda_i + \sum_{i=1}^n 2p^i_j \, \mu_i = 0 \qquad j = 1 \ldots d \\ & & \sum_{i=1}^n \lambda_i = 1 \\ & & \sum_{i=1}^n \mu_i = 1 \\ & & \lambda, \mu \geq 0 \,. \end{array} \tag{3.9}$$

3.4 Optimal Separating Hyperplane

Given two point sets P and Q with a total of n points in d-dimensional space, test whether they can be separated by a hyperplane, and if so, find a separating hyperplane that maximizes the distance to the nearest point, see Figure 3.4. The separation itself can be done by LP, but finding the optimal hyperplane is QP. The problem can be reduced to the primal version of the polytope distance problem. The optimal hyperplane is just the bisector of the segment connecting p and q (cf. Section 3.1)

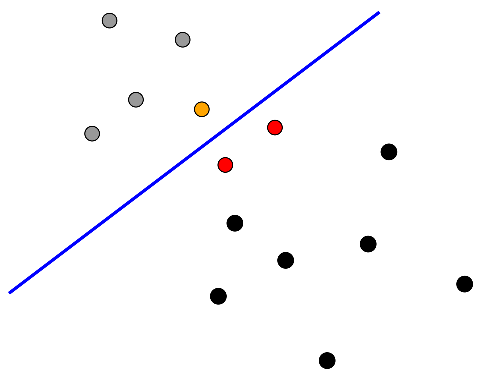


Figure 3.4: *Optimal separating hyperplane*

3.5 Conclusion

We presented four geometric optimization problems that can be formulated as quadratic programs (one of them even as a linear program). All problems share the property of having $\min(n, m)$ small. For the QP problems the rank of D is bounded by the dimension d of the ambient geometric space, which is usually small in the applications. Hence, our QP simplex algorithm described in the previous chapter is suitable to solve these problems.

Chapter 4

A Non-Quadratic Geometric Optimization Problem

A natural generalization of the smallest enclosing ball problem is that of finding the smallest enclosing ellipsoid of a set of points. It is an instance of convex programming and can be solved by general methods in time $O(n)$ if the dimension is fixed. The problem-specific parts of these methods are encapsulated in primitive operations that deal with subproblems of constant size. We derive explicit formulae for the primitive operations of Welzl's randomized method [105] in dimension $d = 2$. The formulae contain only rational expressions, allowing for an exact solution.

4.1 Smallest Enclosing Ellipsoid

Given a point $c \in \mathbb{R}^d$ and a positive definite matrix[1] $M \in \mathbb{R}^{d \times d}$, the set of points $x \in \mathbb{R}^d$ satisfying

[1]i.e. $x^T M\, x > 0$ holds for all $x \neq 0$.

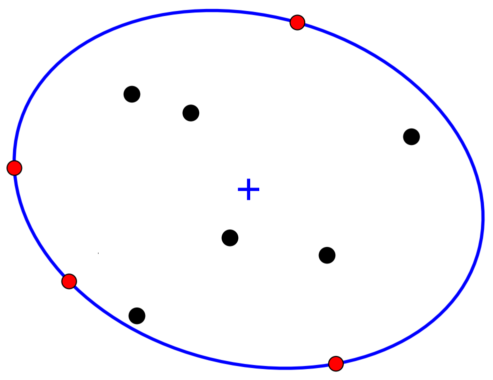


Figure 4.1: *Smallest enclosing ellipsoid*

$$(x - c)^T M\,(x - c) = 1 \tag{4.1}$$

defines an *ellipsoid* with *center* c. The function $f(x) = (x-c)^T M\,(x-c)$ is called the *ellipsoid function*, the set $E = \{x \in \mathbb{R}^d \mid f(x) \leq 1\}$ is the *ellipsoid body*. Note, varying a pair of entries symmetric to the main diagonal of M in (4.1) does not change the ellipsoid as long as the sum remains the same. In the sequel, we assume M to be symmetric, w.l.o.g. The volume of the ellipsoid body E is

$$\mathrm{Vol}(E) = \frac{\mathrm{Vol}(S_d)}{\sqrt{\det(M)}}\,, \tag{4.2}$$

where S_d is the d-dimensional unit sphere. This can be easily seen by choosing the coordinate system according to the principal axes, such that the defining matrix M becomes diagonal [88].

Given a point set $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^d$, we are interested in the ellipsoid body of smallest volume containing P. Identifying the body with its generating ellipsoid, we call this the *smallest enclosing ellipsoid* of P, denoted by SmEll(P), see Figure 4.1. (If P does not span $\mathbb{R}^d$, then SmEll(P) is a lower-dimensional ellipsoid 'living' in the affine hull of P). The problem of finding SmEll(P) can be written as the convex program [76, Section 6.5]

$$\begin{array}{lll} \text{(ME)} & \text{minimize} & -\log\det(M) \\ & \text{subject to} & (p_i - c)^T M\,(p_i - c) \leq 1 \quad \forall\, p_i \in P \\ & & M \text{ positive definite}, \end{array} \tag{4.3}$$

where the entries of M and c are unknown. Since M is assumed to be symmetric, (ME) has $d(d+3)/2$ variables and n constraints. The objective function is quadratic only if $d = 2$, while the constraints are convex sets. Thus, (ME) is not a quadratic program and we cannot use our QP simplex method to solve it. Instead, we will use an algorithm of Welzl [105] described in the next section.

4.2 Welzl's Method

Let us briefly describe Welzl's randomized algorithm for computing the smallest enclosing ellipsoid of an n-point set in d-space [105]. The algorithm is very simple and achieves an optimal expected runtime of $O(n)$ if d is constant.

We start with the following important facts (proofs of which may be found in [88, 62]). By $\textsc{SmEll}(Q, R)$ we denote the smallest ellipsoid containing Q that has R on the boundary.

Proposition 4.1

(i) If there is any ellipsoid with R on its boundary that encloses Q, then $\textsc{SmEll}(Q, R)$ exists and is unique.

(ii) If $\textsc{SmEll}(Q, R)$ exists and $q \notin \textsc{SmEll}(Q \setminus \{q\}, R)$, then $\textsc{SmEll}(Q \setminus \{q\}, R \cup \{q\})$ exists and equals $\textsc{SmEll}(Q, R)$.

(iii) If $\textsc{SmEll}(Q, R)$ exists, then there is a subset $S \subseteq Q$ with $|S| \leq \max(0, d(d+3)/2 - |R|)$ and $\textsc{SmEll}(Q, R) = \textsc{SmEll}(S, R) = \textsc{SmEll}(\emptyset, S \cup R)$.

By (iii), $\textsc{SmEll}(P)$ is always determined by a support set $S \subseteq P$ with at most $\delta := d(d+3)/2$ support points. The value of δ coincides with the number of free variables in the ellipsoid parameters M and c.

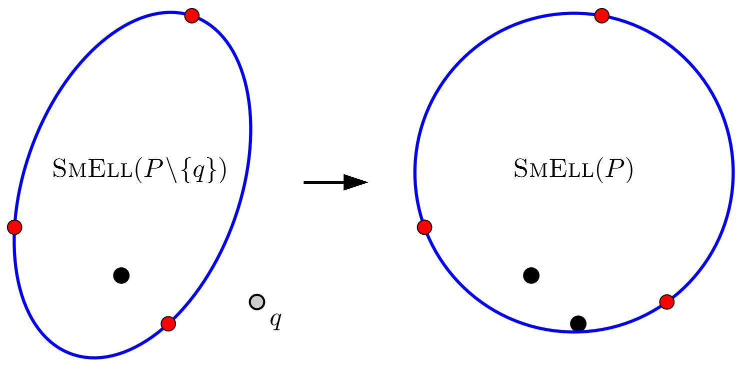


Figure 4.2: *The inductive step in Welzl's algorithm*

The idea of Welzl's algorithm for computing SmEll(P) is as follows: if P is empty, SmEll(P) is the empty set by definition. If not, choose a point $q \in P$ and recursively determine $E :=$ SmEll($P\backslash\{q\}$). If $q \in E$, then $E =$ SmEll(P) and we are done. Otherwise, q must lie on the boundary of SmEll(P), and we get SmEll(P) = SmEll($P\backslash\{q\}, \{q\}$), see Figure 4.2. Computing the latter (in the same way) is now an easier task because one degree of freedom has been eliminated. If the point q to be removed for the recursive call is chosen *uniformly at random* among the points in Q, we arrive at the following randomized algorithm. To compute SmEll(P), we call the procedure with the pair $(P, \emptyset)$.

Algorithm 4.2 *(computes* SmEll(Q, R)*, if it exists)*

```
SmEll($Q, R$):
   IF $Q = \emptyset$ OR $|R| = \delta$ THEN
      RETURN SmEll($\emptyset, R$)
   ELSE
      choose $q \in Q$ uniformly at random
      $E :=$ SmEll($Q\backslash\{q\}, R$)
      IF $q \in E$ THEN
         RETURN $E$
      ELSE
         RETURN SmEll($Q\backslash\{q\}, R \cup \{q\}$)
      END
   END
```

Termination of the procedure is immediate because the recursive calls decrease the size of Q. Correctness follows from Proposition 4.1 and the observation that the algorithm – when called with $(P, \emptyset)$ – maintains the invariant '$\textsc{SmEll}(Q, R)$ exists'. To justify the termination criterion '$|R| = \delta$', we need the following lemma proving that in this case only one ellipsoid E with R on the boundary exists, i.e. we must have $E = \textsc{SmEll}(\emptyset, R) = \textsc{SmEll}(Q, R)$. This is remarkable, because in general, an ellipsoid is *not* uniquely determined by any δ points on the boundary (for example, consider $\delta - 1$ points on the boundary of a $(d-1)$-dimensional ellipsoid E' and some additional point q; then there are many d-dimensional ellipsoids through E and q).

Lemma 4.3 *If R attains cardinality δ during a call to* $\textsc{SmEll}(P, \emptyset)$, *exactly one ellipsoid E with R on its boundary exists.*

Proof. By expanding (4.1), we see that an ellipsoid is a special *second order surface* of the form

$$\{\, p \in \mathbb{R}^d \mid p^T M p + 2\, p^T m + w = 0 \,\},$$

defined by $\delta + 1$ parameters $M \in \mathbb{R}^{d \times d}$ (symmetric), $m \in \mathbb{R}^d$, $w \in \mathbb{R}$.

For a point set $R \subseteq \mathbb{R}^d$ let $\mathcal{S}(R)$ denote the set of $(\delta+1)$-tuples of parameters that define second order surfaces through all points in R. It is clear that $\mathcal{S}(R)$ is a vector space, and we define the *degree of freedom* w.r.t. R to be $\dim(\mathcal{S}(R)) - 1$. Obviously, the degree of freedom is at least $\delta - |R|$, since any point in R introduces one linear relation between the parameters.

Now we claim that during Algorithm 4.2, the degree of freedom w.r.t. R is always exactly $\delta - |R|$. This is clear for $R = \emptyset$. Moreover, if q is added to R in the second recursive call of the algorithm, the degree of freedom goes down, which proves the claim. To see this, assume on the contrary that $\dim(\mathcal{S}(R)) = \dim(\mathcal{S}(R \cup \{q\}))$, hence $\mathcal{S}(R) = \mathcal{S}(R \cup \{q\})$. Then it follows that q already lies on any second order surface through R, in particular on $\textsc{SmEll}(Q \setminus \{q\}, R)$. But then the second recursive call would not have been made, a contradiction.

Now the claim of the lemma follows: if $|R| = \delta$, the degree of freedom is 0, i.e. $\mathcal{S}(R)$ has dimension 1. Since a second order surface is invariant under scaling its parameters, this means that there is a unique second order surface, in this case an ellipsoid, through R. $\square$

The primitive operations of Welzl's method are the computation of $E = \textsc{SmEll}(\emptyset, R)$ and the test $q \in E$. As we will see in the next section, they can be combined into one single operation. Before, we describe a heuristic for tuning the algorithm that has proven to be very efficient in practice.

The move-to-front heuristic. There are point sets on which the algorithm does not perform substantially better than expected; on such point sets, the exponential behavior in $\delta = \Theta(d^2)$ leads to slow implementations already for small d. Although for $d = 2$ the actual runtime is still tolerable for moderately large n, a dramatic improvement (leading to a practically efficient solution for large n as well) is obtained under the so-called *move-to-front* heuristic. This variant keeps the points in an ordered list (initially random). In the first recursive call, q is chosen to be the last point in the list (restricted to the current subset of the points). If the subsequent in-ellipsoid test reveals $q \notin \textsc{SmEll}(Q \setminus \{q\}, R)$, q is moved to the front of the list, after the second recursive call to $\textsc{SmEll}(Q \setminus \{q\}, R \cup \{q\})$ has been completed. Although the move-to-front heuristic does not eliminate the algorithm's exponential behavior in δ, it significantly reduces the number of primitive operations to be called. See [105] for further details and computing times.

4.3 Exact Primitives in the Plane

In the two-dimensional case, the constant-size problems of Algorithm 4.2 involve *smallest enclosing ellipse*s defined by up to five support points, where the difficult case arises when the ellipse is defined by four support points. As we show below, even if the points have rational coordinates, the ellipse will typically have not, so in order to stay with rational expressions, an explicit evaluation of the ellipse has to be avoided.

For a given point set P, Welzl's method computes a support set S of P, provided the following primitive operation is available.

Given $R \subseteq P$, $3 \leq |R| \leq 5$, such that
$\textsc{SmEll}(\emptyset, R)$ exists, and a query point $q \in P \setminus R$, decide
whether q lies inside $\textsc{SmEll}(\emptyset, R)$.

This operation — we call it the *in-ellipse test* — can be reduced to a sign evaluation of a certain derivative. This leads to an elegant and efficient method whose computational primitives are in-ellipse tests over rational ellipses and evaluations of derivatives at rational values.

Our method to deal with is based on the concept of conics.

4.3.1 Conics

A *conic* $\mathcal{C}$ in *linear form* is the set of points $p = (x, y)^T \in \mathbb{R}^2$ satisfying the quadratic equation

$$\mathcal{C}(p) := rx^2 + sy^2 + 2\,txy + 2\,ux + 2\,vy + w = 0\,, \tag{4.4}$$

r, s, t, u, v, w being real parameters. $\mathcal{C}$ is invariant under scaling the vector (r, s, t, u, v, w) by any nonzero factor. After setting

$$M := \begin{pmatrix} r & t \\ t & s \end{pmatrix}, \qquad m := \begin{pmatrix} u \\ v \end{pmatrix}, \tag{4.5}$$

the conic assumes the form

$$\mathcal{C} = \{\, p^T M\, p + 2\, p^T m + w = 0 \,\}\,. \tag{4.6}$$

If a point $c \in \mathbb{R}^2$ exists such that $Mc = -m$, $\mathcal{C}$ is symmetric about c and can be written in *center form* as

$$\mathcal{C} = \{\, (p-c)^T M\, (p-c) - z = 0 \,\}\,, \tag{4.7}$$

where $z = c^T M\, c - w$. If $\det(\mathcal{C}) := \det(M) \neq 0$, a *center* exists and is unique. Conics with $\det(\mathcal{C}) > 0$ define *ellipse*s.

By scaling with -1 if necessary, we can assume w.l.o.g. that $\mathcal{C}$ is *normalized*, i.e. $r \geq 0$. If $\mathcal{E}$ is a normalized ellipse, q lies inside $\mathcal{E}$ if and only if $\mathcal{E}(q) \leq 0$.

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two conics, then the *linear combination*

$$\mathcal{C} := \lambda\, \mathcal{C}_1 + \mu\, \mathcal{C}_2, \quad \lambda, \mu \in \mathbb{R}$$

is the conic given by $\mathcal{C}(p) = \lambda\mathcal{C}_1(p) + \mu\mathcal{C}_2(p)$. If p belongs to both $\mathcal{C}_1$ and $\mathcal{C}_2$, then p also belongs to $\mathcal{C}$.

Now we are prepared to describe the in-ellipse test, for $|R| = 3, 4, 5$.

4.3.2 In-ellipse Test, $|R| = 3$

It is well-known [101, 80, 88] that $\textsc{SmEll}(\emptyset, \{p_1, p_2, p_3\})$ is given in center form (4.7) by

$$c = \frac{1}{3}\sum_{i=1}^{3} p_i, \qquad M^{-1} = \frac{1}{3}\sum_{i=1}^{3}(p_i - c)(p_i - c)^T, \qquad z = 2\,.$$

From this, M is easy to compute. Query point q lies inside $\textsc{SmEll}(\emptyset, R)$ if and only if $(q-c)^T M\,(q-c) - z \leq 0$.

4.3.3 In-ellipse Test, $|R| = 4$

$\textsc{SmEll}(\emptyset, R)$ is some conic through $R = \{\, p_1, p_2, p_3, p_4 \,\}$ (R being in convex position). Any such conic is a linear combination of two special conics $\mathcal{C}_1$ and $\mathcal{C}_2$ through R, see Figure 4.3 [98]. To see that these are indeed conics, consider three points $q_1 = (x_1, y_1)$, $q_2 = (x_2, y_2)$, and $q_3 = (x_3, y_3)$ and define

$$[q_1 q_2 q_3] := \det\begin{pmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{pmatrix}. \tag{4.8}$$

$[q_1 q_2 q_3]$ records the orientation of the point triple; in particular, if the points are collinear, then $[q_1 q_2 q_3] = 0$. This implies

$$\mathcal{C}_1(p) = [p_1 p_2 p][p_3 p_4 p]\,, \qquad \mathcal{C}_2(p) = [p_2 p_3 p][p_4 p_1 p]\,,$$

and these turn out to be quadratic expressions as required in the conic equation (4.4), easily computable from the points in R.

Now, given the query point $q \notin R$, there exists a unique conic $\mathcal{C}_0$ through the five points $R \cup \{q\}$, since R is an (intermediate) support set of Algorithm 4.2, see also [98]. We can compute this conic as $\mathcal{C}_0 = \lambda_0 \mathcal{C}_1 + \mu_0 \mathcal{C}_2$, with $\lambda_0 := \mathcal{C}_2(q)$ and $\mu_0 := -\mathcal{C}_1(q)$. In the sequel we assume that $\mathcal{C}_0$ is normalized. Depending on the type of $\mathcal{C}_0$ we distinguish two cases.

Case 1. $\mathcal{C}_0$ is not an ellipse, i.e. $\det(\mathcal{C}_0) \leq 0$. Then we have the following result.

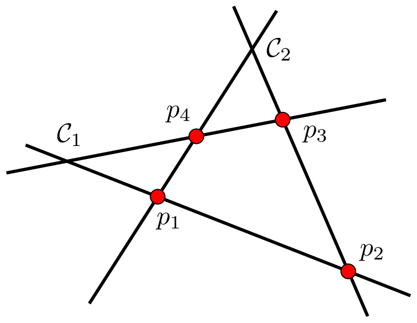


Figure 4.3: *Two special conics through four points*

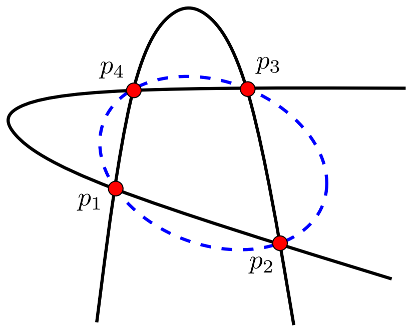


Figure 4.4: *The two parabolas through four points*

Lemma 4.4 *Exactly one of the following statements holds.*

(i) q lies inside any ellipse through R.

(ii) q lies outside any ellipse through R.

Let us give some intuition, before we formally prove the lemma. Since no three of the four support points are collinear, there exist two (possibly degenerate) parabolas through these points, see Figure 4.4. These parabolas cut the plane into regions which determine the type of $\mathcal{C}_0$. Only if q lies strictly inside one parabola and strictly outside the other, $\mathcal{C}_0$ is an ellipse. Otherwise, q either lies inside both parabolas in which case q also lies inside all ellipses through p_1, p_2, p_3, and p_4, or q lies outside both parabolas, also being outside all the ellipses.

