Robust algorithms in a program library for geometric computation

Author(s):
Schorn, Peter

Publication Date:
1991

Permanent Link:
https://doi.org/10.3929/ethz-a-000604568

Rights / License:
In Copyright - Non-Commercial Use Permitted
Robust Algorithms in a Program Library for Geometric Computation

Peter Schorn
Robust Algorithms in a Program Library for Geometric Computation

A dissertation submitted to the Swiss Federal Institute of Technology Zürich for the degree of Doctor of Technical Sciences

presented by
Peter Schorn
Dipl.-Inf. TU München
Ms. Sc. UNC Chapel Hill
born August 19, 1961
citizen of the Federal Republic of Germany

Accepted on the recommendation of
Prof. Dr. J. Nievergelt, examiner
Prof. Dr. G. Gonnet, co-examiner

Zürich 1991
to my parents

Richard and Sonia Schorn
Acknowledgements

I sincerely thank my advisor J. Nievergelt for his encouragement and for supporting this project first at the University of North Carolina at Chapel Hill and then here at ETH Zürich. I am also grateful for the many fruitful discussions we had about geometric algorithms and the problem of consistency which helped to shape my ideas. Furthermore I thank for his constructive criticism and many helpful hints regarding style and presentation of this dissertation.

The following people also deserve special mention:

• **Anders Kierulf** for constructive discussions and his help with the Macintosh Toolbox.
• **Gaston Gonnet** for his willingness to be co-referee and his useful suggestions to the dissertation.
• **Christoph Ammann** and **Michele De Lorenzi** for helping in the evolution of the XYZ GeoBench.
• **Martin Müller** for proofreading and spotting some mistakes.
• **Ralph Gasser** for his careful reading and commenting on this dissertation.
• **Adrian Brüngger, Beat Fawer, Markus Furter, Peter Lippuner, Martin Müller** and **Peter Skrotzky** for contributing implementations of geometric algorithms to the library.
• **Henry Greenside, David Plaisted, Don Stanat** and **Turner Whitted** for their willingness to serve on my doctoral committee in Chapel Hill.

and last but not least

• **Klaus Hinrichs** for introducing me to geometric computation, productive discussions about geometric algorithms and valuable comments on an earlier draft of this dissertation.
# Table of contents

Abstract .................................................................................................................. 8  
Kurzfassung ............................................................................................................. 8  

Preface ...................................................................................................................... 9  

1 How to produce robust software for geometric computation? .................. 10  
1.1 The difficulty of implementing geometric algorithms ......................... 10  
1.2 Requirements and consequences ............................................................. 11  
1.3 Related efforts and how they differ from our approach ....................... 11  
1.4 Results and contributions ......................................................................... 12  

2 Mathematical techniques for robust geometric programs ....................... 13  
2.1 Analytic geometry of points, line segments, bisectors and circles ........ 13  
2.2 Models of arithmetic and their important properties ........................... 15  
2.3 The axiomatic approach ........................................................................... 18  
2.4 Explicit detection of nearly degenerate configurations ......................... 20  
2.5 A topological argument for decreasing the required precision ............ 27  
2.6 Evaluation of the methods ........................................................................ 30  

3 Provably accurate programs for basic geometric problems ..................... 31  
3.1 Principles of plane sweep ......................................................................... 31  
3.2 The closest-pair problem .......................................................................... 32  
3.3 The all-nearest-neighbors problem .......................................................... 38  
3.4 The line segment intersection test in floating point arithmetic ............. 42  
3.5 Finding all intersections of line segments in the plane ......................... 45  

4 The XYZ GeoBench: A workbench for geometric computation .............. 51  
4.1 Goals of the project .................................................................................. 51  
4.2 System structure and class hierarchy ..................................................... 51  
4.3 Universal operations ................................................................................ 53  
4.4 Interchangeable arithmetic and parameterized floating point arithmetic 59  
4.5 Geometric primitives .............................................................................. 60  
4.6 Abstract data types ................................................................................ 61  
4.7 User interface and algorithm animation ................................................. 67  
4.8 The object oriented approach: Experience and evaluation ................. 69  

5 The XYZ Library: Algorithms and experimental results ......................... 73  
5.1 The XYZ Library ...................................................................................... 73  
5.2 Experimental results .............................................................................. 74  

6 Conclusion ......................................................................................................... 88  

Appendix: Syntax of the textual I/O format ...................................................... 89  

References ............................................................................................................ 91
Abstract
Computational geometry, dealing with the efficient algorithmic solution of geometric problems, has developed many algorithms and techniques. This dissertation deals with the question of how to produce good software for geometric computation.

Using 2-dimensional problems, we present three novel techniques for coping with the problem of robustness, i.e. how to create algorithms that correctly process even degenerate input data. In the axiomatic approach we implement the geometric primitives using rounded arithmetic such that the essential properties needed to maintain an algorithm's invariant are preserved. In the method of explicitly detecting nearly degenerate configurations we treat those configurations as exact degeneracies, often resulting in a provably robust implementation. A topological argument reduces the precision requirements for an exact result in certain plane sweep algorithms.

We describe the design and implementation of the XYZ GeoBench (eXperimental geometrY Zurich), a workbench for geometric computation. The GeoBench is a programming environment, implemented in an object oriented language, for the rapid prototyping of geometric software and a testbed for experiments. Algorithm animation is used for demonstration purposes and debugging.

As an answer to the problem of robust geometric software we offer the XYZ Library which is based on the GeoBench and has been tested in various experiments.

Kurzfassung


Als Antwort auf die Frage nach robusten Programmen für das geometrische Rechnen bieten wir die XYZ Bibliothek an. Sie basiert auf der GeoBench und wurde in zahlreichen Experimenten getestet.
Preface

Computational geometry deals with the efficient algorithmic solution of geometric problems. This relatively young branch of computer science has prospered in recent years, witnessed by a continuing growth of publications. Computational geometry has traditionally focused on complexity theory, often neglecting the more practical aspects of geometric computation.

In this dissertation we focus on the question how to produce good software for geometric computation and we identify two essential issues: Mathematical techniques for the creation of robust algorithms and a workbench for the implementation of geometric algorithms. The following diagram presents the structure of the dissertation.

![Diagram: Structure of the dissertation]

Chapter 1 motivates our research. Chapter 2 introduces three mathematical techniques aimed at creating robust geometric programs using simple examples. Chapter 3 applies these techniques to more advanced examples in 2-d geometric computation. Chapter 4 describes the goals, the design and implementation of the XYZ GeoBench (eXperimental geometrY Zurich). Chapter 5 describes the XYZ Library and some of the experiments performed with its algorithms. Chapter 6 gives a brief summary and concludes with future directions.
1 How to produce robust software for geometric computation?

1.1 The difficulty of implementing geometric algorithms

Anyone who tries to solve a problem in geometric computation encounters numerous difficulties. This is due to:

Use of non trivial data structures. Most geometric algorithms rely on data structures such as dictionaries and priority queues. These data structures need to be general and efficient. General, because they must be useful for many different algorithms; efficient, because the efficiency of the whole algorithm depends on them.

Lack of infrastructure. A geometric program usually does not exist by itself. One needs access to a system that can manipulate geometric objects in various ways, e.g. by offering means for displaying, storing, retrieving and interactively editing them. It takes considerable effort to implement an environment useful for testing and program development.

Different algorithms for the same problem. Usually there are different algorithms that solve the same problem, and experiments are needed to identify a practical algorithm. Additional factors such as ease of implementation or memory requirements influence this decision.

Degenerate configurations and robustness. For the sake of simplicity many geometric algorithms assume the input data in general position, that is, only trivial relationships between the geometric objects are allowed to hold. For example, three or more collinear points are not in general position. A library routine must be robust, meaning that it must be able to produce a correct result for all configurations. Degenerate configurations occur in practice and sometimes are even favored by applications. As an example, aligned (axis parallel) polygons are a degenerate case for certain polygon intersection algorithms. But they are standard in VLSI design, architecture and other applications. The correct treatment of degenerate cases is difficult; we present methods on how to approach the problem.

Imprecise primitives and consistency. Geometric algorithms are usually based on geometric primitives like testing whether a given point lies to the left, to the right or on a directed line. A problem arises when a primitive is implemented using rounded arithmetic. Then the decision of such a procedure implemented with imprecise arithmetic can be different from the decision of the same procedure implemented with exact arithmetic. This alone would not pose a problem if one could enforce that subsequent decisions are consistent with this first decision. Otherwise a program might enter an illegal state. Even if consistency could be guaranteed, which is often more difficult than using a higher precision, the question remains how to assess the result’s accuracy. We will see techniques that address the problems of consistency and accuracy.

Hoffmann surveys the problems of accuracy, degeneracy and robustness in geometric computation in [H 89].
1.2 Requirements and consequences

The difficulties presented in the previous section fall into two broad categories: 1) Robustness and numerical difficulties in general, and 2) Software problems. They call for:

1) **Methods for constructing robust algorithms.** Computational geometry has addressed the problem of robustness by proposing a variety of methods. We introduce three new methods in section 2 mostly because we believe that the known methods are sometimes too restrictive or too expensive. For example, Edelsbrunner's and Mücke's Simulation of Simplicity [EM 90] requires (long) integer arithmetic, which can be expensive if high precision is required.

2) **A workbench for the rapid prototyping of geometric programs.** This workbench should provide the necessary infrastructure for geometric computation: A collection of sufficiently general abstract data types, geometric primitives, (interactive) I/O, universal operations like memory management and geometric transformations, algorithm animation for demonstration and testing, support for test data generation and experiments, etc. Section 4 describes the design and implementation of such a workbench, the XYZ GeoBench.

3) **A library for geometric computation.** Implementing a geometric algorithm is a difficult task. Doing it only once saves other people a considerable effort. This is the motivation found behind almost all program libraries. The ideas of reusability and sharing implementation effort are also driving forces behind the concepts of Abstract Data Types and Object Oriented Programming. The library should be extensible and based on the workbench proposed in the previous paragraph. Section 5 describes the XYZ Library, a basic library for geometric computation.

1.3 Related efforts and how they differ from our approach

The project LEDA [MN 89] is similar to our efforts and also focuses on efficient reusable software. LEDA contains many graph algorithms and a small set of geometric algorithms. In contrast to LEDA we concentrate our efforts on geometric computation, emphasizing robustness and accuracy. In addition, we provide an interactive system with algorithm animation for experimental geometric computation, whereas LEDA supports only a programming interface.