Proof. Assume there are two ellipses $\mathcal{E}$ and $\mathcal{E}'$ through R, with $\mathcal{E}(q) \leq 0$ and $\mathcal{E}'(q) > 0$. Then we find $\lambda \in [0,1)$ such that $\mathcal{E}'' := (1-\lambda)\mathcal{E}+\lambda\mathcal{E}'$ satisfies $\mathcal{E}''(q)=0$, i.e. $\mathcal{E}''$ goes through $R \cup \{q\}$. Thus $\mathcal{E}''$ equals $\mathcal{C}_0$ and is not an ellipse. On the other hand, the convex combination of two ellipses is an ellipse again [88, Chapter 1], a contradiction. $\square$

Lemma 4.4 shows that it suffices to test q against *any* ellipse through the four points to obtain the desired result. Let

$$\alpha := r_1 s_1 - t_1^2\,, \qquad \beta := r_1 s_2 + r_2 s_1 - 2\,t_1 t_2\,, \qquad \gamma := r_2 s_2 - t_2^2\,,$$

where r_i, s_i, t_i are the parameters of $\mathcal{C}_i$ in the linear form (4.4). Then $\mathcal{E} := \lambda\mathcal{C}_1+\mu\mathcal{C}_2$ with $\lambda := 2\gamma-\beta$ and $\mu := 2\alpha-\beta$ defines such an ellipse. For this, one observes that

$$\det(\mathcal{E}) = (4\alpha\gamma - \beta^2)(\alpha + \gamma - \beta).$$

We will show that both factors have negative sign, thus proving that the choice of λ and μ indeed yields an ellipse $\mathcal{E}$.

With definition (4.8) we can check that

$$4\alpha\gamma - \beta^2 = -[p_1p_2p_3][p_2p_3p_4][p_3p_4p_1][p_4p_1p_2].$$

After a preprocessing, one can assume that p_1, p_2, p_3, p_4 are in counterclockwise order (they must be in convex position, because otherwise $\textsc{SmEll}(\emptyset, \{p_1, p_2, p_3, p_4\})$ does not exist). This means, each bracketed term has positive sign, and $4\alpha\gamma-\beta^2 < 0$ follows. Moreover, we can easily verify that

$$\alpha + \gamma - \beta = [p_2p_4p_1][p_2p_4p_3] - (\kappa_1+\kappa_2)^2,$$

where

$$\kappa_1 := ((x_1-x_2)(y_3-y_4) - (y_1-y_2)(x_3-x_4))/2$$
$$\kappa_2 := ((x_2-x_3)(y_4-y_1) - (y_2-y_3)(x_4-x_1))/2$$

with $p_i=(x_i,y_i)$, $i=1\ldots4$. The product $[p_2p_4p_1][p_2p_4p_3]$ is negative because p_1 and p_3 lie on different sides of the diagonal $\overline{p_2p_4}$. It follows that $\alpha+\gamma-\beta < 0$ and finally $\det(\mathcal{E}) > 0$.

Case 2. $\mathcal{C}_0$ is an ellipse $\mathcal{E}$, i.e. $\det(\mathcal{C}_0) > 0$. We need to check the position of q relative to $\mathcal{E}^* := \textsc{SmEll}(\emptyset, R)$, given by

$$\mathcal{E}^* = \lambda^* \mathcal{C}_1 + \mu^* \mathcal{C}_2,$$

with unknown parameters λ^* and μ^*. In the form of (4.4), $\mathcal{E}$ is determined by $r_0, \ldots, w_0$, where $r_0 = \lambda_0 r_1 + \mu_0 r_2$. By scaling the representation of $\mathcal{E}^*$ accordingly, we can also assume that $r_0 = \lambda^* r_1 + \mu^* r_2$ holds. In other words, $\mathcal{E}^*$ is obtained from $\mathcal{E}$ by varying λ and μ along the line $\{\lambda r_1 + \mu r_2 = r_0\}$. This means,

$$\begin{pmatrix} \lambda^* \\ \mu^* \end{pmatrix} = \begin{pmatrix} \lambda_0 \\ \mu_0 \end{pmatrix} + \tau^* \begin{pmatrix} -r_2 \\ r_1 \end{pmatrix} \tag{4.9}$$

for some $\tau^* \in \mathbb{R}$. Define

$$\mathcal{E}^\tau := (\lambda_0 - \tau r_2)\, \mathcal{C}_1 + (\mu_0 + \tau r_1)\, \mathcal{C}_2, \quad \tau \in \mathbb{R}\,.$$

Then $\mathcal{E}^0 = \mathcal{E}$ and $\mathcal{E}^{\tau^*} = \mathcal{E}^*$. The function $g(\tau) := \mathcal{E}^\tau(q)$ is linear, hence we get

$$\mathcal{E}^*(q) = \tau^* \left. \frac{\partial}{\partial \tau} \mathcal{E}^\tau(q) \right|_{\tau=0} = \rho\, \tau^*,$$

where $\rho := \mathcal{C}_2(q) r_1 - \mathcal{C}_1(q) r_2$. Consequently, q lies inside $\textsc{SmEll}(\emptyset, R)$ if and only if $\rho\, \tau^* \leq 0$.

The following lemma is proved in [22], see also [88].

Lemma 4.5 *Consider two ellipses $\mathcal{E}_1$ and $\mathcal{E}_2$, and let*

$$\mathcal{E}^\lambda := (1-\lambda)\mathcal{E}_1 + \lambda \mathcal{E}_2$$

be a convex combination, $\lambda \in (0,1)$. Then $\mathcal{E}^\lambda$ is an ellipse satisfying

$$\mathrm{Vol}(\mathcal{E}^\lambda) < \max(\mathrm{Vol}(\mathcal{E}_1), \mathrm{Vol}(\mathcal{E}_2)).$$

Since $\mathcal{E}^\tau$ is a convex combination of $\mathcal{E}$ and $\mathcal{E}^*$ for τ ranging between 0 and τ^*, the volume of $\mathcal{E}^\tau$ decreases as τ goes from 0 to τ^*, hence

$$\mathrm{sgn}(\tau^*) = -\mathrm{sgn}\left(\left. \frac{\partial}{\partial \tau} \mathrm{Vol}(\mathcal{E}^\tau) \right|_{\tau=0} \right).$$

If $\mathcal{E}^\tau$ is given in center form (4.7), its area is

$$\mathrm{Vol}(\mathcal{E}^\tau) = \frac{\pi}{\sqrt{\det(M/z)}},$$

following from (4.2). Consequently,

$$\mathrm{sgn}\left(\frac{\partial}{\partial\tau}\mathrm{Vol}(\mathcal{E}^\tau)\bigg|_{\tau=0}\right) = -\mathrm{sgn}\left(\frac{\partial}{\partial\tau}\det(M/z)\bigg|_{\tau=0}\right).$$

Recall that if M and m collect the parameters of $\mathcal{E}^\tau$ as in (4.5) with $c = M^{-1}m$ being its center, we get $z = c^TMc - w = m^TM^{-1}m - w$, where M, m, and w depend on τ (which we omit in the sequel, for the sake of readability). Noting that

$$M^{-1} = \frac{1}{\det(M)}\begin{pmatrix} s & -t \\ -t & r \end{pmatrix},$$

we get

$$z = \frac{1}{\det(M)}(u^2s - 2\,uvt + v^2r) - w\,.$$

Let us introduce the following abbreviations.

$$d := \det(M), \qquad Z := u^2s - 2\,uvt + v^2r\,.$$

With primes (d', Z', etc.) we denote derivatives w.r.t. τ. Now we can write

$$\frac{\partial}{\partial\tau}\det(M/z) = \left(\frac{d}{z^2}\right)' = \frac{d'z - 2\,dz'}{z^3}\,. \tag{4.10}$$

Since $d(0)$ and $z(0)$ are positive (recall that $\mathcal{E}$ is a normalized ellipse), this is equal in sign to

$$\sigma := d\,(d'z - 2\,dz')\,,$$

at least when evaluated for $\tau = 0$, which is the value we are interested in. Furthermore, we have

$$\begin{aligned} d'z &= d'(\frac{1}{d}Z - w) = \frac{d'}{d}Z - d'w\,, \\ dz' &= d\,(\frac{Z'd - Zd'}{d^2} - w') = \frac{Z'd - Zd'}{d} - dw'\,, \end{aligned}$$

hence

$$\sigma = d'Z - dd'w - 2\,(Z'd - Zd' - d^2w') = 3\,d'Z + d\,(2\,dw' - d'w - 2\,Z')\,.$$

Rewriting Z as $u\,(us-vt) + v\,(vr-ut) =: uZ_1 + vZ_2$, we get

$$\begin{aligned} d &= rs - t^2\,, & Z' &= u'Z_1 + uZ_1' + v'Z_2 + vZ_2'\,,\\ d' &= r's + rs' - 2\,t\,t'\,, & Z_1' &= u's + us' - v't - vt'\,,\\ & & Z_2' &= v'r + vr' - u't - ut'\,. \end{aligned}$$

For $\tau = 0$, all these values can be computed directly from $r(0), \ldots, w(0)$ (the defining values of $\mathcal{E}$) and their corresponding derived values $r'(0), \ldots, w'(0)$. For the latter we get $r'(0) = 0$, $s'(0) = r_1s_2 - r_2s_1$, $\ldots$, $w'(0) = r_1w_2 - r_2w_1$. We obtain that q lies inside $\textsc{SmEll}(\emptyset, R)$ if and only if $\rho\,\sigma(0) \leq 0$.

Note that for deciding the in-ellipse test in the case $|R| = 4$, it was not necessary to know $\textsc{SmEll}(\emptyset, R)$ explicitly. In fact, $\textsc{SmEll}(\emptyset, R)$ is not even representable with rational coordinates in general, as shown by the following example.

Consider the points $p_1 = (0,0)$, $p_2 = (1,0)$, $p_3 = (1/2, 1)$, and $p_4 = (0,1)$, see Figure 4.5. As noted before, $\textsc{SmEll}(\emptyset, \{p_1,p_2,p_3,p_4\})$ is a linear combination $\lambda\mathcal{C}_1 + \mu\mathcal{C}_2$ of two special conics $\mathcal{C}_1$ and $\mathcal{C}_2$ through the four points, as depicted in Figure 4.3. By explicitly computing these conics, one finds that the linear combination in the form of (4.6) is given by

$$M = \begin{pmatrix} \mu & \mu/4 \\ \mu/4 & -\lambda/2 \end{pmatrix}, \qquad m = \begin{pmatrix} -\mu/2 \\ \lambda/4 \end{pmatrix}, \qquad w = 0\,.$$

Standard calculus shows that this defines the ellipse of minimum volume through p_1, p_2, p_3, p_4 if and only if $4\lambda = -(3+\sqrt{13})\mu$ holds. This means, the linear form of $\textsc{SmEll}(\emptyset, \{p_1,p_2,p_3,p_4\})$ contains irrational coordinates, no matter how it is scaled. This also holds for the center form. In particular, the center $c = (x_c, y_c)$ evaluates to

$$x_c = \frac{9 + 3\sqrt{13}}{20 + 8\sqrt{13}} \approx 0.406\,, \qquad y_c = \frac{1 + \sqrt{13}}{5 + 2\sqrt{13}} \approx 0.377\,.$$

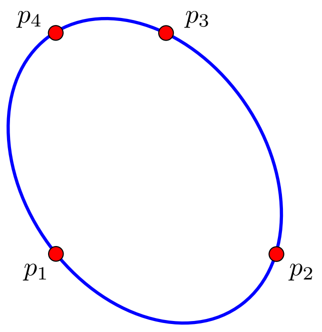


Figure 4.5: *Irrational smallest ellipse through four points*

4.3.4 In-ellipse Test, $|R| = 5$

It is easy to see that in Welzl's method, R attains cardinality five only if before, a test '$p \in \textsc{SmEll}(\emptyset, R \setminus \{p\})$?' has been performed (with a negative result), for some $p \in R$. In the process of doing this test, the unique conic (which we know is an ellipse $\mathcal{E}$) through the points in R has already been computed, see previous subsection. Now we just 'recycle' $\mathcal{E}$ to conclude that q lies inside $\textsc{SmEll}(\emptyset, R)$ if and only if $\mathcal{E}(q) \leq 0$. s

4.4 Conclusion

We have described primitives for Welzl's method leading to a rational arithmetic solution for the problem of computing the smallest enclosing ellipse of a planar point set.

The output of the algorithm is a support set S. In addition, for $|S| \neq 4$, our method determines $\textsc{SmEll}(P) = \textsc{SmEll}(S) = \textsc{SmEll}(\emptyset, S)$ explicitly. For $|S| = 4$, the value τ^* defining $\textsc{SmEll}(\emptyset, S)$ via (4.9) appears among the roots of (4.10); a careful analysis [80, 88] reduces this to a cubic polynomial in τ, thus an exact symbolic representation or a floating point approximation of τ^* and $\textsc{SmEll}(\emptyset, S)$ can be computed in a postprocessing step.

Note that even a number type supporting k-th roots with arbitrary precision like `leda_real` [68] can not handle the smallest enclosing ellipse with four boundary points exactly, because third roots of complex numbers are needed to solve (4.10).

From a practical point of view, the three-dimensional version of the problem is probably most interesting, and one might ask how our techniques apply to this case. Welzl's method as described in Section 4.2 works in any dimension, but the primitive operations are already not sufficiently understood for $d = 3$. First of all, the number of basic cases is larger; we need to do in-ellipsoid tests over ellipsoids defined by $4 \leq k \leq 9$ boundary points. While the extreme cases $k = 4$ and $k = 9$ are easy (they behave similarly to the extreme cases $k = 3, 5$ for $d = 2$), no exact method for any other case is known. Our ideas readily generalize to the case $k = 8$: here we can (as in the planar case) use the fact that eight points – if they appear as a set R during Algorithm 4.2 – determine an ellipsoid up to one degree of freedom, see the proof of Lemma 4.3. Beyond that, it is not clear whether the method generalizes.

In any dimension larger than two, an open problem is to prove the existence of a *rational* expression whose sign tells whether a point $q \in \mathbb{R}^d$ lies inside the smallest ellipsoid determined by $d{+}1 \leq k \leq d(d{+}3)/2$ boundary points. If such an expression exists, how can it be computed, and what is its complexity?

An implementation of Welzl's method in combination with our primitives is available in the Computational Geometry Algorithms Library described in the next chapter.

Part II

Implementation

Chapter 5

CGAL, the Computational Geometry Algorithms Library

The birth of CGAL dates back to a meeting in Utrecht in January 1995. Shortly afterwards, the five authors of [31] started developing the kernel. The development of the whole library has been made possible through the funding of two projects[1] by the European Community. Since the official start of the CGAL project in October 1996, the team of developers has grown considerably. It consists mostly of research assistants, PhD students and postdocs in academia, who are professionals in the field of computational geometry and related areas. They form a heterogeneous team of developers; some of them working part time for CGAL, some of them full time. The CGAL release 2.3 (August 2001) consists of approximately 200,000 lines of C++ source code[2] for the library, plus 100,000 lines for accompanying sources, such as the test suite and example programs. CGAL's WWW home-page[3] `http://www.cgal.org/` pro-

[1]ESPRIT IV LTR Projects No. 21957 (CGAL) and No. 28155 (GALIA)

[2]C++ comments and empty lines are not counted.

[3]The reservation of CGAL's own domain was proposed by the author during the 1999 CGAL Implementation Meeting at Schloß Dagstuhl.

vides a list of publications about CGAL and related research: previous overviews [78], the first design of the geometric kernel [30], recent overviews and descriptions of the current design [31, 53].

5.1 Related Work

An overview on the state of the art of computational geometry software before CGAL including many references is given in [3]. Three approaches of implementing geometric software can be distinguished: collections of standalone implementations, integrated applications, and software libraries.

The approach of collecting isolated or only loosely coupled implementations usually requires some adaptation effort to use and combine such algorithms. Although comparable to the collection of algorithms in the successful *Graphics Gems Series* [49, 4, 59, 52], the adaptation of computational geometry implementations is harder due to the need of more involved data structures and more advanced algorithms. A good collection provides the *Directory of Computational Geometry Software*[4].

Advantages of integrated applications and workbenches are homogeneous environments with animation and interaction capabilities. Disadvantages are monolithic structures which make them hard to extend and hard to reuse in other projects. First implementation efforts were started at the end of the Eighties [29, 24], in particular XYZ GeoBench[5] [77, 92] developed at ETH Zurich is one of CGAL's precursors.

If well designed, the components of a library work seamlessly together. They can be reused in other projects and the library is extensible. Examples are the precursors of CGAL developed by members of the CGAL consortium: PLAGEO and SPAGEO [48], developed at Utrecht University, C++GAL [6], developed at INRIA Sophia-Antipolis, and the geometric part of LEDA[6] [67, 68], developed at Max-Planck-Institut für Informatik, Saarbrücken. Another example is GEOMLIB [7], a computational geometry library implemented in Java at the Center for Geo-

[4]`http://www.geom.umn.edu/software/cglist/`
[5]`http://wwwjn.inf.ethz.ch/geobench/XYZGeoBench.html`
[6]`http://www.mpi-sb.mpg.de/LEDA/`

metric Computing, located at Brown University, Duke University, and John Hopkins University in the United States. They state their goal as *an effective technology transfer from computational geometry to relevant applied fields.*

5.2 Generic Programming

Generic and flexible designs can be achieved following basically one of the two paradigms; *object-oriented programming* or *generic programming.* Both paradigms are supported in C++: Object-oriented programming, using inheritance from base classes with virtual member functions, and generic programming, using class templates and function templates. Both paradigms are also available in other languages, but we stay with the notion used in C++, which is the choice made for CGAL.

The flexibility in the *object-oriented programming paradigm* is achieved with a base class, which defines an interface, and derived classes that implement this interface. Generic functionality can be programmed in terms of the base class and a user can select any of the derived classes wherever the base class is required. The classes actually used can be selected at runtime and the generic functionality can be implemented without knowing all derived classes beforehand. In C++ so-called virtual member functions and runtime type information support this paradigm. The base class is usually a pure virtual base class.

The advantages are the explicit definition of the interface and the runtime flexibility. But there are four main disadvantages: Firstly, the object-oriented programming paradigm cannot provide strong type checking at compile time whenever dynamic casts are used, which is necessary in C++ to achieve flexibility. Secondly, this paradigm enforces tight coupling through the inheritance relationship [61], thirdly, it requires additional memory for each object (in C++ the so-called *virtual function table pointer*) and, fourthly, it adds for each call to a virtual member function an indirection through the virtual function table [63]. The latter is of particular interest when considering runtime performance since virtual member functions can usually not be made *inline* and are therefore not subject to code optimization within the calling function.[7] Modern microprocessor architectures can optimize at runtime, but, besides the difficulty of runtime predictions, these mechanisms are more likely to fail for virtual member functions. These effects are negligible for larger functions, but small functions will suffer a loss in runtime of one or two orders of magnitude. Significant examples are the access of point coordinates and arithmetic for low-dimensional geometric objects (see for example [85]).

The *generic programming paradigm* features what is known in C++ as *class templates* and *function templates*. Templates are incompletely specified components in which a few types are left open and represented by formal placeholders, the *template arguments*. The compiler generates a separate translation of the component with actual types replacing the formal placeholders wherever this template is used. This process is called *template instantiation*. The actual types for a function template are implicitly given by the types of the function arguments at instantiation time. An example is a swap function that exchanges the value of two variables of arbitrary types. The actual types for a class template are explicitly provided by the programmer. An example is a generic list class for arbitrary item types. The following definitions would enable us to use `list<int>`, with the actual type `int` given explicitly, for a list of integers and to swap two integer variables `x` and `y` with the expression `swap(x,y)`, where the actual type `int` is given implicitly.

```
template < class T >  class list {
    // placeholder T represents the item type symbolically
    void  push_back( const T& t);     // append t to the list
};

template < class T >  void swap( T& a, T& b) {
    T tmp = a; a = b; b = tmp;
}
```

The example of the swap function illustrates that a template usually requires certain properties of the template arguments, in this case that

[7]There are notable exceptions where the compiler can deduce for a virtual member function the actual member function that is called, which allows the compiler to optimize this call. The keyword `final` has been introduced in Java to support this intention. However, these techniques are not realized in C++ compilers so far and they cannot succeed in all cases, even though it is arguable that typical uses in CGAL can be optimized. However, distributing a software library in precompiled components will hinder their optimization, which must be done at link time.

variables of type `T` are assignable. An actual type used in the template instantiation must comply with these assumptions in order to obtain a correct template instantiation. We can distinguish between *syntactic requirements* (in our example the assignment operator is needed) and *semantic requirements* (the assignment operator should actually copy the value). If the syntactic requirements are not fulfilled, compilation simply fails. Semantic requirements cannot be checked at compile time. However, it might be useful to connect a specific semantic requirement to an artificial, newly introduced syntactic requirement, e.g. a tag similar to iterator tags in [99]. This technique allows decisions at compile time based on the actual type of these tags.