The second project we are aware of is the Workbench for Computational Geometry (WOCG) developed at Carleton University in Ottawa [ES 90]. Its goals are similar to ours but with less emphasis on efficiency and robustness. The WOCG contains some complicated algorithms in order to demonstrate the feasibility of their implementation. For practical cases, though, one would use simpler algorithms that are inferior in the theoretical sense, but faster most of the time.

There are several investigations on robust algorithms. We discuss their relation to our approaches in chapter 2 when we describe our methods.
1.4 Results and contributions

To the best of my knowledge these are the most significant and novel results of this dissertation.

a) *Three methods for the creation of robust geometric programs.* We provide some solutions for the problem of designing geometric algorithms that deliver provably accurate results even when implemented in rounded arithmetic (floating point arithmetic or integer arithmetic with the truncating division operator 'div').

- In the **axiomatic approach** we implement the geometric primitives using rounded arithmetic such that the essential properties needed to maintain an algorithm’s invariant are preserved. The accuracy of the computed result is then proved using tools from numerical analysis.

- In the second approach, we *explicitly detect nearly degenerate configurations* and treat them either as failure or preferably as exact degenerate configurations, often resulting in a provably correct implementation.

- Finally we give a *topological argument* that is used to reduce the arithmetical precision requirements in plane sweep algorithms.

We apply the described techniques to some simple problems which are still interesting by themselves and give detailed analysis in a separate chapter for the closest-pair problem, the all-nearest-neighbors problem and the problem of finding all pairs of intersecting line segments. All our examples of geometric problems are taken from 2-d geometric computation.

b) *The design and implementation of a workbench for geometric computation.* The XYZ GeoBench is a programming environment and testbed for geometric algorithms featuring algorithm animation for teaching and debugging. We justify its implementation in an object oriented language and cite various benefits.

c) *Implementation and experimental verification of a basic library for geometric computation.* On top of the GeoBench we have implemented the XYZ Library, a basic library for geometric computation. We present numerous experimental results.
2 Mathematical techniques for robust geometric programs

This section discusses some practical methods for constructing geometric programs with provable robustness properties. When describing geometric algorithms we usually assume arithmetic on the real numbers with infinite precision, a requirement rarely met in practice. This forces the algorithm designer to construct algorithms which can be implemented using imprecise arithmetic, such as floating point arithmetic. In order to make provable claims about the accuracy of the computed result, we introduce three techniques for constructing robust programs. We give some basic analytic geometry necessary for the algorithms and introduce models of arithmetic before we present our three methods for robust programs.

2.1 Analytic geometry of points, line segments, bisectors and circles

In this section we collect the most useful definitions, formulas and facts from the analytic geometry of simple geometric objects.

Definition 1 (basic notation): Let p, q and r be points in the plane. Then we define

- the lexicographical order: \( p <_{\text{lex}} q \iff (p_x < q_x) \vee (p_x = q_x \land p_y < q_y) \)
- the directed line from p to q: \( l(p, q) \) for \( p \neq q \).
- the directed line segment from p to q: \( ls(p, q) \) for \( p \neq q \).
- the Euclidean distance between p and q: \( d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \)
- the bisector of p and q: \( bs(p, q) = \{ t : d(p, t) = d(q, t) \} \) for \( p \neq q \).
- the determinant \( \Delta \), also called 'whichSide':
  \[
  \Delta(p, q, r) = (p_x - q_x)(q_y - r_y) - (p_y - q_y)(q_x - r_x)
  \]
  The sign of \( \Delta(p, q, r) \) determines on which side of \( l(p, r) \) the point q lies.

Lemma 1 (determining on which side a point lies relative to a directed line): Given three points \( p, q \) and \( r \) in the plane. Then

\[
\Delta(p, q, r) = \begin{vmatrix}
  p_x & p_y & 1 \\
  q_x & q_y & 1 \\
  r_x & r_y & 1
\end{vmatrix}
\]

and

- \( \Delta(p, q, r) < 0 \iff q \) lies to the left of \( l(p, r) \)
- \( \Delta(p, q, r) = 0 \iff q \) lies on \( l(p, r) \)
- \( \Delta(p, q, r) > 0 \iff q \) lies to the right of \( l(p, r) \)

and \( \Delta(p, q, r) \) is twice the signed area of the triangle with vertices \( p, q \) and \( r \).

Lemma 2 (intersection of two line segments): Given four points \( p, q, r \) and \( s \) in the plane such that \( p <_{\text{lex}} q \) and \( r <_{\text{lex}} s \). Then the algorithm below yields the intersection of the two straight line segments \( ls(p, q) \) and \( ls(r, s) \) or \( \emptyset \) if none exists. We use the conventions that \( ls(u, v) = \emptyset \) if \( v <_{\text{lex}} u \) and that \( ls(u, u) = u \) and define

\[
\text{sign}(x) = \begin{cases}
  +1, & \text{if } x > 0 \\
  0, & \text{if } x = 0 \\
  -1, & \text{if } x < 0
\end{cases}
\]
if \((\Delta(p, r, q) = 0) \land (\Delta(p, s, q) = 0)\) 
then \(ls(\max(p, r), \min(q, s))\)
elsif \((\text{sign}(\Delta(p, r, q)) \neq \text{sign}(\Delta(p, s, q))) \land (\text{sign}(\Delta(r, p, s)) \neq \text{sign}(\Delta(r, q, s)))\)
then \(\frac{\Delta(p, r, q) \cdot s_x - \Delta(p, s, q) \cdot r_x}{\Delta(p, r, q) - \Delta(p, s, q)} = \frac{\Delta(p, r, q) \cdot s_y - \Delta(p, s, q) \cdot r_y}{\Delta(p, r, q) - \Delta(p, s, q)}\)
else \(\emptyset\)
end;

Proof: In parameter form \(ls(p, q)\) is given by \((p_x + \alpha \cdot (q_x - p_x), p_y + \alpha \cdot (q_y - p_y)), 0 \leq \alpha \leq 1,\) and \(ls(r, s)\) is given by \((r_x + \beta \cdot (s_x - r_x), r_y + \beta \cdot (s_y - r_y)), 0 \leq \beta \leq 1.\) We distinguish two cases: The segments are parallel or not. In the former case the intersection is non empty if and only if they overlap, i.e. \(\Delta(p, r, q) = \Delta(p, s, q) = 0 \land \max(p, r) \leq \min(q, s).\) In the latter case of non parallel segments we solve the linear system for \(\alpha\) and \(\beta\) and compute the unique intersection point of the lines through \(p\) and \(q\) and through \(r\) and \(s.\) We obtain
\[
\alpha = \frac{\Delta(r, p, s)}{\Delta(r, p, s) - \Delta(r, q, s)}, \quad \beta = \frac{\Delta(p, r, q)}{\Delta(p, r, q) - \Delta(p, s, q)}
\]
\[(q - p) \times (s - r) = \begin{vmatrix}
q_x - p_x & s_x - r_x \\
q_y - p_y & s_y - r_y
\end{vmatrix} = \Delta(p, r, q) - \Delta(p, s, q) = -(\Delta(r, p, s) - \Delta(r, q, s))
\]
The intersection point lies on both segments if and only if
\[
(\text{sign}(\Delta(p, r, q)) \neq \text{sign}(\Delta(p, s, q))) \land (\text{sign}(\Delta(r, p, s)) \neq \text{sign}(\Delta(r, q, s))
\]
since this condition is equivalent to \(0 \leq \alpha \leq 1 \land 0 \leq \beta \leq 1.\) Backsubstituting either \(\alpha\) or \(\beta\) into the appropriate parameter form yields the coordinates of the intersection point.

Lemma 3 (equation of a bisector): Let \(p\) and \(q, p \neq q,\) be two points in the plane. Then the equation of the bisector \(bs(p, q)\) is given by \(ax + by = c\) with \(a = 2 \cdot (p_x - q_x), b = 2 \cdot (p_y - q_y)\) and \(c = (p_x - q_x) \cdot (p_x + q_x) + (p_y - q_y) \cdot (p_y + q_y) = p_x^2 - q_x^2 + p_y^2 - q_y^2.\)

Proof: This follows immediately from the fact that the point \(1/2 \cdot (p_x + q_x, p_y + q_y)\) lies on the bisector of \(p\) and \(q\) and that the bisector is perpendicular to the line through \(p\) and \(q.\)

Lemma 4 (center and radius of the circle defined by three points): Given three non-collinear points \(p, q\) and \(r\) in the plane (i.e. \(\Delta(p, q, r) \neq 0).\) Then the center of the circle through \(p, q\) and \(r\) is given by
\[
- \frac{1}{2 \cdot \Delta(p, q, r)} \begin{vmatrix}
p_x^2 + p_y^2 & p_x & 1 \\
q_x^2 + q_y^2 & q_x & 1 \\
r_x^2 + r_y^2 & r_x & 1
\end{vmatrix} = \begin{vmatrix}
p_x^2 + p_y^2 & p_x & 1 \\
q_x^2 + q_y^2 & q_x & 1 \\
r_x^2 + r_y^2 & r_x & 1
\end{vmatrix}
\]
Proof: The formula is verified by solving the following system for \( x \) and \( y \), which is a system of linear equations because \( x^2 \) and \( y^2 \) cancel out.

\[
\begin{align*}
(x - p_x)^2 + (y - p_y)^2 &= (x - r_x)^2 + (y - r_y)^2 \\
(x - q_x)^2 + (y - q_y)^2 &= (x - r_x)^2 + (y - r_y)^2
\end{align*}
\]

Lemma 5 (bounded determinant): Given three points \( p, q \) and \( r \), contained in the rectangle given by its opposite corners \((x_{\text{min}}, y_{\text{min}})\) and \((x_{\text{max}}, y_{\text{max}})\), that is:

\[
\begin{align*}
x_{\text{min}} &\leq \min(p_x, q_x, r_x) \leq \max(p_x, q_x, r_x) \leq x_{\text{max}} \quad \text{and} \\
y_{\text{min}} &\leq \min(p_y, q_y, r_y) \leq \max(p_y, q_y, r_y) \leq y_{\text{max}}.
\end{align*}
\]

then \( |\Delta(p, q, r)| \leq (x_{\text{max}} - x_{\text{min}}) \cdot (y_{\text{max}} - y_{\text{min}}) \).

Proof: \( |\Delta(p, q, r)| \) is twice the area of the triangle with vertices \((p_x, p_y), (q_x, q_y), (r_x, r_y)\). The largest triangle which can be inscribed in this rectangle, e.g. the triangle below the diagonal, has half the area of the rectangle. Therefore: \( |\Delta(p, q, r)| \leq \text{area of the rectangle} = (x_{\text{max}} - x_{\text{min}}) \cdot (y_{\text{max}} - y_{\text{min}}) \).

2.2 Models of arithmetic and their important properties

This section discusses various models of arithmetic on a computer and their properties relevant to the implementation of geometric algorithms. In all subsequent sections we denote the set of integers by \( \mathbb{Z} \), the set of rational numbers by \( \mathbb{Q} \) and the set of real numbers by \( \mathbb{R} \).

2.2.1 Integer arithmetic

Integer arithmetic is the most common form of arithmetic, found virtually on any computer. As an advantage, addition, subtraction and multiplication yield an exact result, as long as the magnitudes of the operands do not exceed certain, machine dependent, limits.

Most geometric algorithms perform only computations where the depth of the involved arithmetic expression trees is bounded and independent of the size of the input data. Therefore, an implementation based on integer arithmetic can be exact if 1) the input data lies on a bounded integer grid, and 2) intermediate expressions are still integers and do not generate overflow. We avoid operators that would take us outside of the integers by reformulating geometric primitives. For example, consider the predicate \( \text{isCloser}(p, q, r) \) that decides whether point \( p \) is closer to point \( q \) or whether \( p \) is closer to point \( r \) under the Euclidean metric. Comparing the squares of the distances instead of the distances themselves avoids the use of the expensive square root operation.

The notion that an algorithm can be implemented in \( k \)-fold precision integer arithmetic means that if the input data can be represented with \( m \)-bit binary numbers, the algorithm generates intermediate expressions that can be represented as \((k \cdot m + O(1))\)-bit binary numbers. Typically, the constant in the \( O(1) \) term is small, and \( k = 3 \) or \( k = 5 \).
A difficulty arises with the division operation since in general dividing two integers does not yield an integer. The truncated division operator ‘div’ defined as \( x \div y = \lfloor x / y \rfloor \), where \( \lfloor . \rfloor \) denotes the floor function and ‘/’ real division, introduces an error. In sections 3.3 and 3.5 we present examples based on a method where ‘div’ can nevertheless be used as a substitute for ‘/’.

2.2.2 Rational arithmetic

Rational arithmetic generalizes the range of applications of integer arithmetic by allowing the division operation to be performed error free. Rational arithmetic represents a number \( n/d, d \neq 0 \), as a pair \((n, d)\), where \( n \) and \( d \) are integers. Arithmetic on such pairs is a straightforward implementation of the rules for ordinary fractions and can be done in any integer arithmetic with sufficiently high precision. As a drawback we note the growth of numerator and denominator caused by the operators ‘+’ and ‘-’.

Unfortunately, both integer and rational arithmetic lack expressive power with respect to geometry. The following lemma shows that an object as simple as an equilateral triangle cannot be specified using vertices with rational coordinate values.

Lemma 6: The vertices of an equilateral triangle cannot have rational coordinate values.

Proof: Let \( p, q \) and \( r \) be the vertices of an equilateral triangle with \( q \neq r \). We show that \( \tan(\angle pqr) \) is a rational function of the coordinates of the vertices. This implies the claim since \( \tan(60^\circ) = 3^{1/2} \) is irrational.

\[
\tan(\alpha) = \frac{2 \cdot \mathbf{h} \cdot d(q, r)}{d(q, r)^2} = \frac{2 \cdot (p_x - q_x) \cdot (q_y - r_y) - (p_y - q_y) \cdot (q_x - r_x)}{(q_x - r_x)^2 + (q_y - r_y)^2}
\]

2.2.3 Floating point arithmetic

Floating point arithmetic is the most common form of arithmetic found in scientific computation. Its main advantage is the ability to perform arithmetic on numbers that vary greatly in magnitude without having to worry about the necessary scaling.

Definition 2 (floating point arithmetic): For \( b \in \mathbb{Z}, b > 1, p \in \mathbb{Z}, p > 1 \), we define:

a) the set of normalized \( p \)-digit floating point numbers in base \( b \):
\[
F_{b, p} = \{ s \cdot m \cdot b^e : s \in \{-1, 0, 1\} \land m \in Q \land 1/b \leq m < 1 \land m \cdot b^p \in \mathbb{Z} \land e \in \mathbb{Z} \}.
\]

The number \( s \) is the sign, \( m \) is the mantissa and \( e \) the exponent.

b) the machine epsilon: \( \text{eps} = b^{1-p}/2 \)
c) the rounding function round to nearest: round: \( \mathbb{R} \to \mathbb{F}_{b, p} \) with the properties
\[
\text{round}(x) = x \cdot (1 + \varepsilon), \quad \varepsilon < \text{eps} \quad \text{(bounded error)}
\]
\[
\forall x \in \mathbb{R}, y \in \mathbb{R}: x < y \Rightarrow \text{round}(x) \leq \text{round}(y) \quad \text{(monotonicity)}
\]
d) the rounding function round upwards: \( \text{round}^+: \mathbb{R} \to \mathbb{F}_{b, p} \) with the properties
\[
\text{round}^+(x) = x \cdot (1 + e), \quad 0 \leq e < 2 \cdot \text{eps} \quad \text{(bounded error)}
\]
\[
\forall x \in \mathbb{R}, y \in \mathbb{R}: x < y \Rightarrow \text{round}^+(x) \leq \text{round}^+(y) \quad \text{(monotonicity)}
\]
e) the rounding function round downwards: \( \text{round}^\downarrow: \mathbb{R} \to \mathbb{F}_{b, p} \) with the properties
\[
\text{round}^\downarrow(x) = x \cdot (1 - e), \quad 0 \leq e < 2 \cdot \text{eps} \quad \text{(bounded error)}
\]
\[
\forall x \in \mathbb{R}, y \in \mathbb{R}: x < y \Rightarrow \text{round}^\downarrow(x) \leq \text{round}^\downarrow(y) \quad \text{(monotonicity)}
\]
f) three sets of floating point operations: for \( x \in \mathbb{F}_{b, p}, y \in \mathbb{F}_{b, p} \),
\[
x \oplus y = \text{round}(x + y), \quad x^* y = \text{round}(x \cdot y), \quad x/x y = \text{round}(x/y)
\]
\[
x \oplus^\uparrow y = \text{round}^+ (x + y), \quad x \downarrow y = \text{round}^\downarrow (x - y), \quad x \uparrow y = \text{round}^+(x - y)
\]
\[
x \oplus^\downarrow y = \text{round}^\downarrow (x + y), \quad x \downarrow y = \text{round}^\downarrow (x - y), \quad x \uparrow y = \text{round}^+(x - y)
\]

For the existence of the various rounding functions, their monotonicity properties and the bound on the approximation error see any book on numerical analysis (e.g. Pizer [P 83], Stoer and Bulirsch [SB 79]) or the survey paper by Goldberg [Gb 91].

We interpret definition 2 f) in two ways: In the spirit of forward analysis or in the spirit of backward analysis. As an example we use the standard function 'round to nearest' and the addition operator. In forward analysis we deduce from the equality \( x \oplus y = \text{round}(x + y) \) and the bounded error property of 'round to nearest' that the absolute value of the relative error in the floating point addition, namely the quantity \( |(x \oplus y) - (x + y)| / (x + y) | \), is bounded by the machine epsilon \( \varepsilon \). In backward analysis one rewrites \( x \oplus y = \text{round}(x + y) \) as \( x \oplus y = x \cdot (1 + \varepsilon) + y \cdot (1 + \varepsilon) \) with \( \varepsilon < \varepsilon \), and says that floating point addition can be viewed as the exact operation on slightly perturbed data. We use both views.

Although we find 'round to nearest' to be the predominant method of rounding on computers nowadays we note that the rounding mode can be specified in the IEEE floating point standard [FP 81] and the Apple SANE library [Apple].

2.2.4 Other models: Modular arithmetic and interval arithmetic

Standard modular arithmetic represents \( x \in \mathbb{Z} \) as \( (x_1, x_2, \ldots, x_n) \) where \( x_i = x \mod m_i, \quad 1 \leq i \leq n \), for given pairwise prime moduli \( m_i \in \mathbb{Z}, \quad 1 \leq i \leq n \). Given the vector of remainders, one can reconstruct \( x \) using the Chinese Remainder Algorithm [GCL 91]. For \( \mathbb{op} \in \{+,-,\cdot\}, \quad x = (x_1, x_2, \ldots, x_n) \) and \( y = (y_1, y_2, \ldots, y_n) \) the result of \( x \mathbb{op} y \) is the vector \((x_1 \mathbb{op} y_1) \mod m_1, (x_2 \mathbb{op} y_2) \mod m_2, \ldots, (x_n \mathbb{op} y_n) \mod m_n\). Modular arithmetic could be used for implementing high precision integer arithmetic but unfortunately for the operations of truncated division and comparison no simple procedure is known. The fact that comparison is crucial in geometric algorithms throws some doubts on the usability of modular arithmetic in this context. Rosenberger [Ro 90] comes to a similar conclusion although he is more concerned with computing the sign of a determinant of integer entries.

Interval arithmetic represents a number \( x \in \mathbb{R} \) as a pair of floating point numbers \((l, u)\) such that \( l \leq x \leq u \). One hopes that the difference \((u - l)\) is small. Unfortunately, the pessimistic
nature of the arithmetic operations on intervals causes them to become larger and larger as the
computation progresses. This effect is tolerable for computations of limited arithmetical
deepth. We simulate the effect of giving an upper (lower) bound on a computed quantity by
using upward (downward) rounding as described in the previous section.

2.3 The axiomatic approach

The axiomatic approach is a method to build provably accurate programs from imprecise
primitives. The main idea is to preserve only those properties of the primitives which are
needed to prove an algorithm's correctness and accuracy. This set of essential properties is
often surprisingly small and revealed only by careful inspection of the proofs. In the
following we introduce this method using a simple geometric algorithm.

2.3.1 Example: Improving a Traveling Salesman tour

As our working example we consider the problem of improving a given Traveling Salesman
tour using the heuristic suggested by the following figure.

A corresponding program might look as follows:

```plaintext
var t: array [1 .. n] of 1 .. n;   { Current tour, a permutation: t[i] = t[j] \Rightarrow i = j }
c: array [1 .. n, 1 .. n] of number; { Cost matrix: c[i, j] > 0 }

while \exists i, j: isShorter(c[i - 1, i + 1], c[j, i], c[i, j + 1],
c[i - 1, i], c[i, i + 1], c[j, j + 1]) do
  { isShorter(a, b, c, d, e, f) \iff a + b + c < d + e + f }
    'move point i between point j and point j + 1'
end
```

Before we apply the method to this example we introduce some terminology.

2.3.2 Definitions: Program scheme, interpretation and consistency

We distinguish between the structure of a program and the underlying primitives, thus hiding
the model of arithmetic. This leads us to the following definitions of program scheme,
interpretation, and consistency.