The set of requirements that is needed to obtain a correct instantiation of a member function of a class template is usually only a fraction of all requirements for the template arguments of this class template. If only a subset of the member functions is used in an instantiation of a class template, it would be sufficient for the actual types to fulfill only the requirements needed for this subset of member functions. This is possible in C++, since as long as a C++ compiler is not explicitly forced, the compiler is not allowed to instantiate member functions that are not used, and therefore possible compilation errors due to missing functionality of the actual types cannot occur [18]. This enables us to design class templates with optional functionality, which can be used if and only if the actual types used in the template instantiation fulfill the additional requirements.

A good and well known example illustrating generic programming is the Standard Template Library (STL) [99, 18, 72, 96]. Generality and flexibility have been achieved with the carefully chosen set of *concepts*, where a concept is a well defined set of requirements. In our `swap`-function example, the appropriate concept is named 'assignable' and includes the requirement of an assignment operator [96]. If an actual type fulfills the requirements of a concept, it is a *model* for this concept. Here, `int` is a model of the concept 'assignable'.

Algorithmic abstraction is a key goal in generic programming [73, 74]. One aspect is to reduce the interface to the data types used in the algorithm to a set of simple and general concepts. One of them is the *iterator* concept in STL which is an abstraction of pointers. Iterators serve two purposes: they refer to an item and they traverse over the sequence of items that are stored in a data structure, also known as *container* *class* in STL. Five different categories are defined for iterators: input, output, forward, bidirectional and random-access iterators, according to the different possibilities of accessing items in a container class. The usual C-pointer referring to a C-array is a model for a random-access iterator.

A *sequence* of items is specified by a *range* `[first,beyond)` of two iterators. This notion of a half-open interval denotes the sequence of all iterators obtained by starting with `first` and advancing `first` until `beyond` is reached, but it does not include `beyond`. A container class is supposed to provide a local type, which is a model of an iterator, and two member functions: `begin()` returns the start iterator of the sequence and `end()` returns the iterator referring to the 'past-the-end'-position of the sequence.

Generic algorithms are not written for a particular container class in STL, they use iterators instead. For example, a generic `contains` function can be written to work for any model of an input iterator. It returns `true` if and only if the `value` is contained in the values of the range `[first,beyond)`.

```
template < class InputIterator, class T >
bool contains( InputIterator first,
               InputIterator beyond, const T& value) {
    while ( ( first != beyond) && ( *first != value)) ++first;
    return ( first != beyond);
}
```

The advantages of the generic programming paradigm are strong type checking at compile time during the template instantiation, no need for extra storage nor additional indirections during function calls, and full support of inline member functions and code optimization at compile time [100]. One specific disadvantage of generic programming in C++ is the lack of a notation in C++ to declare the syntactical requirements for a template argument, i.e. the equivalent of the virtual base class in the object-oriented programming paradigm. The syntactical requirements are scattered throughout the implementation of the template. The concise collection of the requirements is left for the code documentation. In general, the flexibility is resolved at compile time, which gives the advantages mentioned above, but it can be seen as a disadvantage if runtime flexibility is needed. However, the generic data structures and algorithms can be parameterized with the base class used in the object-oriented programming to achieve runtime flexibility where needed.

5.3 Library Structure

CGAL is structured into three layers and a support library, which stands apart. The three layers are the core library with basic non-geometric functionality, the geometric kernel with basic geometric objects and operations, and the basic library with geometric algorithms and geometric data structures.

The three layers and the support library are further subdivided into smaller modular units, see Figure 5.1. The modular approach has several benefits. The library is easier to learn, because it is possible for a user to understand a small part without having any knowledge of other parts. For building the library, the modules are a good way of distributing implementation work among the project partners. Testing and maintainance of the library is easier when there are only few dependencies between units [61].

The geometric kernel contains simple geometric objects of constant size. Examples are points, lines, planes, segments, triangles, tetrahedra, circles, spheres and more. There are geometric predicates on those objects, operations to compute intersections of and distances between objects, and affine transformations. The geometric kernel is split up into three parts, which deal with two-dimensional objects, three-dimensional objects, and general-dimensional objects. Geometry in two and three dimensions is well studied and has lots of applications, which is the reason for the special status of the corresponding parts. For all dimensions there are Cartesian and homogeneous representations available for the coordinates.

To solve robustness problems, CGAL advocates the use of exact arithmetic instead of floating point arithmetic. An arithmetic is associated with a number type in CGAL and the classes in the geometric kernel are parameterized by number types. CGAL provides own number types [17, 16] and supports number types from other sources, e.g. from LEDA or the GNU Multiple Precision Library [51]. Since the arithmetic

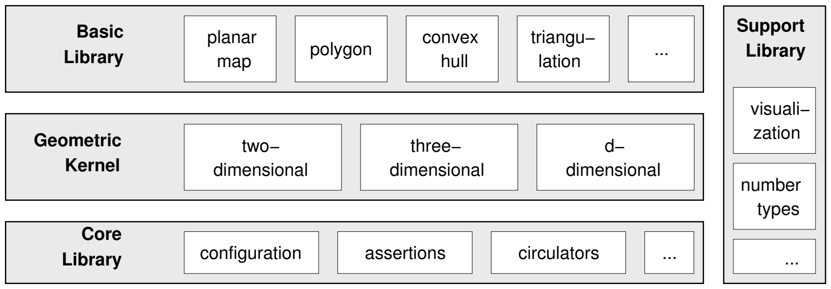


Figure 5.1: *The structure of* CGAL

operations needed in CGAL are quite basic, every library supplying number types can be easily adapted to work with CGAL.

The basic library contains more complex geometric objects and data structures: polygons, planar maps, polyhedra and so on. It also contains algorithms, such as computing the convex hull of a set of points, the union of two polygons, smallest enclosing ellipse and so on. Figure 5.1 indicates the major parts in the basic library. These parts are mostly independent from each other and even independent from the kernel. This independence has been achieved with geometric traits classes as described in Section 5.5.3 below.

The core library offers basic non-geometric functionality needed by the geometric kernel or the basic library. Very important is the support for different C++ compilers which all have their own limitations. The core library also contains circulators and random number generators as well as code for checking assertions, preconditions, and postconditions.

In contrast to the core library, the support library provides functionality that the rest of the library does not depend on. Visualization and external file formats are important aspects of the support library. The list of supported formats contains VRML and PostScript as well as the GeomView program and LEDA windows for 2D and 3D visualization. The adaptation of number types from other libraries is contained in the support library, too. The separation from the rest of the library makes the functionality of the support library orthogonal and open for future extensions.

5.4 Geometric Kernel

The geometric kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, triangle, iso-oriented rectangle and tetrahedron. Each type provides a set of member functions, for example access to the defining objects, the bounding box of the object if existing, and affine transformation. Global functions are available for the detection and computation of intersections as well as for distance computations.

The current geometric kernel provides two families of geometric objects: one family is based on the representation of points using Cartesian coordinates, the other family is based on the representation of points using homogeneous coordinates. The homogeneous representation extends the representation with Cartesian coordinates by an additional coordinate, namely a common denominator. More formally, a point in d dimensional space with homogeneous coordinates $(x_0, x_1, \ldots, x_{d-1}, x_d)$, $x_d \neq 0$, has Cartesian coordinates $(x_0/x_d, x_1/x_d, \ldots, x_{d-1}/x_d)$. This avoids divisions and reduces many computations in geometric algorithms to calculations over the integers. The homogeneous representation is used for affine geometry in CGAL, and not projective geometry where the homogeneous representation is usually known from. Both families are parameterized by the number type used to represent the Cartesian or homogeneous coordinates. The type `CGAL::Cartesian<double>`[8] specifies the Cartesian representation with coordinates of type `double`, and the type `CGAL::Homogeneous<int>` specifies the homogeneous representation with coordinates of type `int`. These representation types are used as template argument in all geometric kernel types, like a two-dimensional point declared as

```
template <class R> CGAL::Point_2;
```

with a template parameter `R` for the representation class. Usually, typedefs are used to introduce conveniently short names for the types. Here is an example given for the point type with the homogeneous representation and coordinates of type `int`:

[8]CGAL provides its own C++ namespace `CGAL`. All classes and functions of the library are defined in this namespace and can be accessed via the scope prefix `CGAL::`.

```
typedef CGAL::Point_2< Homogeneous<int> >  Point_2;
```

The class templates parameterized with `CGAL::Cartesian` or `CGAL::Homogeneous` provide the user with a common interface to the underlying representation, which can be used in higher-level implementations independently of the actual coordinate representation. The list of requirements on the template parameter defines the concept of a *representation class* for the geometric kernel. Details for realizing this parameterization can be found in [30, 31, 53].

CGAL provides clean mathematical concepts to the user without sacrificing efficiency. For example, CGAL strictly distinguishes points and (mathematical) vectors, i.e. it distinguishes affine geometry from the underlying linear algebra. Points and vectors are not the same as it is discussed in [50] with regard to illicit computations resulting from identification of points and vectors in geometric computations. In particular, points and vectors behave differently under affine transformations [104]. We do not even provide automatic conversion between points and vectors but use the geometric concept of an origin instead. The symbolic constant `CGAL::ORIGIN` represents a point and can be used to compute the locus vector as the difference between a point and the origin. Function overloading is used to implement this operation internally as a simple conversion without any overhead. Note that we do not provide the geometrically invalid addition of two points, since this might lead to ambiguous expressions: assuming three points p, q, and r and an affine transformation A, one can write in CGAL the perfectly legal expression $A(p + (q - r))$. The slightly different expression $A((p + q) - r)$ contains the illegal addition of two points. Thinking in terms of coordinates, one might expect the same result with the addition allowed as in the previous, legal expression. But this is not necessarily intended, since the expression within the affine transformation would probably evaluate to a vector, not a point as in the previous expression. Vectors and points behave differently under affine transformations. To avoid these ambiguities, the automatic conversion between points and vectors is not provided.

Class hierarchies are used rarely in CGAL. An example are affine transformations which maintain distinct internal representations specialized on restricted transformations. The internal representations differ considerably in their space requirements and the efficiency of their member functions. For all but the most general representation we gain performance in terms of space and time. And for the most general representation, the performance penalty caused by the virtual functions is negligible, because the member functions are computationally expensive for this representation. Alternatively we could have used this most general representation for affine transformations only. But the use of a hierarchy is justified, since the specialized representations, namely translation, rotation and scaling, arise frequently in geometric computing.

Another design decision was to make the (constant-size) geometric objects in the kernel non-modifiable. For example, there are no member functions to set the Cartesian coordinates of a point. Points are viewed as atomic units (see also [25]) and no assumption is made on how these objects are represented. In particular, there is no assumption that points are represented with Cartesian coordinates. They might use polar coordinates or homogeneous coordinates instead. Then, member functions to set the Cartesian coordinates are expensive. Nevertheless, in current CGAL the types based on the Cartesian representation as well as the types based on the homogeneous representation have both member functions returning Cartesian coordinates and member functions returning homogeneous coordinates. These access functions are provided to make implementing own predicates and operations more convenient.

Like other libraries [68, 13, 57] we use reference counting for the kernel objects. Objects point to a shared representation and each representation counts the number of objects pointing to it. Copying objects increments the counter of the shared representation, deleting an object decrements the counter of its representation. If the counter reaches zero by the decrement, the representation itself is deleted (see [70, Item 29] for further information). The implementation of reference counting is simplified by the non-modifiability of the kernel objects. However, the use of reference counting was not the reason for choosing non-modifiability. Using 'copy-on-write', i.e. a new representation is created for an object whenever its value is changed by a modifying operation, reference counting of modifiable objects is possible and only slightly more involved. A comparison with a prototype of a geometric kernel without reference counting can be found in [85]. The test applications are two-dimensional convex hull algorithms. Reference counting costs about 15% to 30% runtime for the types `double` and `float`, but it gains 2% to 11% runtime for the type `leda_real`. Meanwhile, CGAL provides two additional representation classes without reference counting. They are named `CGAL::Simple_cartesian` and `CGAL::Simple_homogeneous`.

Further details of the geometric kernel can be found in [31], for example the polymorphic behaviour of the return type of the intersection functions. Recent developments towards an even more adaptable and extensible geometric kernel are described in [53].

5.5 Basic Library

The basic library contains more complex geometric objects and data structures, such as polygons, polyhedrons, triangulations (including Delaunay, constrained, and regular triangulations), planar maps, range and segment trees, and kd-trees. It also contains geometric algorithms, such as convex hull, smallest enclosing circle, ellipse, sphere, and annulus, polytope distance, boolean operations on polygons, and map overlay.

5.5.1 Generic Data Structures

Triangulations are an example of a container-like data structure in the basic library. The interface contains, among others, member functions to access the vertices of the triangulation. For example all vertices or the vertices on the convex hull of the triangulation.

```
class Triangulation {
  public:
    Vertex_iterator       vertices_begin();
    Vertex_iterator       vertices_end();

    Convex_hull_iterator  convex_hull_begin();
    Convex_hull_iterator  convex_hull_end();

    // ...
};
```

The whole functionality for accessing vertices is factored out in separate classes, which are models for the iterator concept.[9]

As in the previous example, geometric data structures in the basic library often contain more than one sequence of interest, e.g. triangulations contain vertices, edges, and faces. The names of the member functions that return iterator ranges are prefixed with the name of the sequence, e.g. `vertices_begin()`, `edges_end()`. These names are the canonical extension of the corresponding names `begin()` and `end()` in STL. The iterator based interfaces together with the extended naming scheme assimilate the design of the container-like geometric data structures in the basic library with the C++ Standard.

5.5.2 Generic Algorithms

An example of a geometric algorithm is the convex hull computation. The algorithm takes a set of points and outputs the sequence of points on the convex hull. The function declaration looks like this:

```
template < class InputIterator, class OutputIterator >
OutputIterator  convex_hull( InputIterator  first,
                             InputIterator  beyond,
                             OutputIterator result);
```

Here, the input is read from the iterator range `[first,beyond)` and the output is written to the output iterator `result`. Let the return-value be `result_beyond`, then the iterator range `[result,result_beyond)` contains the sequence of points on the convex hull. This design decouples the algorithm from the container and gives the user the flexibility to use any container, e.g. from STL, from other libraries or own implementations (provided they are STL compliant). It is even possible to use no container at all, for example a sequence of points read from the standard input:

[9]The actual implementation in CGAL differs slightly, i.e. the vertices of the convex hull of the triangulation are accessed with a more efficient circulator [58], since the internal representation of this convex hull is cyclic.

```
convex_hull( istream_iterator<Point>( cin),
             istream_iterator<Point>(),
             ostream_iterator<Point>( cout, "\n"));
```

Points are taken from the standard input and the resulting points on the convex hull are written to the standard output. Here so-called stream iterators [72] from STL are used. This example again demonstrates the flexibility gained from the STL-compliance of the geometric algorithms in the basic library.

5.5.3 Traits Classes

An ideal theoretical paper on a geometric algorithm first declares geometric primitives and thereafter expresses the algorithm in terms of these primitives. Implementing an algorithm or data structure, we collect all necessary types and primitive operations in a single class, called *traits class*, which encapsulates details, such as the geometric representation. Collecting types in a single class is a template technique that is already intensively used in [10]. It is sometimes called *'nested typedefs for name commonality'-idiom*. The approach gains much additional value by the *traits technique* as used in the C++ Standard Library [75], where additional information is associated with already existing types or built-in types.

An example is an iterator which refers to a particular value type. Algorithms parameterized with iterators might need the value type directly. This can be easily encoded as a local type for all iterators that are implemented as classes:

```
struct iterator_to_int {
    typedef  int  value_type;
    // ...
};
```

Since a C-pointer is a valid iterator, this approach is not sufficient. The solution chosen for STL are iterator traits [75], i.e. class templates parameterized with an iterator:

```
template < class Iterator >
struct iterator_traits {
    typedef  typename Iterator::value_type  value_type;
    // ...
};
```

The value type of the iterator example class above can be expressed as `iterator_traits< iterator_to_int >::value_type`. For C-pointers a specialized version of iterator traits exists, i.e. a class template parameterized with a C-pointer.

```
template < class T >
struct iterator_traits<T*> {
    typedef  T  value_type;
    // ...
};
```

Now the value type of a C-pointer, e.g. to `int`, can be expressed as `iterator_traits< int* >::value_type`. This technique of providing an additional, more specific definition for a class template is known as *partial specialization*.

Our approach using traits classes in the basic library does not attach information to built-in types, but to our data structures and algorithms. We use them as a modularization technique that allows a single implementation to be interfaced to different geometric representations and primitive operations. Our traits class is therefore a single template argument for algorithms and data structures in the basic library, for example triangulations:

```
template < class Traits >
class CGAL::Triangulation_2 {
    // ...
};
```

Note that each primitive could as well be provided as a template parameter for itself, but using traits classes simplifies the interface. All primitives are captured in a single argument, which makes it easier to apply already prepared implementations of traits classes.

Traits classes must provide the geometric primitives required by the geometric data structure or algorithm. Default implementations are provided for the geometric kernel of CGAL. They are class templates parameterized with a kernel representation class, for example `CGAL::Triangulation_euclidean_traits_2< CGAL::Cartesian<double> >`. A single traits class for triangulations is sufficient for all representations and number types possible with the kernel. Further traits classes are available in CGAL, for example for using the basic library with the geometric part of LEDA.

For algorithms implemented as functions, a default traits class is chosen automatically if there is none given explicitly in the function call. Thus the user can just ignore the traits class mechanism as in the following example:

```
typedef  CGAL::Cartesian<double> R;
typedef  CGAL::Point_2<R>        Point;
typedef  CGAL::Polygon_2<R>      Polygon;

Polygon  hull;
CGAL::convex_hull_points_2( istream_iterator<Point>( cin),
                            istream_iterator<Point>( ),
                            back_inserter( hull));
```

In the call to the convex hull algorithm no traits class is visible to the user. A default traits class is chosen automatically in the definition of the algorithm:

```
template < class InputIterator, class OutputIterator >
OutputIterator
CGAL::convex_hull_points_2( InputIterator  first,
                            InputIterator  beyond,
                            OutputIterator result) {
    typedef  typename iterator_traits<InputIterator>::value_type
                                             P;
    typedef  typename P::R                   R;
    typedef  CGAL::Convex_hull_traits_2<R>  T;
    return CGAL::convex_hull_point_2( first,beyond,result,T());
}
```

The value type of the iterator `InputIterator` is the point type used in the input. It is determined with the iterator traits described previously. Since the default traits class is supposed to use the geometric kernel of CGAL, we know that the point type must be a CGAL point type and it ‘knows’ its representation type by means of a local type named `R`. Finally, another version of `CGAL::convex_hull_points_2` is called with four arguments. The additional fourth argument is set to `CGAL::Convex_hull_traits_2<R>()`, the default traits class for the convex hull algorithms. This second version of the function template is defined as follows:

```
template < class InputIterator,
           class OutputIterator, class Traits >
OutputIterator
CGAL::convex_hull_points_2( InputIterator  first,
                            InputIterator  beyond,
                            OutputIterator result,
                            const Traits&  traits) {
    // compute the convex hull
    // using only primitives from the traits class
}
```

5.6 Conclusion

We followed mainly the generic programming paradigm to achieve flexibility and efficiency in CGAL. The compliance with STL is important in order to re-use its generic algorithms and container classes, and to unify the look-and-feel of the design with the C++ standard. CGAL is therefore easy to learn and easy to use for those who are familiar with STL.

The abstract concepts used in STL are so powerful that only a few additions and refinements are needed in CGAL. One refinement is the concept of *handles*. Combinatorial data structures might not necessarily possess a natural order on their items. Here, we restrict the concept of iterators to the concept of handles, which is the item denoting part of the iterator concept, and which ignores the traversal capabilities. Any model of an iterator is a model for a handle. A handle is also known as *trivial iterator*. Another refinement is the concept of *circulators* [58], a kind of iterators with slightly adapted requirements to better suit the needs of circular sequences. These occur naturally in several combinatorial data structures, such as the sequence of edges around a vertex in a triangulation.