Definition 3 (program scheme): A program S is a program scheme S(p_1, p_2, ..., p_k) with
respect to the primitives $p_1, p_2, \ldots, p_k$, if the potentially imprecise operations, like addition, multiplication, etc., occur only inside these primitives. The comparison operator 'Ã—' is considered to be precise, because it satisfies its usual properties even in floating point arithmetic.

**Definition 4 (interpretation):** A program scheme is interpreted by giving concrete implementations of the primitives in a specified model of arithmetic.

**Definition 5 (consistency of an interpretation):** An interpretation of a program scheme is consistent with the correctness proof of the program, if the implementations of the primitives satisfy the properties required for carrying out the proof.

**2.3.3 Application to the example**

In the above example of a program scheme we are primarily concerned with the problem of termination, since implementing 'isShorter($a, b, c, d, e, f$)' directly in floating point arithmetic as $a \oplus b \oplus c < d \oplus e \oplus f$ may lead to an infinite loop as the following figure suggests (recall from definition 2 in section 2.2.3 that '\oplus' denotes floating point addition):

![Flip-flop behavior](image)

The reason is that the associative law does not hold in general for floating point addition and applying the transformation to the current tour does not necessarily reduce its length. The property

$$\text{isShorter}(a, b, c, d, e, f) \Rightarrow a + b + c < d + e + f$$

is sufficient to guarantee termination of the algorithm, because the total length of the tour is reduced each time through the loop and there is only a finite number of tours. We also have seen that interpreting the program scheme with a simple minded floating point implementation is inconsistent with the proof of termination because $a \oplus b \oplus c < d \oplus e \oplus f$ does not imply $a + b + c < d + e + f$. A counter example in base 10, two-digit mantissa floating point arithmetic is $(10 \oplus 0.4) \oplus 0.3 < (0.3 \oplus 0.3) \oplus 10$ $(10 < 11$ although $10.7 > 10.6)$.

**2.3.4 A consistent floating point implementation of 'isShorter'**

In order to give a consistent floating point implementation of 'isShorter' we use downward and upward rounding floating point addition as introduced by definition 2 in section 2.2.3.

**Lemma 7 (consistency of 'isShorter'):** The implementation isShorter($a, b, c, d, e, f$) := $a \uparrow b \uparrow c \downarrow d \downarrow e \downarrow f$ is consistent with the termination proof.

**Proof:** Follows immediately from $x + y \leq x \uparrow y$ and $x \downarrow y \leq x + y$. 

2.3.5 Numerical analysis

The trivially consistent interpretation ‘isShorter(a, b, c, d, e, f) := false’ shows that consistent interpretation alone does not guarantee an accurate result. We must perform some numerical analysis.

\[ \neg \text{isShorter}(a, b, c, d, e, f) \Rightarrow a \oplus b \oplus c \geq d \oplus e \oplus f \Rightarrow \]

\[ (a + b + c)(1 + \epsilon_1 + \epsilon_2) \geq (d + e + f)(1 - \epsilon_1 - \epsilon_2), \quad 0 \leq \epsilon_i < 2 \cdot \text{eps}, \quad 1 \leq i \leq 4, \] since we can omit second order error terms and the quantities \( a, b, c, d, e \) and \( f \) are positive

\[ \frac{a + b + c}{d + e + f} \geq \frac{1 - \epsilon_1 - \epsilon_2}{1 + \epsilon_3 + \epsilon_4} = 1 - 8 \cdot \text{eps} \]

When the algorithm using the floating point implementation in 2.3.4 terminates, we have that

\[ \forall i, j: \neg \text{isShorter}(c[i - 1, i + 1], c[j, i], c[i, j + 1], c[i - 1, i], c[i, i + 1], c[j, j + 1]) \]

which implies, using the above analysis, that

\[ \forall i, j: \frac{c[i - 1, i + 1] + c[j, i] + c[i, j + 1]}{c[i - 1, i] + c[i, i + 1] + c[j, j + 1]} \geq 1 - 8 \cdot \text{eps} \]

Even though there could exist \( i \) and \( j \) such that \( L := c[i - 1, i + 1] + c[j, i] + c[i, j + 1] < c[i - 1, i] + c[i, i + 1] + c[j, j + 1] =: R \), the gain for the total tour length would be small, since the above analysis guarantees that \( L / R \) is greater than a number very close to one. This shows that all substantial improvements have been made.

2.3.6 Conclusion

The axiomatic approach begins with an algorithm and its correctness proof. The algorithm is written in terms of primitives which obey certain laws used in the proof. The method succeeds if we are able to implement the primitives in the arithmetic of our choice, preserving the necessary properties. This we call a consistent interpretation. After a consistent interpretation is found, a numerical analysis must be carried out to assess the accuracy of the result obtained.

The main difficulty in applying this method is the unfortunate fact that geometric primitives implemented in ordinary floating point arithmetic usually obey few laws. Consequently we are limited to simple primitives. The main advantage are the conceptual simplicity and the clean separation between consistency issues and numerical analysis. Section 3.2 shows the application of this method to the closest-pair problem.

2.4 Explicit detection of nearly degenerate configurations

We define the notion of degeneracy and give an almost purely mechanical way of generating
reliable programs which can at least detect and often can handle degenerate configurations.

2.4.1 The notion of degeneracy

The notions of degeneracy and degenerate configuration are central to the problem of implementing geometric algorithms robustly. A configuration is degenerate iff its geometric objects are not in general position, i.e. there exist dependencies between the objects which would almost certainly vanish after a random perturbation of the input data is performed. The classic example of three collinear points shows that a configuration is not degenerate per se, but only with respect to a specific algorithm: Three collinear points form no degenerate configuration when we are looking for a closest pair, but they pose a problem when computing the convex hull. Certain types of algorithms introduce degeneracies of their own: points with equal x-coordinates are often degenerate for plane sweep algorithms.

Random perturbations and rotations of the coordinate system are a theoretical method to avoid degeneracies. The transformation of input data usually decreases the accuracy of the result and does not guarantee a degeneracy-free configuration. Another general approach is Simulation of Simplicity [EM 90] which performs a perturbation symbolically but works only with precise (integer) arithmetic.

In the following we always assume arithmetic computations with a fixed depth, that is every geometric quantity of the input data takes part in a fixed and usually small number of arithmetic operations. This assumption is valid for almost all geometric algorithms and emphasizes their combinatorial nature in contrast to their numerical nature.

When we take a closer look at the conditional expressions of any geometric algorithm we see purely combinatorial tests, e.g. checking the range of an index, and geometric tests, e.g. testing on which side of a directed line a given point lies. We focus on the latter tests because they are the source of degeneracies. The 'which side' test, as an example, decides whether a point q lies to the right of a directed line l. Posing this question as a binary predicate is deceptively simple but ignores a special case. What happens if q lies on l? Do we consider this situation as q being on l's right or not? Similar situations arise with almost all geometric primitives. We formulate the following definition of degeneracy.

**Definition 6 (degeneracy):** Let A be a geometric algorithm such that each conditional statement of A involving geometric input data \( v_1, v_2, \ldots, v_m \) and a geometric primitive p is written in the form

```plaintext
if \( p(v_1, v_2, \ldots, v_m) > 0 \) then \( S_1 \)
elsif \( p(v_1, v_2, \ldots, v_m) < 0 \) then \( S_2 \)
else \( p(v_1, v_2, \ldots, v_m) = 0 \) \( S_3 \)
end;
```

A geometric configuration C is **degenerate** for algorithm A iff one of the primitives p evaluates to zero assuming infinite precision arithmetic.

Note that the comparison operators ‘\(<\)' or ‘\(\geq\)' are no longer allowed in conjunction with geometric primitives. The case of equality must appear explicitly in a 3-way branch.
Applying this definition to the ‘which side’ test could have the following effect: Execute $S_1$ when $q$ lies to the right of $l$, execute $S_2$ when $q$ lies to the left of $l$ and execute $S_3$ when $q$ lies on $l$. Fortunately the handling of degenerate cases if often simple since in many algorithms $S_3 = S_1$ or $S_3 = S_2$, i.e. a degenerate case is handled as a non degenerate one.

2.4.2 Degeneracy and imprecise primitives

This definition of degeneracy relies on the fact that we use infinite precision arithmetic. If we had implemented the geometric primitive $p$ in floating point arithmetic the execution of statement $S_3$ means that the configuration is ‘close’ to a degenerate configuration. In fact we treat configurations which result in a small absolute value for $p$ either as $S_1$, $S_2$ or $S_3$. This may be admissible if done consistently or if it does not matter but a proof of correctness and a bound for the resulting error are often hard to obtain.

Section 2.3 shows that it is possible to give a consistent interpretation of the primitives implemented in floating point arithmetic and to analyze the post condition numerically. This method reaches its limits when the necessary axioms for the primitives cannot be enforced any more by a careful use of floating point arithmetic.

We aim to implement the primitives in such a way that most of the time their decision is the same as if we had used infinite precision arithmetic. The remaining cases are treated the same way as degenerate cases in the infinite precision case. Since all laws for primitives not involving degenerate cases hold, most implementations using floating point arithmetic work most of the time: The decisions of the primitives coincide with the correct decision. This guarantees consistency and accuracy at the same time, shifting the problem to a careful analysis of the degenerate cases.

2.4.3 Transforming from infinite precision arithmetic to floating point arithmetic

The following program transformation produces an implementation using floating point arithmetic that computes either the correct result or indicates failure: Rewrite each conditional statement of the form

\[
\text{if } p(v_1, v_2, \ldots, v_m) > 0 \text{ then } S_1 \\
\text{elsif } p(v_1, v_2, \ldots, v_m) < 0 \text{ then } S_2 \\
\text{else } \{ p(v_1, v_2, \ldots, v_m) = 0 \} \text{ then } S_3 \\
\text{end;}
\]

\[
\text{if } \text{downRound}(p(v_1, v_2, \ldots, v_m)) > 0.0 \text{ then } S_1 \\
\text{elsif } \text{upRound}(p(v_1, v_2, \ldots, v_m)) < 0.0 \text{ then } S_2 \\
\text{else } \text{degenerate} := \text{true} \text{ then } S_3 \\
\text{end;}
\]

We limit ourselves to the case where the arithmetic expression $p$ is composed of variables, constants and the operations ‘+’, ‘-’ and ‘.’. Almost all algorithms we are aware of can be written in this form. The following recursive rules, taken from interval arithmetic, define the
computation of downRound(p) and upRound(p). We abbreviate downRound(p) by \( p_\downarrow \), upRound(p) by \( p_\uparrow \) and similarly specify the rounding direction of the arithmetical operators (see section 2.2.3 for the definition of downward and upward rounding).

\[ p_\downarrow = p_\uparrow = p, \text{ if } p \text{ is a variable or a constant.} \]

\[
\begin{align*}
(p_1 + p_2)_\downarrow &= p_1_\downarrow + p_2_\downarrow, & (p_1 + p_2)_\uparrow &= p_1_\uparrow + p_2_\uparrow \\
(p_1 - p_2)_\downarrow &= p_1_\downarrow - p_2_\uparrow, & (p_1 - p_2)_\uparrow &= p_1_\uparrow - p_2_\downarrow \\
(p_1 \cdot p_2)_\downarrow &= \min (p_1_\downarrow \cdot p_2_\downarrow, p_1_\downarrow \cdot p_2_\uparrow, p_1_\uparrow \cdot p_2_\downarrow, p_1_\uparrow \cdot p_2_\uparrow) \\
(p_1 \cdot p_2)_\uparrow &= \max (p_1_\downarrow \cdot p_2_\downarrow, p_1_\downarrow \cdot p_2_\uparrow, p_1_\uparrow \cdot p_2_\downarrow, p_1_\uparrow \cdot p_2_\uparrow)
\end{align*}
\]

The correctness of this transformation follows from the fact that downRound(x) is a lower bound on the correct value of x and that upRound(x) is an upper bound on the correct value of x. In a successful application of this method the execution of statement S3 is correct if we can bound the resulting error. Usually it is possible to prove that a small perturbation of the input data exists which is truly degenerate. We apply the above method to the computation of the winding number.

2.4.4 Example: The winding number

Computing the winding number is an interesting example that shows how to handle all degenerate cases. We study the accuracy of an implementation with floating point arithmetic.

**Definition 7 (winding number):** Given a polygon \( P \) which is not necessarily simple and a point \( t \) which is not on \( P \)'s perimeter, the winding number \( wn(P, t) \) is the number of complete revolutions in the positive direction an observer in \( t \) has to make in order to follow an object that starts on \( P \)'s perimeter and travels on \( P \)'s perimeter in the positive direction until it returns to its starting point.

2.4.5 The standard algorithm and its degenerate cases

The standard algorithm for computing the winding number works as follows. We choose a directed half ray \( r \) emanating from \( t \) and initialize a variable \( wn \) to 0. For each directed edge of \( P \) that crosses \( r \) from left to right, looking in direction \( r \), we increment the variable \( wn \) and for each directed edge that crosses from right to left we decrement \( wn \). Edges not intersecting \( r \) are ignored. After examining all edges of \( P \) the variable \( wn \) contains the winding number. The choice of the ray \( r \) is arbitrary but a horizontal or vertical ray is efficient.

![Figure 2.3: The halfray method for computing the winding number](image-url)
The following figure shows all degenerate cases.

![Figure 2.4: Degenerate cases in the computation of the winding number](image)

### 2.4.6 Symbolic perturbation

Instead of writing special code for each of the special cases, we borrow the idea of a symbolic perturbation from Simulation of Simplicity [EM 90]. Since we are free in our choice of the half ray, we take a half ray emanating at \( t \) and going to \((t_x, t_y + \varepsilon)\) and from there to \((+\infty, t_y + \varepsilon)\) where \( \varepsilon \) is a symbolic quantity only affecting the borderline cases of comparisons. In order to detect whether a horizontal half ray intersects an arbitrary line segment the primitive operation `whichSide(p, q, r)` suffices. According to section 2.1 lemma 1, `whichSide(p, q, r)` has a positive value if \( q \) lies to the right of the directed line from \( p \) to \( r \), is negative if \( q \) lies to the left of this line and is zero if \( q \) lies on the line. We must distinguish three cases.

#### Figure 2.5: The edge \((u, v)\) crosses from right to left

\[
wn := wn + 1 \iff (\text{whichSide}(u, t, v) < 0) \land (u_y \leq t_y < v_y)
\]

#### Figure 2.6: The edge \((u, v)\) crosses from left to right

\[
wn := wn - 1 \iff (\text{whichSide}(u, t, v) > 0) \land (v_y \leq t_y < u_y)
\]

#### Figure 2.7: The ray emanates on an edge

\[
onBoundary \iff (\text{whichSide}(u, t, v) = 0) \land \\
((u_y = v_y) \land ((u_x \leq t_x \leq v_x) \lor (v_x \leq t_x \leq u_x)) \lor (u_y \neq v_y) \land ((u_y \leq t_y \leq v_y) \lor (v_y \leq t_y \leq u_y)))
\]
2.4.7 A provably correct program for computing the winding number

Tieing the above three cases together we obtain the following program.

```pascal
procedure windingNumber(
  P : array of point; { Vertices of the polygon }
  n : integer; { Number of vertices }
  t : point; { We seek the winding number around t. }
  var on : boolean; { 'true' ⇔ t is on P's boundary }
  var wn : integer); { The winding number }
begin
  wn := 0; u := P[1]; P[n + 1] := u; { Sentinel } on := false;
  for i := 2 to n + 1 do
    v := P[i];
    case sign(whichSide(u, t, v)) of
      -1: if uy < ty then wn := wn + 1 end {D}
      0: on := on v (uy = vy) A ((ux < tx < vx) v (vx < tx < ux))
      1: if vy < ty then wn := wn - 1 end
    end;
  u := v
end;
```

This program is provably correct and simpler than a comparable program in Sedgewick's book on algorithms [Se 83] which only performs the point in polygon test and does not detect whether the point t is on P's boundary or not. Furthermore our version is more efficient since for a configuration where t does not lie on P's boundary we need 8-n subtractions and comparisons and only 2-n multiplications whereas the algorithm in [Se 83] needs between 8-n and 16-n subtractions and 5-n to 10-n multiplications.

2.4.8 An accurate floating point implementation

The formula \( \text{whichSide}(p, q, r) = (p_x - q_x)(q_y - r_y) - (q_x - r_x)(p_y - q_y) \) shows that we can compute an exact result using two-fold precision integer arithmetic if the vertices of polygon \( P \) have integer coordinates. Now we consider the case of floating point arithmetic. Applying a slightly modified version of the program transformation from the previous section to 'windingNumber' we obtain 'windingNumberFloat'.

**Theorem 1** (accurate floating point implementation of the winding number): Let 'windingNumberFloat' be the program obtained from 'windingNumber' by replacing the line

```pascal
  case sign(whichSide(u, t, v)) of
```

by

```pascal
  case SignWhichSideFloat(u, t, v) of
```

where
function SignWhichSideFloat(u, t, v: point): { -1 .. +1 };
begin
    swaps := 0;
    if |t_y| > |u_y| then (u, t) := (t, u); swaps := swaps + 1 end;
    if |t_y| > |v_y| then (v, t) := (t, v); swaps := swaps + 1 end;
    [ t_y | < min(u_y, v_y), used for the error analysis. ]
    if downRound((u_x-t_x) * (t_y-v_y)) > upRound((u_y-t_y) * (t_x-v_x))
    then SignWhichSideFloat := (-1)^swaps
    elsif upRound((u_x-t_x) * (t_y-v_y)) < downRound((u_y-t_y) * (t_x-v_x))
    then SignWhichSideFloat := (-1)^(swaps+1)
    else SignWhichSideFloat := 0 end
end;

If after execution of the procedure ‘windingNumberFloat’ the variable on contains ‘false’
then the variable wn contains the correct winding number of the polygon P around the point
where the floating point numbers representing the coordinates of P’s vertices and t are
assumed to be exact. If the variable on contains ‘true’ then there exists a perturbation
bounded by 12·eps for two vertices v_1 and v_2 of P and the point t such that the perturbed
point t^* lies on the perturbed edge e^* determined by v_1^* and v_2^*.

Proof: From the previous discussion and the construction of ‘SignWhichSideFloat’ it is
clear that the results of -1 or +1 are correct in the sense that these values would have been
obtained using even infinite precision arithmetic. For the analysis we consider three cases.
(I) The statement [D] is never executed. Then on = ‘false’ and the claim is correct
according to the previous discussion.

(II) The statement [D] is executed, but on = ‘false’ after completion of the program. In this
case it is conceivable that wn should have been incremented or decremented. In the case
u_y = v_y the variable wn should not change because neither u_y < v_y nor v_y < u_y is
satisfied. In the case u_y ≠ v_y we conclude that ¬(u_y ≤ t_y ≤ v_y) ∧ ¬(v_y ≤ t_y ≤ u_y) (on
remains ‘false’) which in turn prohibits changing the variable wn.

(III) The statement [D] is executed and on = ‘true’. We consider the execution that changed
the value of on from ‘false’ to ‘true’. We can assume that t_y | ≤ min(u_y, v_y) according
to the definition of ‘SignWhichSideFloat’. We use the following abbreviations:

\[ a = (u_x - t_x) \cdot (t_y - v_y), \quad a^\uparrow = \text{upRound}(a), \quad a^\downarrow = \text{downRound}(a) \]
\[ b = (u_y - t_y) \cdot (t_x - v_x), \quad b^\uparrow = \text{upRound}(b), \quad b^\downarrow = \text{downRound}(b) \]

From ‘SignWhichSideFloat’ we obtain \( a^\downarrow \leq b^\uparrow \wedge b^\downarrow \leq a^\uparrow \). If we assume \( a^\downarrow \leq b^\downarrow \) we
get \( a^\downarrow \leq b^\downarrow \leq a^\uparrow \). Otherwise we have \( b^\downarrow \leq a^\downarrow \), resulting in \( b^\downarrow \leq a^\downarrow \leq b^\uparrow \). Since the
cases \( a^\downarrow \leq b^\downarrow \leq a^\uparrow \) and \( b^\downarrow \leq a^\downarrow \leq b^\uparrow \) are symmetric, we assume \( a^\downarrow \leq b^\downarrow \leq a^\uparrow \).