The geometric traits classes offer great flexibility and modularity. There is always a predefined traits class that uses types and operations of the kernel. Where possible, this traits class is chosen by default, so the user can totally ignore the existence of this mechanism.

In a few places we also used the object-oriented programming paradigm. Examples are affine transformations and the polymorphic return type of the intersection functions.

Chapter 6

A Quadratic Programming Engine

This chapter presents the implementation of our algorithm for solving linear and quadratic programs. We describe our design goals and the design decisions made to achive these goals. The realization of the solver and the basis inverse is discussed as well as the implementation of different pricing strategies. We concentrate on the distinguishing features of the code, while further details of the implementation and the whole code can be found in [89, 90, 91].

6.1 Design Goals

The quadratic programming engine (QPE) is carefully designed and implemented in C++. We follow the generic programming paradigm, as it is realized in the Stl. Our design goals are closely related to those of Cgal [31]. Among them, the most important ones are *flexibility* for the user and *efficiency* of the code. For the QPE this means in particular:

1. Our method described in Chapter 2 solves quadratic programs and also linear programs as a special case. If the user knows in

advance (i.e. at compile time), that the problem to solve has a linear objective function, then (almost) no overhead should occur compared to a stand-alone implementation of a solver for linear programs.

2. Different geometric optimization problems come with different representations. We want to allow the user to choose his favorite format without the necessity of converting and copying his problem to a specific representation. It should be also possible to represent the possibly very large and dense objective matrix D implicitly.

3. The correctness and efficiency of the algorithm heavily depends on the underlying arithmetic. The user should be able to choose a suitable (i.e. correctness guaranteeing) number type for the internal computations on the one hand, while specifying the optimization problem with one or several possibly different number types on the other hand.

4. Since different optimization problems perform differently with different pricing strategies in general, there is no superior strategy for all cases. The implementation should provide an easy-to-use interface for choosing one of several predefined pricing strategies. Furthermore, a framework for implementing own pricing strategies should be available to the user.

6.2 Solver

The implementation of the QPE is divided in three parts: the solver [89], the basis inverse [90], and the pricing strategies [91].

The solver is realized as a class template with a representation class as template parameter.

```
template < class QPErep >  class QPE_solver;
```

The template parameter `QPErep` is a traits class in the sense of Section 5.5.3. In the sequel, we describe the requirements for `QPErep`.

6.2.1 Access to Original Problem

The QPE solves optimization problems with n variables and m constraints of the following form:

$$\begin{array}{lll} \text{(QP)} & \text{minimize} & c^T x + x^T D\, x \\ & \text{subject to} & A\, x \lesseqqgtr b \\ & & x \geq 0\,. \end{array} \tag{6.1}$$

Here, A is an $m \times n$-matrix, b an m-vector, c an n-vector, and D a positive semi-definite $n \times n$-matrix. The symbol '$\lesseqqgtr$' indicates that any of the m order relations it stands for can independently be '$\leq$', '$=$', or '$\geq$'. Compared to the general form given in (1.2) on page 16, the explicit bounds on the variables are missing in (6.1). The current implementation of the QPE assumes nonnegativity constraints on the variables.

Different geometric optimization problems come with different representations. We allow the user to choose his favorite format by only asking for *iterators* to access the problem. This also avoids copying overhead and, more important, gives the possibility of representing the $n \times n$-matrix D implicitly. This feature is already used in the solutions for the polytope distance problem and the smallest enclosing ball problem, see next chapter.

We need five iterators for accessing A, b, c, D, and r. The latter represents the m order relations in (6.1). The iterator types are given in the representation class:

```
struct QPErep {
    typedef  ...  A_iterator;
    typedef  ...  B_iterator;
    typedef  ...  C_iterator;
    typedef  ...  D_iterator;

    enum Row_type = { EQUAL, LESS_EQUAL, GREATER_EQUAL };
    typedef  ...  R_iterator;

    // ...
};
```

The iterators for the actual problem are passed as parameters to the solver's `set` method, which is declared as follows:

```
template < class QPErep >
class QPE_solver {
  public:
    void  set( int n, int m,
               QPErep::A_iterator a_it, QPErep::B_iterator b_it,
               QPErep::C_iterator c_it, QPErep::D_iterator d_it,
               QPErep::R_iterator r_it);
    // ...
};
```

All five iterators have to allow random-access, e.g. `b_it[j]` is the j-th entry of b and `c_it[i]` is the i-th entry of c. Since A is accessed column-wise, `a_it[j]` is an iterator referring to the first entry in the j-th column of A, while `d_it[i]` is an iterator referring to the first entry in the i-th row of D.

6.2.2 Tags for Special Properties

Some optimization problems have special properties, like a linear objective function or a symmetric objective matrix. If such a property is known in advance, i.e. at compile time, the implementation of the QPE can be tailored for this special case, thus making it more efficient as in the general case. We support tags for the following three properties:

- The objective function is linear, i.e. $D = 0$.
- The objective matrix D is symmetric.
- The problem has no inequality constraints.

There are two types for defining tags, namely `Tag_true` and `Tag_false`. If a property is present, the corresponding tag is defined to be of type `Tag_true`, otherwise of type `Tag_false`.

```
struct QPErep {
    // ...
    typedef  ...  Is_linear;
    typedef  ...  Is_symmetric;
    typedef  ...  Has_no_inequalities;
    // ...
};
```

In case the objective function is linear, the extension of the simplex method to QP is switched off, resulting in almost no overhead compared to a stand-alone implementation for LP. If the problem has only equality constraints, the whole handling of slack variables is disabled. A symmetric objective matrix halfs the number of accesses to entries of D. All this is done at compile time, when the compiler only generates the code needed for the specific type of problem. We illustrate this technique by the following example.

The current solution of the QPE consists of the values of the original variables and the values of the slack variables. The latter are only present, if the given problem has inequality constraints.

```
template < class QPErep >
class QPE_solver {
    void  compute_solution( )
        {
            // ...
            // compute values of slack variables, if any
            compute_slack_variables(
                typename QPErep::Has_no_inequalities());
            // ...
        }
};
```

The function `compute_slack_variables` is called with an instance of the type `QPErep::Has_no_inequalities`. At this point, the compiler decides based on the actual type of the flag, which one of the two following implementations of `compute_slack_variables` has to be called.

```
template < class QPErep >
class QPE_solver {
    void  compute_slack_variables( Tag_true)
        {
            // nop
        }

    void  compute_slack_variables( Tag_false)
        {
            // compute values of slack variables...
        }
};
```

Thus, the code for computing the values of the slack variables is only generated, if the given problem has inequality constraints.

6.2.3 Exact Arithmetic

The correctness and the efficiency of the algorithm heavily depend on the underlying arithmetic. We rely on an exact number type for the internal computations, which the user can choose independently from the representation of the optimization problem. There only have to be implicit conversions from the entry types of A, b, c, and D to the exact number type.

```
struct QPErep {
    typedef  ...  ET;
    // ...
};
```

The arithmetic requirements for `ET` are quite basic. It has to support addition, subtraction, and multiplication. A division operation is only required for those cases where the remainder is zero. Fulfilling these requirements, `ET` is a model for the concept *RingNumberType* of CGAL.

In Section 6.4.2 below, we describe how to combine the exact arithmetic over `ET` with a very efficient floating point filter to speed up the pricing step considerably.

6.2.4 Symbolic Perturbation

As introduced in Section 2.5 of Part I, we use symbolic perturbation to cope with degenerate quadratic programs. The right-hand-side vector b is perturbed by the vector $\varepsilon := (\epsilon, \epsilon^2, \ldots, \epsilon^m)^T$ for some $0 < \epsilon < 1$, resulting in vector-valued polynomials in ϵ as solutions of the quadratic program.

During the ratio test, we test which basic variable becomes zero first, when we increase the entering variable. This is done by finding the smallest positive quotient $t_i = x_i / q_{x_i}$ for all $i \in B$ (cf. Section 2.3.2).

Now, the solution x_B^* is a polynomial in ϵ, and so are the quotients:

$$\begin{aligned} t_i(\epsilon) &= \frac{x_i(\epsilon)}{q_{x_i}} \\ &= \frac{x_i^0}{q_{x_i}} + \frac{x_i^1}{q_{x_i}}\,\epsilon + \frac{x_i^2}{q_{x_i}}\,\epsilon^2 + \cdots + \frac{x_i^m}{q_{x_i}}\,\epsilon^m\,, \qquad i \in B\,. \end{aligned} \tag{6.2}$$

The second equation is derived using (2.29) on page 35. Since $0 < \epsilon < 1$, we find the smaller of two $t_i(\epsilon)$ by comparing their first quotients as defined in (6.2). Only if these have the same value, we compare their second quotients, and so on. In other words, we compare two vectors $v_i := (x_i^0/q_{x_i}, x_i^1/q_{x_i}, \ldots, x_i^m/q_{x_i})$ lexicographically. Because the first entry of v_i determines the sign of $t_i(\epsilon)$, most comparisons are decided after comparing the first entries. Usually, there are very few cases that need the second or subsequent entries to decide the comparison.

Summarizing, the symbolic perturbation scheme does not introduce any computational overhead in the current ratio test, if the non-perturbed quadratic program has a unique minimal $t_i > 0$. Only if the first entries of the corresponding vectors v_i have the same value, some additional computations are needed. To this end, note that $x_i^1, \ldots, x_i^m$ are just entries of the corresponding row of the basis inverse. This can be seen by replacing b with $b+\varepsilon$ in (2.7) on page 25.

The same technique is used to determine t_j with $\mu_j(t_j) = 0$ and to compare it with the smallest positive t_i.

6.3 Basis Inverse

Given a basis B, the *basis matrix* is defined as

$$M_B := \left(\begin{array}{c|c} 0 & A_{E\dot\cup S_N, B_O} \\ \hline A^T_{E\dot\cup S_N, B_O} & 2D_{B_O,B_O} \end{array}\right), \tag{6.3}$$

and M_B^{-1} is called the *basis inverse*. From our generalized simplex method used in the QPE, we have the following additional structure and requirements.

- We do not handle just one basis matrix M_B, but a sequence of matrices. Here, successive ones only differ slightly, i.e. one row and/or one column is either appended, removed, or replaced. We want to exploit this coherence.
- If the matrix M_B and the vectors $b_{E \dot\cup S_N}$ and $-c_{B_O}$ contain integral entries, then the entries of the solution vectors $x^*_{B_O}$ and λ are rational, and we would like to obtain *exact* rational representations of them. This means that explicit divisions have to be avoided during the solution process (unless they do not leave a remainder).
- If D is the zero-matrix, i.e. the problem to solve is a linear program (LP), we want to have only a small overhead in time and space compared to a stand-alone implementation of the simplex method for solving LPs.

We will refer to the *QP case* or the *LP case* in the sequel, if the problem to solve is a quadratic or linear program, respectively.

6.3.1 Rational Representation

The objective function does not change, if we vary a pair of elements of D symmetric to the main diagonal, as long as its sum remains the same. Thus, we may assume w.l.o.g. that D is symmetric, and so are M_B and M_B^{-1}. Consequently, we will only store the entries on and below the main diagonal.

In the LP case, every basis has size m. The basis matrix assumes the form

$$M_B = \left(\begin{array}{c|c} 0 & A_{E \dot\cup S_N, B_O} \\ \hline A^T_{E \dot\cup S_N, B_O} & 0 \end{array}\right),$$

with $|E| + |S_N| \;=\; |E| + |S| - |S_B| \;=\; m - |B_S| \;=\; |B| - |B_S| \;=\; |B_O|$, i.e. $A_{E \dot\cup S_N, B_O}$ is quadratic. The resulting basis inverse is

$$M_B^{-1} = \left(\begin{array}{c|c} 0 & (A_{E \dot\cup S_N, B_O})^{-1^T} \\ \hline (A_{E \dot\cup S_N, B_O})^{-1} & 0 \end{array}\right) \tag{6.4}$$

and it suffices to store $(A_{E \dot\cup S_N, B_O})^{-1}$, avoiding any space overhead compared to a stand-alone implementation of the simplex method for solving linear programs.

To address the second requirement, we consider Cramer's well-known rule for the inverse of a matrix M in terms of the M^{ji},

$$M^{-1}_{i,j} = \frac{(-1)^{i+j} \det(M^{ji})}{\det(M)}.$$

It follows that the entries of M^{-1} can be written as rationals with a common denominator $\det(M)$, if M contains integral entries. Hence, M^{-1} can also be stored with integral entries, keeping the denominator separately. For practical reasons, we prefer the absolute value $|\det(M)|$ to the signed value $\det(M)$ as the common denominator. Then, for example, the numerators appearing in the matrix-vector products already have the same sign as the true rational values. We store M^{-1} in the form

$$M^{-1}_{i,j} = \frac{\operatorname{sgn}(\det(M))\,(-1)^{i+j} \det(M^{ji})}{|\det(M)|}. \tag{6.5}$$

The integral part of the basis inverse is defined by

$$\hat{M}^{-1} := d\,M^{-1},$$

where $d := |\det(M)|$. In the sequel, any value x and the corresponding $\hat{x}$ satisfy $x = \hat{x}/d$.

6.3.2 Updates

The QPE produces a sequence of bases, where successive ones only differ by one variable. Either a nonbasic variable enters the basis (QP case), a basic variable leaves the basis (QP case), or a nonbasic variable replaces a basic variable in the basis (LP case). Since we distinguish between original and slack variables, we have the following eight different types of updating the basis inverse:

U1 An *original* variable *enters* the basis, i.e. B_O is increased by one element.

U2 An *original* variable *leaves* the basis, i.e. B_O is decreased by one element.

U3 A *slack* variable *enters* the basis, i.e. S_N is decreased by one element.

U4 A *slack* variable *leaves* the basis, i.e. S_N is increased by one element.

U5 An *original* variable *replaces* an *original* variable in the basis, i.e. one element in B_O is replaced.

U6 A *slack* variable *replaces* a *slack* variable in the basis, i.e. one element in S_N is replaced.

U7 An *original* variable *replaces* a *slack* variable in the basis, i.e. B_O and S_N each increase by one element.

U8 A *slack* variable *replaces* an *original* variable in the basis, i.e. B_O and S_N each decrease by one element.

Note, the first four update types belong to the QP case, while the latter four belong to the LP case.

An update of type U1, U4, or U7 enlarges the matrix M by one row and one column. We may assume w.l.o.g. that the row and the column are appended to the bottom and to the right of M, respectively, to simplify the presentation. Let (u^T, w) be the row and $(v^T, w)^T$ the column to add, with w being the entry they have in common. Then the new matrix is

$$M_> = \left(\begin{array}{c|c} M & v \\ \hline u^T & w \end{array}\right).$$

It is easy to verify that

$$M_> = \left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline & x^T & & 1 \end{array}\right) \left(\begin{array}{c|c} & 0 \\ M & \vdots \\ & 0 \\ \hline 0 \dots 0 & z \end{array}\right) \left(\begin{array}{ccc|c} 1 & & & \\ & \ddots & & y \\ & & 1 & \\ \hline & 0 \dots 0 & & 1 \end{array}\right) \qquad (6.6)$$

holds, where

$$x^T := u^T M^{-1}, \quad y := M^{-1} v, \quad z := w - u^T M^{-1} v.$$

This implies for the inverse of the new matrix

$$M_>^{-1} = \left(\begin{array}{ccc|c} 1 & & & \\ & \ddots & & -y \\ & & 1 & \\ \hline 0 & \dots & 0 & 1 \end{array}\right) \left(\begin{array}{c|c} & 0 \\ M^{-1} & \vdots \\ & 0 \\ \hline 0 \dots 0 & 1/z \end{array}\right) \left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline & -x^T & & 1 \end{array}\right)$$

$$= \frac{1}{z} \left(\begin{array}{c|c} z\,M^{-1} + y\,x^T & -y \\ \hline -x^T & 1 \end{array}\right). \tag{6.7}$$

Since we represent M^{-1} in rational form, also x^T, y, and z are represented as (vectors of) rationals with common denominator d, i.e.

$$x^T =: \frac{\hat{x}^T}{d}, \quad y =: \frac{\hat{y}}{d}, \quad z =: \frac{\hat{z}}{d}\,. \tag{6.8}$$

From (6.6) we get

$$\det(M_>) = \det(M)\,z = \operatorname{sgn}(\det(M))\,d\,z = \operatorname{sgn}(\det(M))\,\hat{z},$$

hence $|\hat{z}|$ is the denominator of the rational representation of the inverse of the new matrix. Substituting (6.8) in (6.7) gives

$$M_>^{-1} = \frac{d}{\hat{z}} \left(\begin{array}{c|c} (\hat{z}\,\hat{M}^{-1} + \hat{y}\,\hat{x}^T)/d^2 & -\hat{y}/d \\ \hline -\hat{x}^T/d & 1 \end{array}\right)$$

$$= \frac{\operatorname{sgn}(\hat{z})}{|\hat{z}|} \left(\begin{array}{c|c} (\hat{z}\,\hat{M}^{-1} + \hat{y}\,\hat{x}^T)/d & -\hat{y} \\ \hline -\hat{x}^T & d \end{array}\right) \tag{6.9}$$

$$= \frac{\hat{M}_>^{-1}}{|\hat{z}|}\,.$$

Note, the division by d is without remainder, following from Cramer's rule.

The complementary operation, i.e. the removal of one row and one column, is done by updates of type U2, U3, and U8. We assume w.l.o.g. that the last row and the last column of M should be removed, to simplify the presentation. This can always be achived by swapping the row and the column to be removed with the last row and the last column, respectively. Now we look at the last row and the last column as if they were just appended to M as described above. Equation (6.9) holds by the uniqueness of the inverse, and we get

$$\operatorname{sgn}(\hat{z}) \left(\begin{array}{c|c} \cdots & -\hat{y} \\ \hline -\hat{x}^T & d \end{array} \right) = \hat{M}^{-1} \tag{6.10}$$

with common denominator $|\hat{z}|$. The integral part of the new inverse $M_<^{-1}$ is obtained by

$$\left(\begin{array}{c|c} \hat{M}_<^{-1} & \vdots \\ \hline \cdots & \cdot \end{array} \right) = \frac{1}{\hat{z}} \Big(\operatorname{sgn}(\hat{z})\, d\, \hat{M}^{-1} - \left(\begin{array}{c|c} \hat{y}\, \hat{x}^T & \vdots \\ \hline \cdots & \cdot \end{array} \right) \Big), \tag{6.11}$$

ignoring the last row and the last column of $\hat{M}^{-1}$ in the calculation above. Again, the division by $\hat{z}$ is without remainder, and the new denominator is $d = |\det(M_<)|$. The sign of $\hat{z}$ can be computed easily using the fact that d has to be positive by definition.

The two remaining update types either replace one row (U5) or one column (U6) of M. We can assume w.l.o.g. that the last row or column is replaced. Let u^T be the new row and v the new column, respectively.