Numerical analysis, omitting second order error terms and using the facts from section
2.2.3, shows that \( a^\downarrow = a(1 - \varepsilon_1), \quad a^\uparrow = a(1 + \varepsilon_2) \) and \( b^\downarrow = b(1 + \varepsilon_3) \) with
\( 0 \leq \varepsilon_i < 6\cdot\text{eps} \). Since \( a^\downarrow \leq b^\downarrow \leq a^\uparrow \) there exists \( \varepsilon, |\varepsilon| < 6\cdot\text{eps} \) such that
\[ a(1 + \varepsilon) = b(1 + \varepsilon_3). \] The case where \( u_y = 0 \) or \( v_y = 0 \) is trivial, since in this case \( t_y = 0 \) and the sign of \((u_x - t_x) \cdot (v_y)\) or the sign of \(u_y \cdot (t_x - v_x)\) can be computed exactly which means no perturbation is necessary. Assuming \( u_y \neq 0 \) and \( v_y \neq 0 \) we rewrite the equation \( a(1 + \varepsilon) = b(1 + \varepsilon_3) \) as

\[
(u_x - t_x) \cdot (t_y - v_x) \cdot (1 + \varepsilon) = (u_y - t_y) \cdot (t_x - v_x) \cdot (1 + \varepsilon_3)
\]

\[ \iff (u_x - t_x) \cdot (t_y - v_x) \cdot \left(1 + \varepsilon \cdot \left(1 - \frac{t_x}{v_y}\right)\right) = (u_y - t_y) \cdot \left(1 + \varepsilon_3 \cdot \left(1 - \frac{t_x}{u_y}\right)\right) \cdot (t_x - v_x) \]

From \( |\varepsilon| < 6 \text{-eps} \land |\varepsilon_3| < 6 \text{-eps} \land |t_x - t_y|/v_y < 2 \land |t_x - t_y|/u_y < 2 \) we conclude the existence of the 12-eps perturbation making the equation exact.

### 2.4.9 Comparison with Epsilon Geometry

Similarly to the method of Epsilon Geometry [SSG 89] we use ideas from interval arithmetic and from numerical backward analysis (see section 2.2.3). We bound quantities by giving a lower and an upper bound and prove the existence of small perturbations when we make approximately degenerate configurations degenerate in the strict sense. Epsilon Geometry is based on \( \varepsilon \)-predicates which are true if and only if there exists a small perturbation making them true in Euclidean Geometry. This contrasts to our imprecise predicates which normally give the same answers as implemented in infinite precision arithmetic and only fail in nearly degenerate cases in which a small perturbation is needed to make the configuration truly degenerate. The advantages of requiring correct decisions in most of the cases are simplified reasoning about the correctness and easier error analysis. In both cases we need worry only about the rarer, nearly degenerate cases.

Applied to the relatively simple example of the winding number, Epsilon Geometry would give us an algorithm that computes the winding number of a slightly perturbed configuration. Unfortunately we do not know how far we are from the correct result and we do not know the sensitivity of the result against perturbations. Our method on the other hand guarantees a result that coincides with the infinite precision result most of the time and resorts to a perturbation only when needed in nearly degenerate cases.

### 2.5 A topological argument for decreasing the required precision

This section shows that sometimes the precision requirements of plane sweep algorithms can be lowered by slight modifications of the algorithm and its invariant.

High precision requirements for exact implementations of plane sweep algorithms often stem from the fact that the x-coordinates of events must be compared that were computed by the algorithm. The exact representation of such a coordinate tends to use much higher precision than ordinary input data. A typical example is the plane sweep for finding all intersections of
line segments (see also sections 3.1, 3.4 and 3.5). The naive approach needs five fold precision arithmetic when computing in rational arithmetic: An intersection of two line segments can be represented as a fraction of a threefold precision number and a twofold precision number and we need to compare two such numbers in order to determine the order of the corresponding events. In the following we show that under certain circumstances we can process events like the above in arbitrary order and still maintain a weaker version of the necessary invariants.

The idea presented not only works for straight lines but for arbitrary continuous functions of \( x \). We consider the following problem:

Given:

a) Continuous functions \( f_i(x) \), \( 1 \leq i \leq n \), defined on the closed interval \([x_0, x_1]\) such that \( f_i(x) \) and \( f_j(x) \) intersect at most once for \( i \neq j \).

b) The vertical order in which they intersect the vertical line \( x = x_0 \).

c) A primitive \( \text{intersect}(i, j) \) which determines for \( 1 \leq i \neq j \leq n \) whether \( f_i(x) \) and \( f_j(x) \) intersect on the interval \([x_0, x_1]\).

We seek an efficient algorithm for computing the new vertical order at \( x = x_1 \) and the set \( S = \{(i, j): i \neq j \land \text{intersect}(i, j)\} \).

**Theorem 2 (finding intersections of continuous functions):** The following algorithm solves the problem in time \( O(n + k \cdot \log k) \) if the dictionary \( S \) is implemented as a balanced tree. \( k \) denotes the number of intersections reported.

```plaintext
s := \emptyset; x := \emptyset; \{ X is similar to the x-queue of a plane sweep. \}
Y[1 .. n] := 'the permutation of 1 .. n corresponding to the vertical order at x_0';
for i := 1 to n - 1 do
    if \( \text{intersect}(Y[i], Y[i + 1]) \) then X := X \cup \{ i \} end;
end;
{Invariant: X = \{ i: (1 \leq i < n) \land \text{intersect}(Y[i], Y[i + 1]) \land (Y[i], Y[i + 1]) \notin S \} }
while X \neq \emptyset do
    e := \text{chooseAnyElement}(X);
    S := S \cup \{ (Y[e], Y[e + 1]), (Y[e + 1], Y[e]) \};
    Y[e], Y[e + 1] := Y[e + 1], Y[e];
    X := X \setminus \{ e - 1, e, e + 1 \};
    if \( \text{intersect}(Y[e - 1], Y[e]) \land (Y[e - 1], Y[e]) \notin S \) then
        X := X \cup \{ e - 1 \}
    end;
    if \( \text{intersect}(Y[e + 1], Y[e + 2]) \land (Y[e + 1], Y[e + 2]) \notin S \) then
        X := X \cup \{ e + 1 \}
    end;
end;
{Postcondition: S contains the set of intersections and Y contains the vertical order at x_1. }
```
Proof: The correctness proof proceeds by induction on \( k \), the number of intersections reported. In the case \( k = 0 \) or \( k = 1 \) the algorithm is obviously correct. For the induction step we describe how to rearrange the configuration of the \( n \) functions such that any intersection of two functions which are neighbors in the \( y \)-table can be processed first. The following figure contains the idea of the transformation.

![Figure 2.8: Move to front of an intersection point](image)

We claim that this transformation does not change the set \( S \), the set of intersecting pairs of functions. This follows directly from

(I) We choose \( x_\varepsilon \) such that there are no intersections with \( x \)-coordinate in \( [x_0, x_\varepsilon] \). This can be done since the set of functions is finite.

(II) No function \( f_j \) starts or ends in the shaded region.

(III) Any function \( f_j \) entering the shaded region must intersect both \( f_{Y[i]} \) and \( f_{Y[i+1]} \) since \( f_j \) cannot lie between \( f_{Y[i]} \) and \( f_{Y[i+1]} \) in the \( y \)-table at \( x_0 \) and \( f_j \) intersects \( f_{Y[i]} \) and \( f_{Y[i+1]} \) at most once. Therefore the set of intersections is unchanged.

(IV) The topology to the right of the original intersection between \( f_{Y[i]} \) and \( f_{Y[i+1]} \) has not been changed.

(V) All other functions remain the same.

The x-structure is implemented as a combination of a bit-vector and a linked list such that the operations 'chooseAnyElement', 'insert' and 'delete' can be performed in constant time after an \( O(n) \) initialization phase. The dictionary \( S \) is implemented as a balanced tree guaranteeing logarithmic performance for insert and delete.

The algorithm above is similar to the standard plane sweep method for finding all intersections of a set of line segments in the plane with the difference that we process intersection events in an arbitrary order. This is admissible because the x-structure only contains intersection events of segments that are currently neighbors in the \( y \)-table. This algorithm is less general since it does not handle the events 'start of a continuous function' and 'end of a continuous function' and in fact this method would fail in the general setting.

The following figure shows that an ordinary implementation of the plane sweep for finding all intersecting line segments may produce an incorrect order in its \( y \)-table if intersection events are processed in an arbitrary order.
The reason for this incorrect behavior comes from exchanging segments that are no longer direct neighbors in the y-table when processing intersection events.

A direct application of theorem 2 is the computation of all intersections of $n$ line segments using integer arithmetic in section 3.5. The reduction of the necessary precision in the algorithm for the all-nearest-neighbors problem in section 3.3 uses similar ideas.

2.6 Evaluation of the methods

In the preceding sections we have introduced three new methods for the construction of provably robust algorithms. Unfortunately not all methods work for all algorithms and we must answer the questions 1) how to choose a method for a given algorithm and 2) when does a certain method work for a certain algorithm.

The topological argument for decreasing precision only works for plane sweep algorithms that maintain boundaries in their y-structure, sorted by y-coordinate. Furthermore this method requires integer arithmetic and is therefore rather limited, although we present applications in chapter 3.

The two other methods are much more general. The axiomatic approach should be tried first since the resulting program still resembles the infinite precision version. On the other hand, finding a consistent, rounded arithmetic implementation of the underlying primitives is often impossible.

When the axiomatic approach fails, one can try the method of explicitly detecting nearly degenerate configurations. At least this method guarantees correct results or the indication of failure in nearly degenerate cases. Treating these configurations as degenerate often results in a program which is correct in the spirit of backwards analysis: The result of the computation is correct for slightly changed input data. The method fails when we cannot prove this property or we can show that no such small perturbation exists.
3 Provably accurate programs for basic geometric problems

This chapter applies the techniques from the previous one to the following problems: Closest-pair, all-nearest-neighbors, and two versions of the line segment intersection problem. We briefly present the algorithmic principle of plane sweep which is instrumental to the algorithms of this chapter.

3.1 Principles of plane sweep

Plane sweep is one of the most successful algorithmic principles in 2-d computational geometry. Incremental in nature, it avoids costly bookkeeping overhead and complicated data structures often found in divide and conquer algorithms. In its basic form a vertical line, called front, is swept from left to right over the geometric configuration. The invariant keeps the result of the computation pertaining to the configuration to the left of the front. The front does not move continuously but stops at discrete events whenever the invariant must be updated. The y-table stores the information needed to update the invariant while the x-queue determines the event schedule.

As an example consider the problem of detecting whether a given set of n straight line segments contains an intersecting pair. This problem and a generalization thereof serves as an example in sections 3.4 and 3.5.

As an invariant we keep the order in which the front intersects the given line segments. The crucial observation is that, in a non-degenerate configuration, two segments can only intersect if they have been neighbors in this order. Therefore it suffices to maintain this order in the y-table and to perform an intersection test whenever the neighbor relation in the y-table changes. This is only possible at three types of events when the sweep line encounters

(I) the starting point (left end) of a segment s. Segment s is inserted into the y-table and tested for intersection against its lower and its upper neighbor.

(II) the end of a segment s. Its upper neighbor is tested for intersection against its lower neighbor before s is removed from the y-table.