We define $x^T := u^T M^{-1}$ and $y := M^{-1} v$ and get new matrices M_r (row replaced) and M_c (column replaced) as

$$M_r = \left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline x_1 & \dots & x_{k-1} & x_k \end{array} \right) M \tag{6.12}$$

and

$$M_c = M \left(\begin{array}{ccc|c} 1 & & & y_1 \\ & \ddots & & \vdots \\ & & 1 & y_{k-1} \\ \hline 0 & \dots & 0 & y_k \end{array} \right). \tag{6.13}$$

The inverses of the new matrices are

$$M_r^{-1} = M^{-1}\,\frac{1}{x_k}\left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline -x_1 & \dots & -x_{k-1} & 1 \end{array}\right)$$

and

$$M_c^{-1} = \frac{1}{y_k}\left(\begin{array}{ccc|c} 1 & & & -y_1 \\ & \ddots & & \vdots \\ & & 1 & -y_{k-1} \\ \hline 0 & \dots & 0 & 1 \end{array}\right) M^{-1}.$$

By substituting $M^{-1} = \hat{M}^{-1}/d$, $x^T = \hat{x}^T/d$, and $y = \hat{y}/d$, we obtain the integral parts as

$$\begin{aligned} M_r^{-1} &= \frac{\hat{M}^{-1}}{d}\,\frac{d}{\hat{x}_k}\left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \\ \hline -\hat{x}_1/d & \dots & -\hat{x}_{k-1}/d & 1 \end{array}\right) \\ &= \frac{\operatorname{sgn}(\hat{x}_k)}{|\hat{x}_k|}\Bigl(\hat{M}^{-1}\left(\begin{array}{ccc|c} d & & & 0 \\ & \ddots & & \vdots \\ & & d & 0 \\ \hline -\hat{x}_1 & \dots & -\hat{x}_{k-1} & d \end{array}\right)\Bigr)/d = \frac{\hat{M}_r^{-1}}{|\hat{x}_k|} \end{aligned} \tag{6.14}$$

and

$$\begin{aligned} M_c^{-1} &= \frac{d}{\hat{y}_k}\left(\begin{array}{ccc|c} 1 & & & -\hat{y}_1/d \\ & \ddots & & \vdots \\ & & 1 & -\hat{y}_{k-1}/d \\ \hline 0 & \dots & 0 & 1 \end{array}\right)\frac{\hat{M}^{-1}}{d} \\ &= \frac{\operatorname{sgn}(\hat{y}_k)}{|\hat{y}_k|}\Bigl(\left(\begin{array}{ccc|c} d & & & -\hat{y}_1 \\ & \ddots & & \vdots \\ & & d & -\hat{y}_{k-1} \\ \hline 0 & \dots & 0 & d \end{array}\right)\hat{M}^{-1}\Bigr)/d = \frac{\hat{M}_c^{-1}}{|\hat{y}_k|}. \end{aligned} \tag{6.15}$$

The new denominators are $|\hat{x}_k|$, since $\det(M_r) = x_k \det(M) = \hat{x}_k$ holds by (6.12), and $|\hat{y}_k|$, since $\det(M_c) = y_k \det(M) = \hat{y}_k$ holds by (6.13).

Updates of the first four types can be performed in time $O((m + |B_O|)^2)$ and updates of the latter four types in time $O(m^2)$. This is substantially cheaper than computing the basis inverse from scratch.

The technique of updating the basis inverse when a column in the basis matrix is replaced has been proposed before by Edmonds and termed 'Q-pivoting' [28]. It is also used, for example, by Gärtner in his exact implementation of the simplex method [39] and by Avis in his vertex enumeration algorithm [5].

6.3.3 Implementation

The basis inverse is realized in the class template `QPE_basis_inverse`. It is parameterized with an exact number type `ET` and a compile time tag `IsLP`. The latter indicates whether the problem to solve is a linear program.

```
template < class ET, class IsLP >  class QPE_basis_inverse;
```

The solver of the QPE contains a basis inverse as data member.

```
template < class QPErep >
class QPE_solver {
    // ...
    QPE_basis_inverse<typename QPErep::ET,
                      typename QPErep::Is_linear>  inv_M_B;
    // ...
};
```

As in the implementation of the solver, the tag `IsLP` lets the compiler decide which code to generate for the basis inverse, resulting in an efficient tailored implementation for the specific problem to solve.

Following the rational representation (6.5), we store the integral part of the basis inverse as a vector of rows, each row being a vector of entries, representing the numerators. The denominator is kept separately.

```
template < class ET, class IsLP >
class QPE_basis_inverse {
    // ...
    std::vector< std::vector<ET> >  M;    // numerators
    ET                              d;    // denominator
    // ...
};
```

From the description of the update types we know, that in the QP case the index sets B_O and $E \,\dot{\cup}\, S_N$ can increase and decrease independently. This means that the number of rows (columns) in the submatrix of A (A^T) can increase and decrease. To handle this, one could append new rows and columns always at the lower right part of the matrix, and fill up a removed row and column by the last row and column of the matrix, respectively. The major drawback of this approach is, besides the additional work for the swapping, that the index sets get mixed up after some updates, which makes multiplying vectors with the matrix complicated and expensive. We want to keep the rows and columns corresponding to the same index set together, without having to move or copy parts of the basis inverse when rows are added to the submatrix of A. Therefore we must make sure that enough space is available in the upper left part of the matrix. Fortunately, the size of $E \,\dot{\cup}\, S_N$ is bounded by $l := \min\{n, m\}$ (cf. Section 2.4), so we are able to reserve enough space for the upper left part at once when the matrix is initialized, see Figure 6.1.

A full description of the basis inverse implementation can be found in [90].

6.4 Pricing Strategies

Pricing is the process of finding the entering variable, i.e. a nonbasic variable x_j with $\mu_j < 0$ (cf. Section 2.3.1), where μ_j is defined as

$$\mu_j = c_j + A_j^T \lambda + 2D_{B,j}^T x_B^* \,.$$

Taking inequality constraints into account (cf. Section 2.4), we get

$$\mu_j = c_j + A_{E \dot{\cup} S_N, j}^T \lambda_{E \dot{\cup} S_N} + 2D_{B_O,j}^T x_{B_O}^* \,.$$

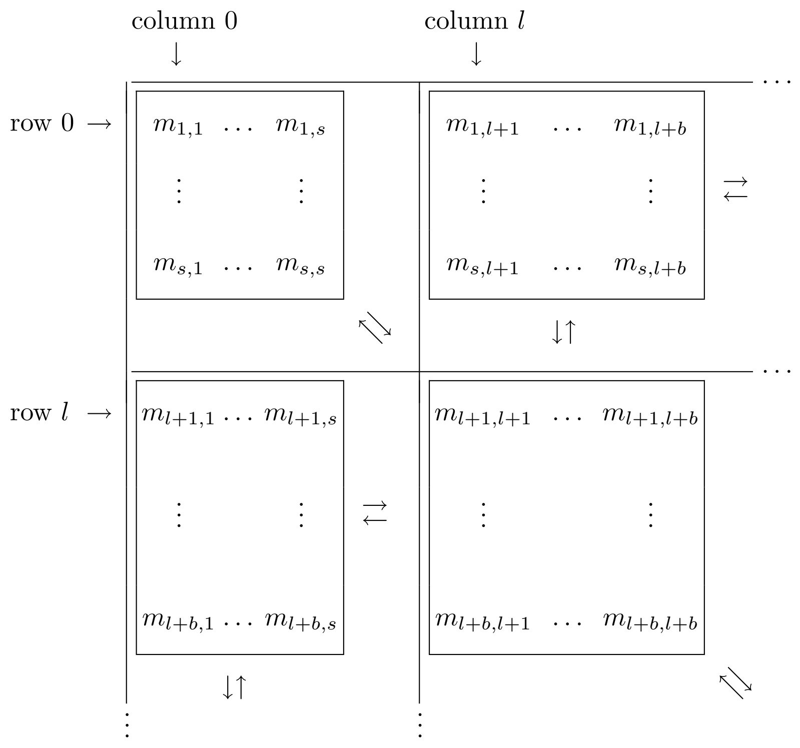

column 0
column l
row 0
row l
$m_{1,1}$ … $m_{1,s}$
$m_{s,1}$ … $m_{s,s}$
$m_{1,l+1}$ … $m_{1,l+b}$
$m_{s,l+1}$ … $m_{s,l+b}$
$m_{l+1,1}$ … $m_{l+1,s}$
$m_{l+b,1}$ … $m_{l+b,s}$
$m_{l+1,l+1}$ … $m_{l+1,l+b}$
$m_{l+b,l+1}$ … $m_{l+b,l+b}$

Figure 6.1: *Memory layout of basis inverse, $s := |E \dot\cup S_N|$, $b := |B_O|$*

Since the basis inverse has a rational representation, so has the current solution, and the entries of the vectors $\lambda_{E \dot\cup S_N}$ and $x^*_{B_O}$ are quotients with common denominator d. We obtain

$$\hat\mu_j = d\, c_j + A^T_{E \dot\cup S_N, j} \hat\lambda_{E \dot\cup S_N} + 2 D^T_{B_O, j} \hat x^*_{B_O} \,, \tag{6.16}$$

where $\hat\lambda_{E \dot\cup S_N}$ and $\hat x^*_{B_O}$ contain the numerators of $\lambda_{E \dot\cup S_N}$ and $x^*_{B_O}$, respectively. The values μ_j and $\hat\mu_j$ agree in sign because d is positive by definition.

Usually many nonbasic variables qualify for entering the current basis, in which case we have the freedom to choose one of them. The actual choice is done by the *pricing strategy* according to some *pivot rule*.

6.4.1 Partial Pricing

Testing all nonbasic variables in one iteration of the pivot step can be very expensive, if the set of nonbasic variables N is large. This is the case for optimization problems with high *variables-to-constraints* ratio, which we have in our applications. The idea of *partial pricing* is to maintain a set S of active variables, which is initially relatively small (see below). The entering variable is chosen from the variables in S. Only if no active variable qualifies for entering the current basis, the remaining nonbasic variables are tested.

Define $\min(R)$ to be the first index j with $x_j \in R$ and $\mu_j < 0$ for an ordered set $R \subseteq N$.

Algorithm 6.1 *(returns an entering variable x_j or* `optimal`*)*

```
PartialPricing($S$):
    $j := \min(S)$
    IF $\mu_j < 0$ THEN
        RETURN $x_j$
    ELSE
        $V := \{k \in N \backslash S \mid \mu_k < 0\}$
        IF $V = \emptyset$ THEN
            RETURN optimal
        ELSE
            $S := S \cup V$
            RETURN $\min(V)$
        END
    END
```

When the current basis is updated, the entering variable x_j is removed from S. Any variable leaving the basis is appended to S.

In the LP case, the choice of $\min(S)$ (respectively $\min(V)$) as the entering variable is known as *Dantzig's rule*. Here, the idea is that variables with negative but large absolute μ_j yield a fast decrease in the objective function's value, because such variables violate the KKT conditions by a larger 'amount' than others.

The intuition behind partial pricing is that S and V are always small and that S is augmented only a few times. In this case, most pricing steps are cheap, because they operate on a small set of active variables. Furthermore, only a few runs through the whole set N of nonbasic variables to find the set V are needed. Exactly the same intuition lies behind Clarkson's LP algorithm [21]. It works in the dual setting (i.e. with few variables and many constraints) and can easily be formulated as a dual simplex method. The interpretation of Clarkson's algorithm as a dual partial pricing scheme has already been suggested by Adler and Shamir [1].

A theoretical analysis of partial pricing in the LP case has been done by Gärtner and Welzl [47]. They showed the following under the assumption of a nondegenerate linear program: if we choose $|S| = m\sqrt{n/2}$, then S is augmented at most m times and the expected size of S is bounded by $2(m{+}1)\sqrt{n/2}$. Although this assumption does not always hold in practice, we nevertheless keep applying the partial pricing strategy and find that it still works well. We use $m\sqrt{n/2}$ as the initial size of S in our implementation.

6.4.2 Filtered Pricing

Another way of speeding up the pricing step is to use fast floating point arithmetic instead of exact arithmetic. The following scheme is based on a similar scheme proposed by Gärtner [39] for linear programs, which we adapted to our quadratic programming solver.

Instead of evaluating (6.16) exactly, we compute floating point approximations

$$\tilde{\mu}_j = (\tilde{d} \otimes c_j) \oplus (A^T_{E\dot{\cup}S_N,j} \odot \tilde{\lambda}_{E\dot{\cup}S_N}) \oplus (2D^T_{B_O,j} \odot \tilde{x}^*_{B_O}) \,, \qquad (6.17)$$

where $\tilde{\lambda}_{E\dot{\cup}S_N}$, $\tilde{x}^*_{B_O}$, and $\tilde{d}$ are nearest floating point approximations to $\hat{\lambda}_{E\dot{\cup}S_N}$, $\hat{x}^*_{B_O}$, and d, respectively. They are computed once in the beginning of the pricing step. In case d or one of the entries of $\hat{\lambda}_{E\dot{\cup}S_N}$ or $\hat{x}^*_{B_O}$ is larger than the largest representable floating point number, $\hat{\lambda}_{E\dot{\cup}S_N}$, $\hat{x}^*_{B_O}$, and d are scaled by a suitable power of two in advance. The operators $\oplus$, $\otimes$, and $\odot$ denote the floating point addition, multiplication, and inner product, respectively.

Using Dantzig's rule, the obvious candidate for the entering variable is the nonbasic variable x_j with smallest value $\tilde{\mu}_j$. For the correctness of the algorithm, it does not matter whether $x_j = \min(N)$ really holds, i.e. if we found the 'best' entering variable according to the rule. The important property of x_j is that $\hat{\mu}_j$ is negative, which can be easily checked using exact arithmetic. The benefit of Dantzig's rule in our context is that $\hat{\mu}_j$ is very likely to be negative, because it has been found to be the smallest among all, with floating point arithmetic computed values.

If the exact check of $\hat{\mu}_j$ succeeds, x_j is the entering variable. Otherwise we have to verify, that really no improving variable exists. The straightforward solution would be to recompute all $\hat{\mu}_j$s with exact arithmetic to check whether an improving variable has been missed due to rounding errors in the floating point computations. This would be quite expensive and is not necessary in most cases. We already know that all inexact values $\tilde{\mu}_j$, $j \in N$ are nonnegative, and a candidate for the entering variable has been missed if and only if some exact value $\hat{\mu}_j$ is negative. Lemma 6.2 below gives two error bounds on $\tilde{\mu}_j$ that let us deduce $\tilde{\mu}_j \geq 0$ in case $\tilde{\mu}_j$ is sufficiently far above zero. Only the variables that cannot be decided using these error bounds need to be dealt with exact arithmetic. The hope is that most values are 'caught' by this high-level *floating point filter*, which is usually the case in our applications.

Lemma 6.2 (Gärtner [39]) *Let*

$$\begin{aligned}
R_0 &:= \max_{i=1}^{n} |c_i| \,, && \\
R_k^A &:= \max_{i=1}^{n} |A_{k,i}| \,, && k \in [m], \\
R_k^D &:= \max_{i=1}^{n} |2D_{k,i}| \,, && k \in [n], \\
C_j &:= \max\{\, |c_j| \,,\, \max_{i=1}^{m} |A_{i,j}| \,,\, \max_{i=1}^{n} |2D_{i,j}| \,\} \,, && j \in [n],
\end{aligned}$$

be the row and column maxima of the quadratic program. Define

$$\begin{aligned}
U &:= \max\{\, \tilde{d} \otimes R_0 \,,\, \max_{i \in E \dot\cup S_N} (\, |\tilde{\lambda}_i| \otimes R_i^A \,) \,,\, \max_{i \in B_O} (\, |\tilde{x}_i^*| \otimes R_i^D \,) \,\} \,, \\
W &:= \max\{\, \tilde{d} \,,\, \max_{i \in E \dot\cup S_N} |\tilde{\lambda}_i| \,,\, \max_{i \in B_O} |\tilde{x}_i^*| \,\} \,.
\end{aligned}$$

If the floating point arithmetic has p bits of precision, then

$$|\tilde{\mu}_j - \hat{\mu}_j| \leq \min\{\, U \otimes q \,,\, W \otimes q \otimes C_j \,\}, \qquad j \in N \,, \tag{6.18}$$

where $q = (1 + 1/64)\,(|E \dot\cup S_N| + |B_O| + 1)\,(|E \dot\cup S_N| + |B_O| + 2)\, 2^{-p}$.

Note that the bounds in (6.18) can be exactly evaluated with floating point arithmetic. This is important, because otherwise rounding errors could occur in computing the error bounds. Since q is quite small, the bounds are usually very good. A good choice for the floating point arithmetic is the type *Double* as defined in the IEEE standard 754 for binary floating-point arithmetic [55], which has $p = 53$ bits of precision. In C++ it is realized as the type `double`, which we use in our implementation of filtered pricing.

Proof. Let x and y be vectors of length l and let $u := 2^{-p}$ be the *unit roundoff* of the floating point arithmetic, also known as *machine epsilon.* Classical results of Forsythe and Moler [34] give

$$\tilde{x}_i = x_i(1{+}\delta_i)\,, \qquad x_i = \tilde{x}_i(1{+}\varepsilon_i)\,, \qquad \tilde{x}_i y_i = \tilde{x}_i \otimes y_i\,(1{+}\eta_i)\,,$$

with $|\delta_i| \leq u$, $|\varepsilon_i| \leq u$, and $|\eta_i| \leq u$ for $i \in [l]$. Furthermore, if $lu < 0.01$, then

$$\tilde{x}^T \odot y = \sum_{i=1}^{l} \tilde{x}_i y_i (1{+}1.01\,l\Theta_i u)$$

holds, with $|\Theta_i| \leq 1$ for $i \in [l]$. Using this, the error of the inner product in floating point arithmetic can be estimated as

$$\begin{aligned}
|\,\tilde{x}^T \odot y - x^T y\,| &= \Big|\sum_{i=1}^{l} (x_i y_i (1{+}\delta_i)(1{+}1.01\,l\Theta_i u) - x_i y_i)\Big| \\
&= \Big|\sum_{i=1}^{l} \tilde{x}_i y_i (1{+}\varepsilon_i)(1.01\,l\Theta_i u + \delta_i + 1.01\,l\delta_i\Theta_i u)\Big| \\
&\leq \Big|\sum_{i=1}^{l} \tilde{x}_i y_i\Big|\,(1{+}u)(1.01\,lu + u + 1.01\,lu^2) \\
&\leq \max_{i=1}^{l} |\tilde{x}_i y_i|\,(1{+}u)(1.01\,l^2 u + lu + 1.01\,l^2 u^2) \\
&\leq \max_{i=1}^{l} |\tilde{x}_i \otimes y_i|\,(1{+}u)^2(1.01\,l^2 u + lu + 1.01\,l^2 u^2) \\
&\leq \max_{i=1}^{l} |\tilde{x}_i \otimes y_i|\;1.01\,l\,(l{+}1)\,u\,.
\end{aligned}$$

The last inequation holds for any practical value of l, if $u = 2^{-53}$.

We bound the maximum from above in two ways. Firstly, we have

$$\max_{i=1}^{l} |\tilde{x}_i \otimes y_i| \leq \max_{i=1}^{l}(\,|\tilde{x}_i| \otimes \max_{y} |y_i|\,)\,,$$

where y runs over all vectors we consider during the pricing. Secondly, we get

$$\max_{i=1}^{l} |\tilde{x}_i \otimes y_i| \leq \max_{i=1}^{l} |\tilde{x}_i| \otimes \max_{i=1}^{l} |y_i|\,.$$

Majorizing the constant 1.01 by (1+1/64) yields a bound exactly computable with floating point arithmetic. Defining $x^T := (d, \hat{\lambda}^T_{E\dot{\cup}S_N}, \hat{x}^{*\,T}_{B_O})$ and $y^T := (c_j, A^T_{E\dot{\cup}S_N,j}, 2D^T_{B_O,j})$ completes the proof. □

We apply the bounds of the lemma in the following way. At first, we check whether $\tilde{\mu}_j \geq U \otimes q$ holds. If not, we test $\tilde{\mu}_j$ against the second bound $W \otimes q \otimes C_j$, which requires one additional multiplication per variable. Only in case the second test also fails, we have to resort to exact arithmetic to determine the sign of $\hat{\mu}_j$.

6.4.3 Implementations

The whole pricing step is encapsulated in a class. It can be replaced with other pricing strategies derived from a given base class. This feature has been used to perform the tests involving full and partial pricing and different arithmetic, see Chapter 8.

The class template `QPE_pricing_strategy` is the base class for all pricing strategies. It is parameterized with the representation class `QPErep` of the ambient solver and optionally with a fast and possibly inexact number type used in the filtered pricing.

```
template < class QPErep, class NT = double >
class QPE_pricing_strategy {
  protected:
    const QPE_solver<QPErep>*  solverP;
    // ...
};
```

The pointer to the ambient solver gives access to the original problem and the current solution. The variable `solverP` is set, when the pricing strategy becomes the current pricing strategy of the solver.

The member function `pricing` performs the actual pricing step. It is called by the ambient solver and returns either the index of the entering variable or -1 to indicate optimality.

```
template < class QPErep, class NT >
class QPE_pricing_strategy {
  public:
    virtual  int  pricing( ) const = 0;
    // ...
};
```

Since `pricing` is declared *pure virtual*, it has to be reimplemented by the derived pricing strategy classes.