(III) an intersection of two segments s and t with s < t in the current order. Segments s and t are exchanged in the y-table, s is tested for intersection against its new upper neighbor and, symmetrically, t is tested for intersection against its new lower neighbor.

In the first version of the segment intersection test (existence) the algorithm stops as soon as an intersection is found or if all segments have been processed. No events of type (III) exist in this case. The x-queue contains the 2-n end points of the segments sorted by their x-coordinates. The following figure shows a typical snapshot of this algorithm.
Note that the intersection between segment 3 and segment 5 is not detected when segment 3 starts but when segment 4 ends.

### 3.2 The closest-pair problem

Given a set $S$ of $n$ points in the plane, find two points with smallest distance. Note that we usually cannot identify a correct closest pair when using floating point arithmetic. Instead we focus on the distance $\delta$ of the closest pair. Implemented in floating point arithmetic, the trivial $O(n^2)$ algorithm achieves the most accurate result since every possible pair is considered. Using the axiomatic approach from section 2.3 we design an optimal plane sweep algorithm that realizes the same accuracy.

#### 3.2.1 Review: A plane sweep approach to the closest-pair problem

We review the algorithm presented in [HNS 88], then prove its accuracy.
The points are sorted by x-coordinate and the configuration of points is swept from left to right stopping at each data point. As an invariant we keep the distance $\delta$ of the closest pair of the points to the left of the sweep line. The y-table contains the points of the $\delta$-slice, sorted by their y-coordinates. The $\delta$-slice consists of all points that differ in their x-coordinate by at most $\delta$ from the x-coordinate of the sweep line. Processing a data point $p$ consists of removing all points $q$ from the y-table for which $p_x - q_x > \delta$, inserting the new data point $p$ into the y-table and checking all its neighbors $r$ in the y-table for which $|p_y - r_y| < \delta$.

The algorithm’s complexity is $O(n \log n)$ which is optimal in the algebraic decision tree model of computation. The proof depends on the fact that for each data point at most $\delta$ points in the y-table must be examined.

The proof of its accuracy proceeds according to the axiomatic approach in two steps: 1) We show that an implementation using floating point arithmetic is consistent with the invariants, and 2) we bound the error using a lemma from numerical analysis.

### 3.2.2 The algorithm and its correctness proof

**Theorem 3 (sufficient conditions for the correctness of 'ClosestPair'):** If the axioms $A$ hold for the primitives $d$, $dx$, $dy_r$ and $dy_t$, then the algorithm 'ClosestPair' is correct.

\[
A = \{
\begin{align*}
&d: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} & \text{[Normally: } (p_x - q_x)^k + (p_y - q_y)^k]^{1/k}] \\
&dx: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} & \text{[Symmetry]} \\
&dy_r: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} & \text{[Normally: } p_y - q_y] \\
&dy_t: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} & \text{[Normally: } q_y - p_y] \\
&dy_{r,t}: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} & \text{[Monotonicity in the second argument]} \\
&dy_{r,t}: \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} & \text{[Lower bound with respect to the distance]} \\
\end{align*}
\]

\[
\text{ClosestPair} = \begin{array}{c}
\text{S := sortByX(S); c := 3; t := 1; } \delta := d(S[1], S[2]); \ Y := \{ S[1], S[2] \}; \\
\text{[A] while } c \leq n \text{ do} \\
\text{[ } \text{Invariant: } \delta = \delta(c) \wedge Y = Y(c) \wedge t = t(c) \text{ where} \\
\text{[ } \delta(c) = \min( d(S[i], S[j]): 1 \leq i < j < c) \\
\text{[ } Y(c) = \{ S[i]: 1 \leq i < c \wedge dx(S[i]-1), S[i] < \delta(c-1) \} \\
\text{[ } t(c) = \min( i: S[i] \in Y(c) ) \\
\text{transition; } c := c + 1 \\
\text{[B] end; } \text{[Post condition PC: } \delta = \min( d(S[i], S[j]): 1 \leq i < j \leq n )]
\end{array}
\]
transition =

\[ p := S[c]; \]
\[ \text{while } dx(p, S[t]) \geq \delta \text{ and } (t < c) \text{ do } \]
\[ \text{deleteY}(S[t]); t := t + 1 \quad \{ \text{Normally: } p_x - S[r]_x \geq \delta \} \]
\[ \text{end;} \]
\[ \text{insertY}(p); \]
\[ q := p; \]
\[ \text{repeat} \]
\[ q := \text{predY}(q); \text{if } d(p, q) < \delta \text{ then } \delta := d(p, q) \quad \{ \text{Normally: } p_y - q_y > \delta \} \]
\[ \text{until } dy_x(p, q) > \delta; \quad \{ \text{Normally: } q_y - p_y > \delta \} \]
\[ q := p; \]
\[ \text{repeat} \]
\[ q := \text{succY}(q); \text{if } d(p, q) < \delta \text{ then } \delta := d(p, q) \quad \{ \text{Normally: } p_y - q_y > \delta \} \]
\[ \text{until } dy_x(p, q) > \delta; \quad \{ \text{Normally: } q_y - p_y > \delta \} \]

**Proof:** We adopt the conventions that \( \min(\emptyset) = +\infty \) and that the successor (predecessor) of the last (first) point in the y-table is a point at infinity. Since termination is clear, it suffices to show the validity of the invariant.

**Claim 1:** The invariant is true at \( \{A\} \).

**Proof:**
\[
\delta(3) = \min( d(S[i], S[j]) : 1 \leq i < j \leq 2 ) = d(S[1], S[2])
\]
\[
Y(3) = \{ S[i] : 1 \leq i \leq 2 \wedge dx(S[2], S[i]) < \delta(2) \} = \{ S[1], S[2] \}, \text{ since } \delta(2) = +\infty
\]
\[
t(3) = \min( i : S[i] \in Y(3) ) = 1
\]

**Claim 2:** The invariant is true at \( \{B\} \).

**Proof:**
\[
a) \text{ The y-table } Y \text{ is correct: } Y = Y(c + 1) \wedge t = t(c + 1)
\]
\[
(I) \quad 1 \leq i < t(c) \Rightarrow
\]
\[
dx(S[c], S[i]) \geq dx(S[c-1], S[i]) \quad \{ \text{Monoticity of } dx, \}
\]
\[
\geq \delta(c - 1) \quad \{ \text{S is sorted by x-coordinate.} \}
\]
\[
\geq \delta(c) \quad \{ \text{Monoticity of min} \}
\]
\[
\Rightarrow S[i] \notin Y(c + 1)
\]
\[
(II) \quad t(c) \leq i < t \Rightarrow
\]
\[
dx(S[c], S[i]) \geq \delta(c) \quad \{ \text{Invariant of while-loop in procedure transition} \}
\]
\[
\Rightarrow S[i] \notin Y(c + 1)
\]
\[
(III) \quad t \leq i \leq c \Rightarrow
\]
\[
dx(S[c], S[i]) \leq dx(S[c], S[i]) \quad \{ \text{Inverse monotonicity of } dx, \}
\]
\[
< \delta(c) \quad \{ \text{S is sorted by x-coordinate.} \}
\]
\[
\Rightarrow S[i] \in Y(c + 1)
\]
\[
(I) \wedge (II) \wedge (III) \Rightarrow Y = Y(c + 1) \wedge t = t(c + 1)
\]
b) The shortest distance is correct: $\delta = \delta(c + 1)$

\[(IV) \quad d_{y_T}(S[c], q) > \delta \Rightarrow d_{y_T}(S[c], \text{succ}^k(q)) > \delta \quad \{ \text{Monotonicity of } d_{y_T} \} \]
\[\Rightarrow d(S[c], \text{succ}^k(q)) > \delta \quad \{ \text{Lower bound property for } d_{y_T} \} \]

\[(V) \quad d_{y_I}(S[c], q) > \delta \Rightarrow d_{y_I}(S[c], \text{pred}^k(q)) > \delta \quad \{ \text{Monotonicity of } d_{y_I} \} \]
\[\Rightarrow d(S[c], \text{pred}^k(q)) > \delta \quad \{ \text{Lower bound property for } d_{y_I} \} \]

\[(VI) \quad (I) \land (II) \Rightarrow 1 \leq i < r: dx(S[c], S[i]) \geq \delta(c) \Rightarrow d(S[c], S[i]) \geq \delta(c) \quad \{ \text{Lower bound property for } dx \} \]

Note that the distance function $d$ requires only symmetry for a correct result.

### 3.2.3 Consistent interpretation

This section presents a floating point implementation of the primitives that is consistent with the previous correctness proof. In the following we denote the floating point implementation of the distance function $d$ by $d^*$. Recall from definition 2 in section 2.2.3 that $-^*$ denotes floating point subtraction.

**Theorem 4 (consistent floating point interpretation for 'ClosestPair')**: The interpretation of 'ClosestPair' with

\[
d(p, q) = \max(ldx(p, q), ldy_T(p, q), d^*(p, q))
\]
\[
dx(p, q) = p_x -^* q_x
\]
\[
dy_T(p, q) = p_y -^* q_y \quad \text{and}
\]
\[
dy_I(p, q) = q_y -^* p_y
\]

is consistent.

**Proof**: The monotonicity laws for floating point subtraction hold for any reasonable implementation, since we only require that $a - b$ does not increase if $b$ is increased. The lower bound property of $dx$, $dy_T$ and $dy_I$ follows directly from the definition of $d(p, q)$.

Trivial interpretations, e.g. setting all functions identical to zero, show that consistency is not enough to guarantee an accurate result. We must analyze the resulting post condition PC numerically.