Any pricing strategy has to check the sign of μ_j for all nonbasic variables, or at least for a subset of them. We provide two member functions which compute $\hat{\mu}_j$ and $\tilde{\mu}_j$, respectively. One uses exact arithmetic over the number type `QPErep::ET`, the other one uses the fast and possibly inexact number type `NT` (as described in the previous section).

```
template < class QPErep, class NT >
class QPE_pricing_strategy {
  protected:
    typedef  typename QPErep::ET  ET;
    ET  mu   ( int j) const;
    NT  mu_NT( int j) const;
    // ...
};
```

These member functions are called several times during the pricing step. Thus, we provide a member function `init_NT` that computes the floating point approximations $\tilde{\lambda}_{E\dot{\cup}S_N}$, $\tilde{x}^*_{B_O}$, and $\tilde{d}$ and stores the values for later use by the member function `mu_NT`. Pricing strategies using this function have to call `init_NT` at the beginning of their `pricing` member function.

Some pricing strategies maintain an internal status, e.g. the set of active variables in case of partial pricing. The necessary initialization can be done in a reimplementation of the `init` member function, which is called by the ambient solver. At the end of the `pricing` member function, the pricing strategy knows the entering variable, but what about leaving variables? The member function `leave_basis` is called by the ambient solver with the corresponding index each time a variable leaves the basis. In addition, the pricing strategy is notified by a call to the `transition` member function if the solver switches from phase I to phase II.

```
template < class QPErep, class NT >
class QPE_pricing_strategy {
  public:
    virtual  void  init( ) { }
    virtual  void  transition( ) { }
    virtual  void  leave_basis( int i) { }
    // ...
};
```

The default implementations of `init`, `transition`, and `leave_basis` do nothing, but they can be reimplemented by the derived pricing strategy classes.

We provide predefined pricing strategies for all four combinations of full/partial pricing and exact/filtered pricing:

```
template < class QPErep >  class QPE_full_exact_pricing;
template < class QPErep >  class QPE_partial_exact_pricing;
template < class QPErep, class NT = double >
                           class QPE_full_filtered_pricing;
template < class QPErep, class NT = double >
                           class QPE_partial_filtered_pricing;
```

In the sequel, we describe, as an example, the implementation of the most sophisticated and usually most efficient strategy, namely the combination of partial and filtered pricing.

The pricing starts with finding the active variable $x_j \in S$ with smallest value $\tilde{\mu}_j$ using floating point arithmetic. If the exact check $\hat{\mu}_j < 0$ succeeds, x_j is returned. Otherwise the smallest $\tilde{\mu}_j$ among the remaining nonbasic variables is computed using floating point arithmetic a second time. Again, if the exact check $\hat{\mu}_j < 0$ succeeds, x_j is the entering variable. In this case, S is augmented with V, i.e. all nonbasic variables with $\tilde{\mu}_j < 0$ found during the last scan are added to the set of active variables. If no entering variable has been found so far, we have to verify that really no entering variable exists. For each nonbasic variable, we check $\tilde{\mu}_j$ against the two bounds of Lemma 6.2. If this does not decide the sign of $\hat{\mu}_j$, we use exact arithmetic to determine it. In case a negative $\hat{\mu}_j$ is found, x_j is returned, otherwise optimality has been certified. The following pseudocode describes the partial pricing strategy.

Algorithm 6.3 *(returns an entering variable x_j or* `optimal`*)*

```
PartialFilteredPricing($S$):
  $j := \arg\min\{ \tilde{\mu}_k \mid k \in S \}$
  IF $\hat{\mu}_j < 0$ THEN
      RETURN $x_j$
  ELSE
      $V := \{k \in N \backslash S \mid \tilde{\mu}_k < 0\}$
      IF $V \neq \emptyset$ THEN
          $j := \arg\min\{ \tilde{\mu}_k \mid k \in V \}$
          IF $\hat{\mu}_j < 0$ THEN
              $S := S \cup V$
              RETURN $x_j$
          END
      ELSE
          FOREACH $k$ IN $N$ DO
              IF (NOT $\tilde{\mu}_k \geq U \otimes q$) AND (NOT $\tilde{\mu}_k \geq W \otimes q \otimes C_k$) THEN
                  IF $\hat{\mu}_k < 0$ THEN
                      RETURN $x_j$
                  END
              END
          END
          RETURN optimal
      END
  END
```

The approximate values $\tilde{\mu}_k$ and the exact values $\hat{\mu}_k$ are computed using the base class' member functions `mu_NT` and `mu`, respectively.

To apply the error bounds of the floating point filter, we need the row and column maxima of the quadratic program as defined in Lemma 6.2.

Computing these maxima completely during the initialization of the pricing strategy would require to access all entries of A and D at least once, resulting in $O(mn+n^2)$ initialization time. Instead, we use a variant of *lazy evaluation*, i.e. only the needed maxima are computed respectively updated 'on-the-fly' in each pricing step. Initially, only R_0 is computed while the other row maxima R_k^A and R_k^D are set to zero. The column maxima C_j are initialized with $|c_j|$ for $j \in [n]$.

Just before the `FOREACH` loop in Algorithm 6.3, we compute R_k^A for each $k \in E \dot\cup S_N$ and R_k^D for each $k \in B_O$. The C_js are updated with the absolute values of A's rows in $E \dot\cup S_N$ and D's rows in B_O. To avoid repeated computations of the same values, we store all maxima computed so far and keep track of the rows already handled. This scheme avoids unnecessary computations, thus providing the needed maxima efficiently at minimal cost.

Further details of the implementations can be found in [91].

6.5 Conclusion

We presented a C++ implementation of our quadratic programming algorithm. It was carefully designed to be efficient and easy to use at the same time. We reached our design goals by using generic programming with class templates, compile time tags and iterators. The flexibility for the pricing strategy was achieved through an object-oriented design with a base class and virtual functions. The combination of exact and floating point arithmetic in the partial filtered pricing strategy leads to a considerable speed-up.

A preliminary version of the quadratic programming engine is already a (hidden) part of CGAL. It is used to solve geometric quadratic programming problems, see the next chapter.

Chapter 7

Geometric Optimization Problems in Cgal

This chapter describes the implementations of some geometric optimization problems in Cgal, namely the first three problems presented in Chapter 3.

7.1 Polytope Distance

7.1.1 Definition

An object of the class `Polytope_distance_d<Traits>` represents the (squared) distance between two convex polytopes, given as the convex hulls of two finite (multi)sets of points in d-dimensional Euclidean space $\mathbb{E}_d$. For point sets P and Q we denote by $pd(P, Q)$ the distance between the convex hulls of P and Q. Note that $pd(P, Q)$ can be degenerate, i.e. $pd(P, Q) = \infty$ if P or Q is empty.

We call two inclusion-minimal subsets S_P of P and S_Q of Q with $pd(S_P, S_Q) = pd(P, Q)$ a *pair of support sets*, the points in S_P and S_Q are the *support points*. A pair of support sets has size at most $d+2$

(by size we mean $|S_P| + |S_Q|$). The distance between the two polytopes is *realized* by a pair of points p and q lying on the convex hull of S_P and S_Q, respectively, i.e. $||p - q|| = pd(P, Q)$. In general, neither the support sets nor the realizing points are necessarily unique.

The underlying solver can cope with all kinds of input, e.g. P and Q may be in non-convex position or points may occur more than once. The algorithm computes a pair of support sets S_P and S_Q and the corresponding realizing points p and q.

7.1.2 Solution

Given the two point sets as $P = \{ p_1, \ldots, p_r \}$ and $Q = \{ q_1, \ldots, q_s \}$ with $r+s = n$, let $x^* = (x^*_1, \ldots, x^*_n)$ be an optimal solution to

$$\begin{array}{lll} \text{(PD)} & \text{minimize} & x^T C^T C\, x \\ & \text{subject to} & \sum_{i=1}^{r} \quad x_i = 1 \\ & & \sum_{i=r+1}^{n} x_i = 1 \\ & & x \geq 0\,, \end{array} \tag{7.1}$$

with $C = (p_1, \ldots, p_r, -q_1, \ldots, -q_s)$. Then the support sets are determined by the positive x^*_is, namely

$$\begin{aligned} S_P &= \{p_i \in P \mid x^*_i > 0\}\,, \\ S_Q &= \{q_i \in Q \mid x^*_{r+i} > 0\}\,. \end{aligned}$$

The realizing points are convex combinations of the given points, i.e.

$$\begin{aligned} p &= \sum_{i=1}^{r} x^*_i\, p_i\,, \\ q &= \sum_{i=1}^{s} x^*_{i+r}\, q_i\,. \end{aligned}$$

7.1.3 Implementation

Matrix A in (7.1) has two rows of length n. The first row contains a block of r ones, followed by a block of s zeros. The second row contains the same blocks with ones and zeros swapped. Since A is accessed column-wise, we store it as a vector of C-arrays of length 2. Since b is the 1-vector and c the 0-vector, we represent them with a special iterator referring always to the same constant value.

The interesting part is the representation of the objective matrix. Each entry of D is an inner product of two input points, in particular

$$D_{i,j} = \begin{cases} p_i^T\, p_j & \text{if } 1 \leq i \leq r\ , 1 \leq j \leq r \\ -p_i^T\, q_{j-r} & \text{if } 1 \leq i \leq r\ , r < j \leq n \\ -q_{i-r}^T\, p_j & \text{if } r < i \leq n\ , 1 \leq j \leq r \\ q_{i-r}^T\, q_{j-r} & \text{if } r < i \leq n\ , r < j \leq n \end{cases} \tag{7.2}$$

by the definition of C in (7.1). The idea for the implicit representation of D is as follows. If the entry in the i-th row and the j-th column is accessed, the corresponding input points are multiplied 'on-the-fly' and the signed result is returned.

We store matrix C in the variable `points`, which is a vector of signed points, thus allowing random-access to the input points. The class template `PD_D_iterator` is initialized with the iterator `points.begin()` referring to the first input point in C. When dereferenced at position i, it returns an object of type `PD_D_row_iterator` initialized with the index i and the iterator `points.begin()`. This object is another iterator implicitly representing the i-th row of D. When dereferenced at position j, it accesses the points at positions i and j, computes their inner product, and returns the result.

```
template < class Point >
class PD_D_row_iterator {
    // types
    typedef  typename std::vector<Point>::const_iterator
                  Point_it;
    typedef  ...  CT;           // points' coordinate type

    // data members
    Point_it  points;           // iterator to C
    int       i;                // row    index
    int       j;                // column index

  public:
    PD_D_row_iterator( Point_it it, int row)
```

```
        : points( it), i( row), j( 0) { }
    CT  operator * ( )
        { return inner_product( points[ i], points[ j]); }
    // ...
};

template < class Point >
class PD_D_iterator {
    // types
    typedef  typename std::vector<Point>::const_iterator
                                       Point_it;
    typedef  PD_D_row_iterator<Point>  Row_it;

    // data members
    Point_it  points;           // iterator to C
    int       i;                // row index

  public:
    PD_D_iterator( Point_it it) : points( it), i( 0) { }
    Row_it  operator * ( ) { return Row_it( points, i); }
    // ...
};
```

The primal version of the polytope distance problem we use here has the special properties of a symmetric objective matrix and no inequality constraints. The following class template collects all types needed by the QPE to solve (7.1).

```
template < class Traits, class ET_ >
class PD_rep {
    typedef  ...  Point;        // point type (from Traits)
    typedef  ...  CT;           // points' coordinate type
  public:
    typedef  std::vector< CT[ 2] >::const_iterator  A_iterator;
    typedef  Const_value_iterator< CT >             B_iterator;
    typedef  Const_value_iterator< CT >             C_iterator;
    typedef  PD_D_iterator< Point >                 D_iterator;

    enum Row_type { EQUAL };
    typedef  Const_value_iterator< Row_type >       R_iterator;

    typedef  Tag_false  Is_linear;
    typedef  Tag_true   Is_symmetric;
    typedef  Tag_true   Has_no_inequalites;
    typedef  ET_   ET;
};
```

An implementation of the dual version of the polytope distance problem has been done for the experiments described in the next chapter. It will become part of CGAL in a future release.

7.2 Smallest Enclosing Ball

7.2.1 Definition

An object of the class `Min_sphere_d<Traits>` is the unique sphere of smallest volume enclosing a finite (multi)set of points in d-dimensional Euclidean space $\mathbb{E}_d$. For a set P we denote by $ms(P)$ the smallest sphere that contains all points of P. Note that $ms(P)$ can be degenerate, i.e. $ms(P) = \emptyset$ if $P = \emptyset$ and $ms(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $ms(S) = ms(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most $d{+}1$, and all its points lie on the boundary of $ms(P)$. In general, neither the support set nor its size are unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S together with the center and the (squared) radius of the smallest enclosing sphere.

7.2.2 Solution

Given the point set as $P = \{p_1, \ldots, p_n\}$, let $x^* = (x_1^*, \ldots, x_n^*)$ be an optimal solution to

$$\begin{array}{lll} \text{(MB)} & \text{minimize} & x^T C^T C\, x - \sum_{i=1}^n p_i^T p_i\, x_i \\ & \text{subject to} & \sum_{i=1}^n x_i = 1 \\ & & x \geq 0\,, \end{array} \tag{7.3}$$

with $C = (p_1, \ldots, p_n)$. Then the support set is determined by the positive x_i^*s, namely

$$S = \{p_i \in P \mid x_i^* > 0\}\,.$$

7.2.3 Implementation

Similar to the polytope distance problem, the quadratic program (7.3) can be represented as follows. Matrix A contains only a single row of ones and b is the one-dimensional 1-vector. The entries of c are the input points' negated squared lengths. They are computed in advance and stored in a vector of the points' coordinate type. Again, the implicit representation of D is the most interesting part. Here, we can reuse the two iterator classes described in the previous section. The only difference is that C contains only the points of P.

The smallest enclosing ball problem has the special properties of a symmetric objective matrix and no inequality constraints. The following class template collects all types needed by the QPE to solve (7.3).

```
template < class Traits, class ET_ >
class MB_rep {
    typedef  ...  Point;        // point type (from Traits)
    typedef  ...  CT;           // points' coordinate type
  public:
    typedef  Const_value_iterator< Const_value_iterator<CT> >
                                                 A_iterator;
    typedef  Const_value_iterator< CT >         B_iterator;
    typedef  std::vector< CT >::const_iterator  C_iterator;
    typedef  PD_D_iterator< Point >             D_iterator;

    enum Row_type { EQUAL };
    typedef  Const_value_iterator< Row_type >   R_iterator;

    typedef  Tag_false  Is_linear;
    typedef  Tag_true   Is_symmetric;
    typedef  Tag_true   Has_no_inequalites;

    typedef  ET_   ET;
};
```

7.3 Smallest Enclosing Annulus

7.3.1 Definition

An object of the class `Min_annulus_d<Traits>` is the unique annulus (region between two concentric spheres with radii r and R, $r \leq R$) enclosing a finite (multi)set of points in d-dimensional Euclidean space $\mathbb{E}_d$, where the difference $R^2 - r^2$ is minimal. For a point set P we denote by $ma(P)$ the smallest annulus that contains all points of P. Note that $ma(P)$ can be degenerate, i.e. $ma(P) = \emptyset$ if $P = \emptyset$ and $ma(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $ma(S) = ma(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most $d+2$, and all its points lie on the boundary of $ma(P)$. In general, the support set is not necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S together with the center and the (squared) radii of the smallest enclosing annulus.

7.3.2 Solution

Given the point set as $P = \{\, p_1, \dots, p_n \,\}$, consider an optimal solution to the linear program

$$\begin{array}{lll} \text{(MA')} & \text{minimize} & \sum_{i=1}^{n} {p^i}^T p^i \, \mu_i - \sum_{i=1}^{n} {p^i}^T p^i \, \lambda_i \\ & \text{subject to} & \sum_{i=1}^{n} 2p_j^i \, \lambda_i + \sum_{i=1}^{n} 2p_j^i \, \mu_i = 0 \qquad j = 1 \dots d \\ & & \sum_{i=1}^{n} \lambda_i = 1 \\ & & \sum_{i=1}^{n} \mu_i = 1 \\ & & \lambda, \mu \geq 0 \,. \end{array} \tag{7.4}$$

Then the values of the dual variables determine the solution to the smallest enclosing annulus problem. Let A_B^{-1} be the lower left part of the final basis inverse (cf. Section 6.3.1). The center c^* is obtained by

$$(\alpha^*, \beta^*, c_1^*, \dots, c_d^*) = -c_B^T A_B^{-1}$$

while the squared radii are computed from α^* and β^* as

$$r^2 = \alpha^* + ||c^*||^2 \,,$$
$$R^2 = \beta^* + ||c^*||^2 \,.$$

7.3.3 Implementation

The representation for (7.4) is straight forward. We keep A, b, and c explicitly as (a vector of) vectors of the points' coordinate type. All entries are computed and stored in advance.

The dual version of the smallest enclosing annulus problem we use here has the special properties of a linear objective function and no inequality constraints. The following class template collects all types needed by the QPE to solve (7.4).

```
template < class Traits, class ET_ >
class MA_rep {
    typedef  ...  Point;        // point type (from Traits)
    typedef  ...  CT;           // points' coordinate type
  public:
    typedef  std::vector< std::vector<CT> >::const_iterator
                                A_iterator;
    typedef  std::vector< CT >  B_iterator;
    typedef  std::vector< CT >  C_iterator;
    typedef  CT**               D_iterator;    // not used

    enum Row_type { EQUAL };
    typedef  Const_value_iterator< Row_type >  R_iterator;

    typedef  Tag_true   Is_linear;
    typedef  Tag_false  Is_symmetric;
    typedef  Tag_true   Has_no_inequalites;

    typedef  ET_   ET;
};
```

An implementation of the primal version of the smallest enclosing annulus problem has been done for the experiments described in the next chapter. It will become part of CGAL in a future release.

7.4 Conclusion

We sketched the implementations of three geometric quadratic programming problems as part of CGAL. These are the primal version of the polytope distance problem, the smallest enclosing ball problem, and the dual version of the smallest enclosing annulus problem.

We showed how to represent the programs efficiently, especially the implicit representation of the dense objective matrix D in the polytope distance and the smallest enclosing ball problems has been discussed.

Implementations of the dual version of the polytope distance problem and the primal version of the smallest enclosing annulus problem are already done and will be available in a future release of CGAL.

Chapter 8

Experimental Results

We tested our solver on three different problems, namely *polytope distance*, *smallest enclosing ball*, and *smallest enclosing annulus*, with various settings. In addition, we tested our implementation (within CGAL) of the *smallest enclosing ellipse* problem, which uses Welzl's method combined with our exact primitives as described in Chapter 4.

8.1 Test Environment

As our major test problem, we chose the smallest enclosing ball problem, because there the largest number of competing implementations was available to us. We performed extensive tests, and compared the results to the ones obtained using the other codes (including the QP solver of CPLEX).

The smallest enclosing annulus problem is mainly selected as a test problem, because it is an LP problem, and benchmarks have already been obtained for an exact LP solver on that problem. It turns out that although we solve LP as a special case of QP, our method is faster than the dedicated LP code described in [39]. This is due to the fact that great care has been taken to ensure runtime efficiency (see last but one chapter).

All tables show the performance of different codes on data of different dimensions, for different numbers of points, and different arithmetics (averaged over several runs in each case[1]). For smallest enclosing ball, the codes that were used are the `double`-only code `Miniball` of Gärtner [40], the algorithms of LEDA [68] and CGAL [19] (in the latter case also the dedicated 2-dimensional version), our new QP based method, and finally the QP-solver of CPLEX. For the smallest enclosing annulus, the exact LP solver of Gärtner [39] was compared with our new method.

Most tests have been performed with pseudo-random input, i.e. the coordinates were chosen as the lower-order 24 bits of the pseudo-random numbers generated by the function `random` from the C++ standard library. It uses a non-linear additive feedback random number generator with a very large period, which is approximately $16\,(2^{31}-1)$. Since `random` does not use a linear congruential generator, it is not subject to 'non-random' effects as described in [41], as far as we know. For smallest enclosing ball, annulus, and ellipse, we further have tested with degenerate inputs, sets of points lying exactly or almost on a circle or sphere.

The three types of arithmetics used are `double` (floating point only), `filtered` (a standard floating point filter approach in case of LEDA, and our hybrid scheme in case of the QP method), as well as `exact`, denoting full multiple precision number arithmetic.

In case of the QP solver, two pricing strategies are available, full and partial. Partial pricing is almost always faster than full pricing, which is therefore not tabulated for all tests.

With one exception, runtimes have been measured on a notebook PC with a Mobile Pentium III CPU running at 500 MHz under Linux-2.4, using the GNU `g++` compiler, version 2.95.2, at optimization level `-O3`. Because CPLEX was available to us only on a SUN Ultra-60 workstation, the results in Table 8.9 refer to that machine.

[1]In most cases, we averaged the runtimes over 100 runs. Only in a few cases, where a single run needed several minutes to complete, we averaged over 10 runs.

	random points	
d	10,000	100,000
2	1.4 s	15.8 s
3	1.7 s	18.2 s
5	2.7 s	25.5 s
10	5.1 s	48.3 s
15	8.1 s	1:13 min
20	11.9 s	1:44 min
30	29.2 s	3:51 min
50	1:36 min	10:06 min

Table 8.1: *Runtimes on random polytope distance problems*

8.2 Polytope Distance

Table 8.1 shows the results of our solver (the best available setting: partial filtered pricing) for polytope distance problems with various random point sets in various dimensions. The main 'message' of the table is that the solver is able to handle *large* instances *fast* and *exact*. A more detailed picture is obtained from the results for smallest enclosing balls, where we compete against other codes.

8.3 Smallest Enclosing Ball

While all methods that are considered can handle large point sets in dimension 2, there are quite some differences in efficiency between floating point, filtered and exact versions of the same code (Table 8.2). Usually, our exact QP solver is only slightly slower than the `double` versions of the other codes, but dramatically faster than their exact implementations. The LEDA version is still comparable, because it applies a filtered approach itself. In cases the table entry is blank, we did not wait for the result, the time here is at least half an hour.

As degenerate inputs for $d = 2$, we have chosen two sets of points lying exactly on a circle. The small set has coordinates such that the squares still fit into a `double` value, while the large one has not. The only `double` implementations that can reasonably handle those sets are the `Miniball` routine, and CGAL's dedicated 2-dimensional code (Table 8.3). Again,

Algorithm ($d = 2$)		**random points**		
		10,000	100,000	1,000,000
Miniball	(double)	4 ms	50 ms	646 ms
LEDA Min_circle	(double)	14 ms	223 ms	2.8 s
CGAL Min_sphere	(double)	13 ms	169 ms	1.7 s
CGAL Min_circle	(double)	45 ms	487 ms	4.6 s
QP Solver	(partial,double)	14 ms	160 ms	1.6 s
QP Solver	(full,double)	28 ms	331 ms	3.4 s
LEDA Min_circle	(filtered)	51 ms	561 ms	6.2 s
QP Solver	**(partial,filtered)**	31 ms	327 ms	3.2 s
QP Solver	(full,filtered)	44 ms	498 ms	5.5 s
QP Solver	(partial,exact)	1.7 s	16.9 s	2:49 min
QP Solver	(full,exact)	3.3 s	31.8 s	
CGAL Min_sphere	(exact)	3.2 s	30.4 s	
CGAL Min_circle	(exact)	5.3 s	52.2 s	

Table 8.2: *Runtimes on random miniball problems,* $d = 2$

Algorithm ($d = 2$)		**points on circle**		
		6,144	13,824	6,144 (perturbed)
Miniball	(double)	<1 ms	5 ms	5 ms
CGAL Min_circle	(double)	10 ms		29 ms
LEDA Min_circle	(filtered)	1.27 s	3.1 s	6.5 s
QP Solver	**(partial,filtered)**	412 ms	2.5 s	60 ms
QP Solver	(full,filtered)	431 ms	4.8 s	110 ms
QP Solver	(partial,exact)	416 ms	7.1 s	1.3 s
QP Solver	(full,exact)	845 ms	23.7 s	4.3 s
CGAL Min_sphere	(exact)	423 ms	1.0 s	3.4 s
CGAL Min_circle	(exact)	894 ms	1.9 s	3.4 s

Table 8.3: *Runtimes on degenerate miniball problems,* $d = 2$

Algorithm ($d = 3$)		**points on sphere**
		10,000 (perturbed)
Miniball	(double)	19 ms
QP Solver	**(partial,filtered)**	123 ms
QP Solver	(full,filtered)	127 ms
QP Solver	(partial,exact)	10.9 s
QP Solver	(full,exact)	11.6 s
CGAL Min_sphere	(exact)	23.8 s

Table 8.4: *Runtimes on degenerate miniball problem,* $d = 3$

the exact QP solver is faster than LEDA's filtered approach, and much faster than all exact versions. An interesting phenomenon occurs when one slightly perturbs the points, so that they are no longer cocircular. The QP solver becomes much faster because the perturbation already suffices to make the built-in error bounds work effectively: to verify optimality in the last iteration, no exact checks are necessary anymore, while they extensively happen for the non-perturbed degenerate input.

In $d = 3$, we have chosen a point set almost on a sphere, obtained by tiling the sphere according to longitude and latitude values. The only `double` code still able to handle this problem is `Miniball`; our filtered approach, however, is only by a factor of six slower, while the exact versions are out of the game (Table 8.4).

We also have results for higher dimensions (Tables 8.5, 8.6, and 8.7). These show how the missing 'curse of dimensionality' in the QP solver compensates for the effect of exact arithmetic, so that our exact solver is already faster than the inexact solver `Miniball` in dimension 20, for 10,000 points. (For $n = 100{,}000$ we are not far off, but for 1,000,000 points we reach the machine's memory limit). Note that the table entry is blank for most pure `double` versions in higher dimensions, which means that the results were too inaccurate.

Probably most interesting is that problems up to dimension 100 can routinely be handled, and even for 10,000 points in $d = 200$, we only need about three and a half minutes (Table 8.8). It should be noted that these results hold for random points, where we observed that the number of points that determine the final ball is quite small (much smaller than the dimension itself). In this situation, the QP-bases the algorithm needs to handle are relatively small, which makes the exact arithmetic fast.

Finally, we have performed comparisons with the QP solver of CPLEX (an interior point code) using formulation (3.7) on page 41, showing that such codes are not competitive in our scenario (Table 8.9). It is interesting to note that the performance of the CPLEX solver mainly depends on the product of n and d (which is closely related to the number of nonzeros in the problem description), while we pay a penalty for larger values of d. However, the results show that our exact method is still superior to CPLEX for dimensions below 30, which is the range of dimensions our code is made for. It is clear that CPLEX's method will become superior as d goes higher up.

Algorithm *d*	**random points** (10,000)					
	2	3	5	10	15	20
Miniball (double)	4 ms	7 ms	16 ms	57 ms	168 ms	1.0 s
CGAL Min_sphere (double)	13 ms	18 ms				
QP Solver (partial,double)	14 ms	17 ms	26 ms	62 ms		
QP Solver (partial,filtered)	31 ms	41 ms	74 ms	219 ms	408 ms	847 ms
QP Solver (partial,exact)	1.7 s	2.1 s	3.4 s	7.2 s	11.5 s	17.1 s
CGAL Min_sphere (exact)	3.2 s	6.1 s	15.7 s	1:07 min	3:40 min	

Table 8.5: Runtimes on random miniball problems, $n = 10{,}000$

Algorithm *d*	**random points** (100,000)					
	2	3	5	10	15	20
Miniball (double)	50 ms	102 ms	249 ms	792 ms	1.5 s	3.2 s
CGAL Min_sphere (double)	169 ms	256 ms	546 ms			
QP Solver (partial,double)	160 ms	183 ms				
QP Solver (partial,filtered)	327 ms	403 ms	644 ms	1.4 s	2.5 s	5.9 s
QP Solver (partial,exact)	16.9 s	20.8 s	52.4 s	1:02 min	1:33 min	2:15 min
CGAL Min_sphere (exact)	30.4 s	1:01 min	2:50 min			

Table 8.6: Runtimes on random miniball problems, $n = 100{,}000$

Algorithm		**random points** (1,000,000)				
	d	2	3	5	10	15
Miniball	(double)	646 ms	1.1 s	2.5 s	9.2 s	7.3 s
CGAL Min_sphere	(double)	1.7 s	2.6 s	4.8 s	15.6 s	
QP Solver	(partial,double)	1.6 s	1.8 s			
QP Solver	**(partial,filtered)**	3.2 s	3.8 s	5.6 s	14.0 s	23.4 s
QP Solver	(partial,exact)	2:49 min				

Table 8.7: *Runtimes on random miniball problems, $n = 1{,}000{,}000$*

	random points	
d	10,000	100,000
30	3.3 s	13.1 s
50	10.0 s	53.3 s
100	25.2 s	1:32 min
200	3:23 min	
300	10:18 min	

	random points	
d	10,000	100,000
30	42	48
50	53	61
100	55	70
200	81	
300	114	

Table 8.8: *Runtimes and number of iterations on random miniball problems, QP solver with partial filtered pricing*

Algorithm		**random points**	
		100,000 (d=3)	10,000 (d=30)
CPLEX	(double)	12.8 s	10.2 s
QP Solver	**(partial,filtered)**	671 ms	6.8 s

Table 8.9: *Runtimes on two random miniball problems, compared to CPLEX*

8.4 Smallest Enclosing Annulus

We have tested our QP solver against the exact solver described in [39] which employs basically the same combination of exact and floating point arithmetic (denoted as LP simplex in the tables). However, as the results show, our solver is even faster than the dedicated LP solver. This is due to the fact that we invested much effort in optimizing the code.

Algorithm	**random points** ($d = 2$)	
	10,000	100,000
QP Solver (partial,double)	7 ms	115 ms
QP Solver (full,double)	31 ms	416 ms
LEDA Min_annulus (double)	2.2 s	28.7 s
QP Solver (partial,filtered)	20 ms	192 ms
QP Solver (full,filtered)	41 ms	546 ms
LP Simplex (partial,filtered)	246 ms	2.4 s
LEDA Min_annulus (filtered)	19.6 s	3:52 min
QP Solver (partial,exact)	4.0 s	48.1 s
QP Solver (full,exact)	13.8 s	2:20 min

Table 8.10: *Runtimes on random annulus problems, $d = 2$*

Algorithm	**points on circle** ($d = 2$)		
	6,144	13,824	6,144 (perturbed)
QP Solver (partial,filtered)	0.8 s	2.1 s	24 ms
QP Solver (full,filtered)	0.8 s	2.1 s	38 ms
LP Simplex (partial,filtered)	0.9 s	1.7 s	159 ms
LEDA Min_annulus (filtered)	2.8 s	6.5 s	
QP Solver (partial,exact)	0.8 s	2.1 s	3.3 s
QP Solver (full,exact)	4.1 s	13.5 s	14.4 s

Table 8.11: *Runtimes on degenerate annulus problems, $d = 2$*

Algorithm	**points on sphere** ($d = 3$)
	10,000 (perturbed)
QP Solver (partial,double)	39 ms
QP Solver (full,double)	58 ms
QP Solver (partial,filtered)	88 ms
QP Solver (full,filtered)	102 ms
LP Simplex (partial,filtered)	311 ms
QP Solver (partial,exact)	18.2 s
QP Solver (full,exact)	29.5 s

Table 8.12: *Runtimes on degenerate annulus problem, $d = 3$*

Algorithm	*d*	**random points** (10,000)						
		2	3	5	10	15	20	30
QP Solver	(partial,double)	7 ms	15 ms	20 ms	64 ms	143 ms	337 ms	
QP Solver	(full,double)	31 ms	76 ms	135 ms	546 ms	1.0 s	2.0 s	
QP Solver	**(partial,filtered)**	20 ms	44 ms	101 ms	795 ms	4.1 s	14.2 s	1:31 min
QP Solver	(full,filtered)	41 ms	92 ms	182 ms	957 ms	3.7 s	11.7 s	
LP Simplex	(partial,filtered)	246 ms	273 ms	359 ms	1.3 s	4.9 s	16.5 s	1:33 min
QP Solver	(partial,exact)	4.0 s	5.8 s	10.3 s	40 s	2:11 min	5:12 min	
QP Solver	(full,exact)	13.8 s	26.7 s	55.7 s	4:25 min	13:36 min		

Table 8.13: Runtimes on random annulus problems, $n = 10{,}000$

Algorithm	*d*	**random points** (100,000)						
		2	3	5	10	15	20	30
QP Solver	(partial,double)	115 ms	118 ms	164 ms	340 ms	747 ms	1.7 s	
QP Solver	(full,double)	466 ms	748 ms	1.6 s	4.9 s			
QP Solver	**(partial,filtered)**	192 ms	214 ms	350 ms	1.2 s	5.3 s	18.2 s	1:52 min
QP Solver	(full,filtered)	547 ms	841 ms	1.7 s	6.1 s			
LP Simplex	(partial,filtered)	1.1 s	2.4 s	2.6 s	4.0 s	8.8 s	23.7 s	4:23 min
QP Solver	(partial,exact)	49.1 s	53.9 s	1:28 min	4:04 min			
QP Solver	(full,exact)	2:22 min	5:43 min	11:25 min				

Table 8.14: Runtimes on random annulus problems, $n = 100{,}000$

As before, the filtered approach is much faster than the exact approach, and comparable to pure floating point solutions. Unlike in the case of smallest enclosing ball, `double` versions are still able to compute the correct result in higher dimensions, which indicates that true QP is more challenging for the numerics than LP (Tables 8.10, 8.13, and 8.14). We used the dual version (3.9) on page 43 for our tests, which turned out to be always faster than the primal version (3.8) on page 42. The more involved pivot step of the primal version (due to the handling of the slack variables) and a higher number of iterations resulted in a slow down by a factor of at least two.

In case of degenerate input (points on a circle), we observe the same phenomenon as with smallest enclosing balls: as soon as we slightly perturb the points, the filtered approach gets much faster, because less exact computations are necessary (Tables 8.11 and 8.12).

8.5 Smallest Enclosing Ellipse

Table 8.15 shows the results of our CGAL implementation for smallest enclosing ellipse problems. We tested the method on random point sets of different size as well as on degenerate sets consisting of points lying on or almost on a circle (described above). Note, CGAL `Min_circle` implements the same method of Welzl for the smallest enclosing circle problem. A comparison of the runtimes here with those in Tables 8.2 and 8.3 shows the following. As expected, the much more involved

Point Set ($d = 2$)	n	**CGAL Min_ellipse** (double)	(exact)
random	10,000	72 ms	50.1 s
	100,000	750 ms	8:51 min
	1,000,000	7.3 s	
on circle	6,144	4 ms	6.6 s
	13,824	12 ms	14.0 s
on circle (perturbed)	6,144	15 ms	5.6 s

Table 8.15: *Runtimes on smallest enclosing ellipse problems, $d = 2$*

arithmetic in the primitive operations in case of the smallest enclosing ellipse causes a significant increase in runtimes.

8.6 Conclusion

We performed numerous tests of our solver on the quadratic programming problems *polytope distance*, *smallest enclosing ball*, and *smallest enclosing annulus*, with different types of input data.

Our major test problem, namely the smallest enclosing ball problem, was used to compare our solver with other implementations. Among the competitors were mathematical programming codes as well as dedicated geometric algorithms.

Almost always the partial filtered pricing strategy was best. Using this combination of exact and floating point arithmetic yields very good runtimes. They are not far behind pure floating point implementations, but much faster than codes using exact arithmetic for all numerical operations.

For the smallest enclosing annulus problem, which is an LP problem, we even outperformed a dedicated LP code. This is due to the fact that great care has been taken to ensure runtime efficiency.

Finally, we showed that smallest enclosing ellipse problems with up to 100,000 points can be solved exactly using Welzl's method combined with our exact primitives.

Part III

Applications

Chapter 9

Two Applications

This chapter presents two applications, where our method was used to solve 'real-world' problems. As part of a joint project with a big Swiss bank, our quadratic programming solver was used to solve a large portfolio optimization problem. In another cooperation with a German company developing medical navigation systems, the smallest enclosing annulus implementation was used to calibrate the tip of medical probes.

9.1 Portfolio Optimization

In this section we describe the application of the quadratic programming solver in asset management.

9.1.1 Application Background

Recently, a coherent measure of risk, the so called *Conditional Value-at-Risk*, is gaining ground both in academia and in financial industry [103, 84, 65]. An extension of this methodology for multi-period portfolio optimization is desirable [23, 14]. In this context, a diploma thesis' project was assigned to two students at the Institute for Operations Research (IFOR), ETH Zurich. Their subject was to compare the risk measures 'Mean-Variance and Conditional Value-at-Risk in Portfolio Optimization' [8]. The asset management division of Credit Suisse (CSAM) in Zurich acted as industrial partner for this diploma thesis.

The models used in portfolio optimization are usually formulated as linear or quadratic programs. One of the tasks in the project was to compare different solvers for linear and quadratic programming problems. A preliminary version of our exact implementation described in Chapter 6 was one of them. During the project meetings, the need for exact solutions was discussed. The issue was raised by CSAM, because they encountered numerical problems when they tried to solve large portfolio optimization problems.

9.1.2 Problem Description

CSAM provided us with one of their 'problematic' problems. It is a multi-period portfolio optimization problem of a Swiss pension fonds ('Schweizer Pensionskasse').

The model includes 11 financial instruments, i.e. stocks or investment fonds, for which the best combination should be found. Transaction costs for buying and selling instruments are also modelled, resulting in 33 variables for one time period. The model contains five periods, so it has 165 variables in total. The problem is formulated as a quadratic program by using the Markowitz model to controll the risk of the portfolio.

At CSAM, they tried to solve this problem using CPLEX. It was neither able to find a feasible solution nor to show infeasibility of the quadratic program. CPLEX just terminated after some amount of time, indicating that it detected 'cycling'. Further details, e.g. version and parameter settings of CPLEX and the particular reason for its failure, were not available to us.

9.1.3 Results

The quadratic program of CSAM was given in `MPS` format, so we reused an `MPS` reader from a previous implementation of an inexact solver developed together with Bernd Gärtner. We extended the reader for quadratic programs, i.e. we added the handling of a `QMATRIX` section.

The problem finally feeded into the solver consisted of 165 original and 211 slack variables, resulting in a total of 371 variables (the preliminary version of the solver used here was not able to handle slack variables implicitly). It contained 271 constraints, devided in 60 equality and 211 inequality constraints.

Our solver (using full exact pricing) needed an overall runtime of 1 hour and 53 minutes. After 395 iterations in phase I, it returned that the problem is `INFEASIBLE`. We checked the solution and found it to be correct.

Since we were able to 'solve' the problem, our approach seems promising in this application domain. Although the computation took quite a while, future versions using partial filtered pricing on up-to-date hardware will need only a fraction of the observed runtime. At least we showed that our exact solver might be used as a tool in portfolio optimization.

9.2 Stylus Calibration

We describe the solution of a calibration problem that arised from a medical application in neurosurgery.

9.2.1 Application Background

In recent years, *intraoperative navigation systems* were recognized as a very powerful tool for surgical cases. On the one hand, they improve the results of such operations, while on the other hand, they make surgical interventions yet possible that where considered inoperable before. One such system is the computer aided *neurosurgery* navigation system NEN [87] developed by Functional Imaging Technologies (FIT) in Berlin, Germany.

The general aim of these systems is to enable a surgeon to utilize pre-operatively gathered information more effectively during the operation. This information is usually a three-dimensional image set of the region of interest, obtained by a computed tomography (CT) or magnetic resonance (MR) scanner. The overall idea consists in linking the information from the image set directly to the operating environment for navigation purposes. Like in most such guidance systems (see for example [33] and the references there) the basic approach to solve this question relies on a technique called *image registration*. A set of markers (so called *fiducials*) is attached to the skin (or even bone implanted in other systems) and special points on the markers serve as orientation landmarks. These landmarks are visible in the pre-operative volumetric image set and define a set of 3D 'image' points. Then, immediately prior to the intervention, a 3D *tracking system* is used to localize the landmark points a second time in the operating environment while the patient's head is fixed. This defines another set of 3D 'world' points. The image registration process computes a transformation defined by the two point sets, that maps each point from 'world' space to its counterpart in 'image' space.[1]

9.2.2 Problem Description

The 3D tracking system used by the NEN navigation system is based on electro-magnetic fields. A transmitter generates a pulsed DC magnetic field, which allows a corresponding receiver to localize itself within the electro-magnetic field. The position and orientation of the receiver in 3-space is measured at regular intervals and the information is handed over to the navigation system.

The surgeon uses a medical probe, the so called *stylus*, to navigate in the operating environment, see Figure 9.1. The receiver of the tracking system is integrated in the handle of the stylus. Since we are interested in the position (and orientation) of the stylus' tip, we have to *calibrate* the stylus, i.e. the offset vector between the receiver in the handle and the tip of the stylus has to be determined.

[1]Actually, the cooperation with FIT started with a project on the image registration problem, see [54].

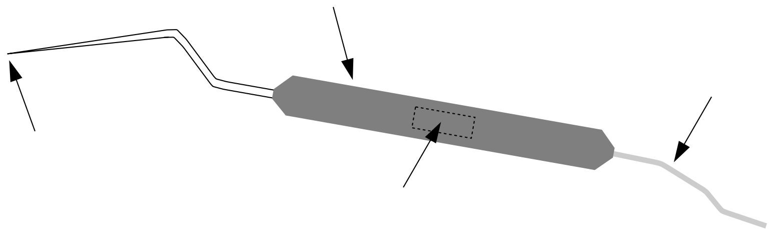


Figure 9.1: *Stylus of the navigation system*

9.2.3 Solution

The idea of the calibration algorithm is as follows. We place the tip of the stylus in a special tool, such that the stylus can be moved and rotated but the tip's position remains fixed, see Figure 9.2. When moving the stylus, all measured positions of the receiver lie (theoretically) on a particular half-sphere in 3-space. The tip's position is the center of the *smallest enclosing annulus* of the set P of measured positions. Note that the smallest enclosing ball of P cannot be used here, because the acquired positions cover only a half-sphere. In general, the center of this half-sphere does not coincide with the center of the smallest enclosing ball of P.

In practice, the points in P are distorted due to measurement errors. There might also be outliers due to extraneous causes (e.g. through metallic objects) that distort the electro-magnetic field. We use the following heuristic to cope with these problems. After computing the smallest enclosing annulus of P, we check its width, i.e. the difference of R and r, cf. Section 3.3. If the annulus is wider than a predefined bound, we remove points from P that are 'most extreme' with respect to the current annulus. By 'most extreme' we mean those points, whose distance to the annulus' center deviates most from the average distance to the center of all points in P. Then the smallest enclosing annulus is recomputed. This step is repeated, until the width of the resulting annulus is smaller than the predefined bound. We end up with the following algorithm ($\textsc{ma}(P)$ denotes the smallest enclosing annulus of P):

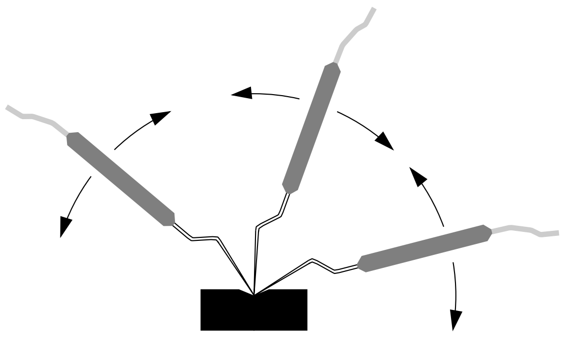


Figure 9.2: *Calibration setup*

Algorithm 9.1 *(returns the position of the stylus' tip)*

```
StylusCalibration($P$):
    $a := \text{MA}(P)$
    WHILE width($a$) $>$ bound DO
        $P := P \setminus \{p \in P \mid p$ is a 'most extreme' outlier$\}$
        $a := \text{MA}(P)$
    END
    RETURN center($a$)
```

9.2.4 Experimental Results

The navigation system has two kind of styli. The difference lies in the length of the pointer, resulting in different length of the offset vectors. The short version has an offset of about 12cm, the long version of about 19cm.

For each stylus to calibrate, we acquired between 5,000 and 10,000 position samples. Usually one or two iterations in Algorithm 9.1 were sufficient to obtain an annulus of width smaller than 2mm (short version) and 3mm (long version). The annulus' center was then used to compute the offset vector for each point in the final set P. Averaging over all these vectors gave the desired offset vector for the stylus.

In a postprocessing step, we checked the quality of the calibration. For each point in the final P, the position of the tip was computed using the now available offset vector of the stylus. These positions were compared to the 'real' position of the tip, namely the center of the final annulus. The maximal deviations of 1.2mm for the short styli, and 1.8mm for the long version, were perfectly satisfying for this application.

Bibliography

[1] I. Adler and R. Shamir. A randomized scheme for speeding up algorithms for linear and convex programming with high constraints-to-variable ratio. *Math. Programming*, 61:39–52, 1993.

[2] P. K. Agarwal, B. Aronov, S. Har-Peled, and M. Sharir. Approximation and exact algorithms for minimum-width annuli and shells. *Discrete Computational Geometry*, 24(4):687–705, 2000.

[3] N. Amenta. Computational geometry software. In *Handbook of Discrete and Computational Geometry*, pages 951–960. CRC Press, 1997.

[4] J. Arvo, editor. *Graphics Gems II.* The graphics gems series. Academic Press, 1991.

[5] D. Avis. A `C` implementation of the reverse search vertex enumeration algorithm. `ftp://mutt.cs.mcgill.ca/pub/C`.

[6] F. Avnaim. *C++GAL: A C++ Library for Geometric Algorithms.* INRIA Sophia-Antipolis, 1994.

[7] J. E. Baker, R. Tamassia, and L. Vismara. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).

[8] S. Barbey and P. Schiltknecht. Portfolio Optimierung mit Markowitz und Conditional Value-at-Risk. Diploma thesis, ETH Zurich, 2001.

[9] V. Barnett. The ordering of multivariate data. *J. Roy. Statist. Soc.*, 139:318–354, 1976.

[10] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1997.

[11] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, 1995.

[12] R. E. Bixby, J. W. Gregory, I. J. Lustig, R. E. Marsten, and D. F. Shanno. Very large-scale linear programming: a case study in combining interior point and simplex methods. *Operations Research*, 40(5):885–897, 1992.

[13] G. Booch and M. Vilot. Simplifying the Booch components. In S. Lippman, editor, *C++ Gems*, pages 59–89. SIGS publications, 1996.

[14] A. Boriçi and H.-J. Lüthi. Multi-period portfolio optimization by conditional value-at-risk: a proposal. (preprint), 2001.

[15] C. Bouville. Bounding ellipsoids for ray-fractal intersection. In *SIGGRAPH*, pages 45–52, 1985.

[16] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1–2):25–47, 2001.

[17] 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.

[18] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.

[19] *CGAL Reference Manual*, 2001. Ver. 2.3. `http://www.cgal.org/`.

[20] V. Chvátal. *Linear Programming*. W. H. Freeman, New York, NY, 1983.

[21] K. L. Clarkson. A Las Vegas algorithm for linear programming when the dimension is small. *J. ACM*, 42(2):488–499, 1995.

[22] L. Danzer, D. Laugwitz, and H. Lenz. Über das Löwnersche Ellipsoid und sein Analogon unter den einem Eikörper eingeschriebenen Ellipsoiden. *Arch. Math.*, 8:214–219, 1957.

[23] A. C. Davison, C. G. Diderich, and V. Nicoud. Analytical multi-period portfolio optimization. (preprint), February 2000.

[24] P. de Rezende and W. Jacometti. Geolab: an environment for development of algorithms in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 175–180, 1993.

[25] T. D. DeRose. Geometric programming: A coordinate-free approach. In *Theory and Practice of Geometric Modeling*, pages 291–303, Blaubeuren, FRG (Oct 1988), 1989. Springer-Verlag.

[26] M. E. Dyer. A class of convex programs with applications to computational geometry. In *Proc. 8th Annu. ACM Symp. on Computational Geometry*, pages 9–15, 1992.

[27] B. C. Eaves and R. M. Freund. Optimal scaling of balls and polyhedra. *Math. Programming*, 23:138–147, 1982.

[28] J. Edmonds and J. Maurras. Note sur les Q-matrices d'Edmonds. *Rech. Opér (RAIRO)*, 31(2):203–209, 1997.

[29] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11:404–428, 1994.

[30] 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. ACM Workshop on Applied Computational Geometry*, volume 1148 of *Lecture Notes in Computer Science*, pages 191–202. Springer-Verlag, 1996.

[31] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30:1167–1202, 2000.

[32] G. Fischer. *Lineare Algebra – Eine Einführung für Studienanfänger.* 12., verbesserte Auflage. Vieweg, Braunschweig, Germany, 2000.

[33] J. M. Fitzpatrick, J. B. West, and C. R. Maurer. Predicting error in rigid-body point-based registration. *IEEE Transactions on Medical Imaging*, 17(5):694–702, 1998.

[34] G. Forsythe and C. Moler. *Computer Solutions of Linear Algebra Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1967.

[35] R. M. Freund and J. B. Orlin. On the complexity of four polyhedral set containment problems. *Math. Programming*, 33:139–145, 1985.

[36] K. Fukuda. `cdd+` reference manual. `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html`.

[37] B. Gärtner. *Randomized Optimization by Simplex-Type Methods*. PhD thesis, Freie Universität Berlin, 1995.

[38] B. Gärtner. Geometric optimization. Equinoctial School on *Geometry and Computation*, ETH Zurich, 1997.

[39] B. Gärtner. Exact arithmetic at low cost – a case study in linear programming. *Computational Geometry - Theory and Applications*, 13:121–139, 1999.

[40] B. Gärtner. Fast and robust smallest enclosing balls. In *Proc. 7th Annu. European Sympos. Algorithms*, volume 1643 of *Lecture Notes in Computer Science*, pages 325–338. Springer-Verlag, 1999.

[41] B. Gärtner. Pitfalls in computing with pseudorandom determinants. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 148–155, 2000.

[42] B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. *Information Processing Letters*, 68(1):33–38, 1998.

[43] B. Gärtner and S. Schönherr. Smallest enclosing circles – an exact and generic implementation in C++. Technical Report B 98-04, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 1998.

[44] B. Gärtner and S. Schönherr. Smallest enclosing ellipses – an exact and generic implementation in C++. Technical Report B 98-05, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 1998.

[45] B. Gärtner and S. Schönherr. An efficient, exact, and generic quadratic programming solver for geometric optimization. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 110–118, 2000.

[46] B. Gärtner and E. Welzl. Linear programming – randomization and abstract frameworks. In *Proc. 13th Annu. Symp. on Theoretical Aspects of Computer Science (STACS)*, volume 1046 of *Lecture Notes in Computer Science*, pages 669–687. Springer-Verlag, 1996.

[47] B. Gärtner and E. Welzl. A simple sampling lemma: Analysis and applications in geometric optimization. *Discrete Computational Geometry*, 25:569–590, 2001.

[48] G.-J. Giezeman. *PlaGeo, a Library for Planar Geometry and SpaGeo, a Library for Spatial Geometry.* Utrecht University, 1994.

[49] A. S. Glassner, editor. *Graphics Gems.* The graphics gems series. Academic Press, 1990.

[50] R. N. Goldman. Illicit expressions in vector algebra. *ACM Transaction on Graphics*, 4(3):223–243, 1985.

[51] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, October 2000. Version 3.1.1. `http://swox.com/gmp/`.

[52] P. S. Heckbert, editor. *Graphics Gems IV.* The graphics gems series. Academic Press, 1994.

[53] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th Workshop on Algorithms Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 76–91. Springer-Verlag, 2001.

[54] F. Hoffmann, K. Kriegel, S. Schönherr, and C. Wenk. A simple and robust geometric algorithm for landmark registration in computer assisted neurosurgery. Technical Report B 99-21, Freie Universität Berlin, Fachbereich Mathematik und Informatik, 1999.

[55] IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754, 1995. `http://standards.ieee.org/`.

[56] F. John. Extremum problems with inequalities as subsidiary conditions. In *Studies and Essays presented to R. Courant on his 60th Birthday*, pages 187–204. Interscience Publishers, NY, 1948.

[57] T. Keffer. The design and architecture of tools.h++. In S. Lippman, editor, *C++ Gems*, pages 43–57. SIGS publications, 1996.

[58] L. Kettner. *Software Design in Computational Geometry and Contour-Edge Based Polyhedron Visualization.* PhD thesis, ETH Zurich, 1999. Diss. ETH No. 13325.

[59] D. Kirk, editor. *Graphics Gems III.* The graphics gems series. Academic Press, 1992.

[60] R. Kumar and D. Sivakumar. Roundness estimation via random sampling. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 603–612, 1999.

[61] J. Lakos. *Large Scale C++ Software Design*. Addison-Wesley, 1996.

[62] K. Leichtweiß. Über die affine Exzentrizität konvexer Körper. *Arch. Math.*, 10:187–199, 1959.

[63] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[64] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

[65] H. Mausser and D. Rosen. Managing risk with expected shortfall. In S. P. Uryasev, editor, *Probabilistic Constrained Optimization: Methodology and Applications*, pages 204–225. Kluwer Academic Publishers, 2000.

[66] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231, North-Holland, Amsterdam, 1994. Elsevier Science B.V.

[67] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[68] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual*, 2001. Version 4.3.

[69] S. Meyers. *Effective C++*. Addison Wesley, 1992.

[70] S. Meyers. *More Effective C++*. Addison Wesley, 1996.

[71] K. G. Murty and Y. Fathi. A critical index algorithm for nearest point problems on simplicial cones. *Math. Programming*, 23:206–215, 1982.

[72] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[73] D. R. Musser and A. A. Stepanov. Generic programming. In *1st Intl. Joint Conf. of ISSAC-88 and AAEC-6*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1989.

[74] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software – Practice and Experience*, 24(7):623–642, 1994.

[75] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 1995.

[76] Y. Nesterov and A. Nemirovskii. *Interior–Point Polynomial Algorithms in Convex Programming*, volume 13 of *SIAM studies in applied mathematics*. Society of Industrial and Applied Mathematics, 1993.

[77] J. Nievergelt, P. Schorn, M. de Lorenzi, C. Ammann, and A. Brüngger. XYZ: a project in experimental geometric computation. In *Computational Geometry — Methods, Algorithms and Applications: Proc. Internat. Workshop Comput. Geom.*, volume 553 of *Lecture Notes in Computer Science*, pages 171–186. Springer-Verlag, 1991.

[78] M. H. Overmars. Designing the computational algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Proc. ACM Workshop on Applied Computational Geometry*, volume 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[79] A. L. Peressini, F. E. Sullivan, and J. J. Uhl. *The Mathematics of Nonlinear Programming*. Undergraduate Texts in Mathematics. Springer-Verlag, 1988.

[80] M. J. Post. Computing minimum spanning ellipses. Technical Report CS-82-16, Department of Computer Science, Brown University, USA, 1982.

[81] M. J. Post. Minimum spanning ellipsoids. In *Proc. 16th Annu. ACM Sympos. Theory Comput.*, pages 108–116, 1984.

[82] L. Pronzato and E. Walter. Minimum-volume ellipsoids containing compact sets: Applications to parameter bounding. *Automatica*, 30(11):1731–1739, 1994.

[83] V. T. Rajan. Optimality of the delaunay triangulation in $\mathbb{R}^d$. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 357–363, 1991.

[84] R. T. Rockafellar and S. Uryasev. Optimization of conditional value-at-risk. *Journal of Risk*, 2(3):21–41, 2000.

[85] S. Schirra. A case study on the cost of geometric computing. In *Proc. ALENEX'99*, volume 1619 of *Lecture Notes in Computer Science*, pages 156–176. Springer-Verlag, 1999.

[86] S. Schirra. Precision and robustness issues in geometric computation. In *Handbook on Computational Geometry*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1999.

[87] A. Schleich, S. Jovanovic, B. Sedlmaier, U. Warschewske, F. Oltmanns, S. Schönherr, W. Oganovsky, B. Ohnesorge, T. Hoell, and H. Scherer. Electromagnetic ENT navigation: the NEN system. In *Proc. 4th Annu. Conf. International Society for Computer Aided Surgery*, 2000.

[88] S. Schönherr. Berechnung kleinster Ellipsoide um Punktemengen. Diploma thesis, Freie Universität Berlin, Germany, 1994.

[89] S. Schönherr. A quadratic programming engine for geometric optimization – part I: the solver. Technical Report 387, ETH Zurich, Institute of Theoretical Computer Science, December 2002.

[90] S. Schönherr. A quadratic programming engine for geometric optimization – part II: the basis inverse. Technical Report 388, ETH Zurich, Institute of Theoretical Computer Science, December 2002.

[91] S. Schönherr. A quadratic programming engine for geometric optimization – part III: pricing strategies. Technical Report 389, ETH Zurich, Institute of Theoretical Computer Science, December 2002.

[92] P. Schorn. Implementing the XYZ GeoBench: a programming environment for geometric algorithms. In *Computational Geometry — Methods, Algorithms and Applications: Proc. Internat. Workshop Comput. Geom.*, volume 553, pages 187–202, 1991.

[93] R. Seidel. Personal communication, 1997.

[94] K. Sekitani and Y. Yamamoto. A recursive algorithm for finding the minimum norm point in a polytope and a pair of closest points in two polytopes. *Math. Programming*, 61(2):233–249, 1993.

[95] N. Z. Shor and O. A. Berezovski. New algorithms for constructing optimal circumscribed and inscribed ellipsoids. *Opt. Methods and Software*, 1:283–299, 1992.

[96] Silicon Graphics Computer Systems. *Standard Template Library Programmer's Guide*, 1997. `http://www.sgi.com/tech/stl/`.

[97] B. W. Silverman and D. M. Titterington. Minimum covering ellipses. *SIAM J. Sci. Statist. Comput.*, 1:401–409, 1980.

[98] B. Spain. *Analytical Conics*. Pergamon Press, 1957.

[99] A. Stepanov and M. Lee. *The Standard Template Library*, October 1995. `http://www.cs.rpi.edu/~musser/doc.ps`.

[100] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[101] D. M. Titterington. Optimal design: some geometrical aspects of D-optimality. *Biometrika*, 62(2):313–320, 1975.

[102] D. M. Titterington. Estimation of correlation coefficients by ellipsoidal trimming. *Appl. Statist.*, 27(3):227–234, 1978.

[103] S. Uryasev. Conditional value-at-risk: Optimization algorithms and applications. *Financial Engineering News*, 14, 2000.

[104] B. Wallis. Forms, vectors, and transforms. In A. S. Glassner, editor, *Graphics Gems*, pages 533–538. Academic Press, 1990.

[105] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[106] P. Wolfe. The simplex method for quadratic programming. *Econometrica*, 27:382–398, 1959.

[107] P. Wolfe. Finding the nearest point in a polytope. *Math. Programming*, 11:128–149, 1976.

Index

Curriculum Vitae

Sven Rolf Schönherr

born on March 17th, 1968, in Berlin, Germany
citizen of Germany

Education

1998 – 2002	Swiss Federal Institute of Technology Zurich / Freie Universität Berlin, Germany Ph.D. Studies in Computer Science Candidate for **Doctor of Technical Sciences**
1987 – 1994	Freie Universität Berlin, Germany / University of Technology Berlin, Germany Studies of Mathematics and Computer Science **Diploma in Mathematics** (FU Berlin, grade: 1.7)
1980 – 1987	Beethoven-Gymnasium (high school), Berlin, Germany **Allgemeine Hochschulreife** (grade: 1.4)

Experience

since 04/2001	Mayfield Technologies, Waltersdorf (Berlin), Germany **Software Developer / Project Leader** Conception, design, implementation, and certification of a complex application for medical navigation systems
2000 – 2001	Swiss Federal Institute of Technology Zurich **Research Assistant / Teaching Assistant** in the group of Prof. Welzl Design and implementation of a software library of geometric data structures and algorithms within the scope of a project funded by the European Community and the Swiss Federal Office for Education and Science
1995 – 1999	Freie Universität Berlin, Germany **Scientific Programmer / Research Assistant** in the group of Prof. Welzl and Prof. Alt

Design and implementation of a software library of geometric data structures and algorithms within the scope of several projects funded by the European Community

1990 – 1992 Freie Universität Berlin, Germany
Student Assistant
in the group of Prof. Welzl
Implementation of geometric and optimization algorithms

1989 – 1994 **Self-employed Software Developer**
Design and realization of several software projects