### 3.2.4 Numerical analysis

The following two theorems establish a connection between the computed result and the exact result.
Theorem 5 (error analysis for the minimum function): The minimum of \( n \) imprecise quantities with a common relative error bound \( \varepsilon \) is an \( \varepsilon \)-perturbation of the minimum of the precise quantities with \( \varepsilon_\text{rel} < \varepsilon \):

\[
\min(a^*_i:1 \leq i \leq n) = \min(a_i:1 \leq i \leq n) \cdot (1 + \varepsilon) \quad \text{where} \quad a^*_i = a_i \cdot (1 + \varepsilon_i), \quad |\varepsilon_i| < \varepsilon, \quad \varepsilon < \varepsilon_\text{rel} < 1.
\]

**Proof:** By induction over \( n \). The case \( n = 1 \) is trivial, hence we concentrate on \( n = 2 \).

\[
m := \min(a^*, b^*) = \min(a \cdot (1 + \varepsilon_a), b \cdot (1 + \varepsilon_b)), \quad |\varepsilon_a| < \varepsilon, \quad |\varepsilon_b| < \varepsilon.
\]

**Case 1:** \( a \geq 0, b \geq 0 \).

\[
a \cdot (1 - \varepsilon) < a^* = a \cdot (1 + \varepsilon_a) < a \cdot (1 + \varepsilon) < b \cdot (1 + \varepsilon_a) < b \cdot (1 + \varepsilon)
\]

\[
\min(a \cdot (1 - \varepsilon), b \cdot (1 - \varepsilon)) < m < \min(a \cdot (1 + \varepsilon), b \cdot (1 + \varepsilon))
\]

\[
(1 - \varepsilon) \cdot \min(a, b) < m < (1 + \varepsilon) \cdot \min(a, b)
\]

\[
\Rightarrow \exists \varepsilon: -\varepsilon < \varepsilon < \varepsilon_\text{rel} : m = (1 + \varepsilon) \cdot \min(a, b)
\]

**Case 2:** \( a < 0, b < 0 \): analogously to case 1.

**Case 3:** \( a < 0, b \geq 0 \)

\[
m = a \cdot (1 + \varepsilon_a) = (1 + \varepsilon_a) \cdot \min(a, b)
\]

**Case 4:** \( a \geq 0, b < 0 \): analogously to case 3.

**Induction step:**

\[
\min(a^*_i:1 \leq i \leq n+1) = \min(\min(a^*_i:1 \leq i \leq n), a^*_{n+1}) = \min((1 + \varepsilon) \cdot \min(a_i:1 \leq i \leq n), a^*_{n+1}) = (1 + \varepsilon) \cdot \min(a_i:1 \leq i \leq n+1)
\]

Theorem 6 (accuracy of the floating point implementation of 'ClosestPair'): Interpreting the program scheme 'ClosestPair' with the interpretation of theorem 4 yields an implementation which computes the distance of the closest pair with the same or better relative error bound as the distance function \( d^* \).

**Proof:** Theorem 4 shows that the floating point interpretation is consistent which proves the validity of the post condition PC. Theorem 5 proves that the minimum of the computed distances is a minimum of the correct distances with a perturbation of the same order of magnitude with which we compute distances. The interpretation \( d(p, q) = \max( |dx(p, q)|, |dy(p, q)|, d^*(p, q) ) \) could have a better relative error bound than \( d^*(p, q) \) in cases where \( d^*(p, q) < |px - qx| \) or \( d^*(p, q) < |py - qy| \).

We can implement \( d^*(p, q) \) such that the relative error is bounded by \( 8 \cdot \varepsilon \), assuming a floating point square root function \( \sqrt{\text{rel}}(x) \) such that \( \sqrt{\text{rel}}(x) = \sqrt{x} \cdot (1 + \varepsilon), \varepsilon < \varepsilon_\text{rel} < 1 \).

Note also that implementing the line 'while \( p_x - S[i]_x \geq \delta \) do' as 'while \( p_x - S[i]_x \geq \delta \) do' leads to a highly increased error bound if the distances and coordinates differ greatly in magnitude. As an example, consider floating point arithmetic with a two digit decimal mantissa with \( p_x = 10^2 \), \( S[i]_x = 99 \), \( d = d^* = 1.4 \) and \( p_y = S[i]_y \). The latter implementation of the while loop would discard the point \( S[i] \) although \( d(p, S[i]) = 1 < 1.4 \) creating a relative error of 0.4 which is much greater than the machine epsilon of 0.05.
The analysis of this section can be applied to the case where we use $d_K(p, q)$ instead of $d(p, q)$ in order to avoid the costly root extraction operations.

### 3.2.5 Generalization and extension to higher dimensions

The all-nearest-neighbors problem asks for a nearest neighbor for each given point. In section 3.3 we describe an optimal algorithm for the 2-dimensional problem. The following algorithm [FBS 75, PB 80] finds nearest neighbors using projections. The nearest neighbor of a point $p$ is found by simultaneously examining the horizontal and the vertical $2\delta$ slice centered around $p$, stopping as soon as one of them has been totally processed. $\delta$ denotes $p$'s distance to its nearest neighbor and examining a slice is done by computing the minimum distance between $p$ and all points $q$ in the slice.

![Figure 3.3: The projection method for finding nearest neighbors](image)

**Input:** A set $S$ of $n$ points in the plane

**Output:** For each point the distance to a nearest neighbor

```plaintext
x := sortByX(S); y := sortByY(S); \{ X, Y: array [1..n] of point \}
FOR EACH p IN S DO
    left := find(p, x); down := find(p, y); \{ By binary search \}
    right := left; up := down; $\delta := \infty$;
    REPEAT
        left := left - 1; right := right + 1; down := down - 1; up := up + 1
    UNTIL (halfSliceDone(X, left) ∨ halfSliceDone(X, right)) ∨
    (halfSliceDone(Y, down) ∨ halfSliceDone(Y, up));
    $\delta(p) := \delta$
END;

FUNCTION halfSliceDone(Z, dir): boolean;
BEGIN
    halfSliceDone := (dir ≤ 0) ∨ (dir > n) ∨ ($|Z[dir]|_2 - p_2| ≥ \delta$);
    IF ¬halfSliceDone ∧ (d(Z[dir], p) < $\delta$) THEN $\delta := d(Z[dir], p)$ END
END;
```
For the L_\infty-metric the version generalized to d-dimensions has worst case running time \(Q(n^{2-1/d})\) which is achieved by placing \(n\) points on the integer grid points inside the \(d\)-dimensional cube \([0, n^{1/d}]^d\). For other L_p-metrics the worst case running time is \(\Theta(n^2)\) because the space left between the L_p-circle and its circumscribing square can be used to construct a sequence \(p_i\) of points such that the point \(p_i\) examines \(p_{i+1}, p_{i+2}, \ldots, p_n\) [BH 90].

The probabilistic analysis in [BH 90] shows that the running time is still \(\Theta(n^{2-1/d})\) for uniformly distributed points in the \(d\)-dimensional unit cube.

This simple algorithm is useful in practice. The even simpler version that examines only the vertical 2-8-slice is faster than the optimal algorithm for point sets with a size less than a few thousand points (see section 5.2.1 for experimental results). The analysis from the closest-pair algorithm carries over and we can compute nearest neighbors provably highly accurate.

### 3.3 The all-nearest-neighbors problem

In this section we present a sweep algorithm for solving the following problem: Given a set \(S\) of \(n\) points in the plane, find a nearest neighbor for each of them. A detailed description of the algorithm is in [HNS 90]; we concentrate on the issues of degeneracy and accuracy. In order to make this section self-contained we give a brief explanation of the algorithm.

As a first step we break the symmetry of the problem: Instead of finding a nearest neighbor for a point \(p \in S\) we find a nearest neighbor that comes lexicographically before \(p\), i.e. a nearest neighbor to the left. The actual nearest neighbor is found by performing the sweep from left to right and from right to left. This step is necessary because in a typical plane sweep algorithm we can only use information about the geometrical configuration already seen as opposed to the whole configuration.

**Definitions and facts**

rightTurn(u, v, w) \(\iff\) \(A(u, v, w) < 0\) (see also definition 1 in section 2.1)

center(u, v, w): the center of the circle through \(u, v\) and \(w\) for \(A(u, v, w) \neq 0\).

L: The sweep line, a vertical line at x-coordinate \(x_L\).

\(S_{left}\): The set of those points of \(S\) that lie to the left of \(L\) or on \(L\).

\(s \in S_{left}\) is active \(\iff\) \(\exists p \in L \land q \in S_{left}: d(s, p) \leq d(q, p)\)

In words, a point in \(S_{left}\) is active iff it is a nearest neighbor to the left of some point \(p\) on \(L\).

\(S_{active}\): The set of all active points. Notice: \(S_{active} \subseteq S_{left} \subseteq S\)

When we wish to emphasize that \(S_{active}\) depends on \(L\), we write \(S_{active}(L)\).

**y-order \(\prec_y\) on \(S_{active}\):** For \(u, v \in S_{active}\), define \(u \prec_y v \iff u_y < v_y\)

\(u, v, w \in S_{active}\) form a consecutive triple \(\iff u = \text{pred}(v), \text{succ}(v) = w\)

For a consecutive triple \((u, v, w)\) with \(\text{center}(u, v, w)\) to the right of \(L\), i.e. \(\text{center}(u, v, w) \geq x_L\), \(\text{center}(u, v, w)\) is called the ‘deactivation event of \(v\)’. 
Fact (local activity test): For a consecutive triple \((u, v, w) \in S_{\text{active}}\):
\[
\text{rightTurn}(u, v, w) \Rightarrow \text{center}(u, v, w)_x > x_L
\]

The sweep invariant
As an invariant we maintain in the y-table the set \(S_{\text{active}}(L)\) which is updated when the sweep line \(L\) reaches either two types of events, deactivations and data points. Data points are given a priori, and deactivation events are generated during the sweep. Although several deactivation events for the same point may be generated we only keep the one with smallest x-coordinate.

Deactivation event
The center of the circle of the consecutive triple \((u, v, w)\) with rightTurn\((u, v, w)\) is called a 'deactivation event for \(v\)'. It is processed by removing \(v\) from the y-table, then testing whether the consecutive triples \((\text{pred}(u), u, w)\) and \((u, w, \text{succ}(w))\) generate new deactivation events to be inserted into the x-queue.

![Figure 3.4: Deactivation event for \(v\)](image)

Data point
A new data point \(p \in S\) is inserted into the y-table. We must test whether the consecutive triples \((p, \text{succ}(p), \text{succ(succ}(p)))\) and \((\text{pred(pred}(p)), \text{pred}(p), p)\), if they exist, pass the local activity test. The figure shows an example where the triple \((p, \text{succ}(p), \text{succ(succ}(p)))\) fails the local activity test: It forms a right turn, and center\((p, \text{succ}(p), \text{succ(succ}(p)))\) lies to the left of \(L\).

![Figure 3.5: Data point event \(p\)](image)

Hence \(\text{succ}(p)\) is no longer active and is therefore removed from the y-table. The old \(\text{succ}(\text{succ}(p))\) becomes the new \(\text{succ}(p)\). We continue moving upwards until the triple \(T := (p, \text{succ}(p), \text{succ(succ}(p)))\) passes the local activity test. If rightTurn\((T)\) then a deactivation event at position center\(_x(T)\) is generated. The triple \((\text{pred(pred}(p)), \text{pred}(p), p)\) is handled symmetrically by moving downwards. The remaining triple \((\text{pred}(p), p, \text{succ}(p))\) automatically passes the local activity test since \(-\text{rightTurn}(\text{pred}(p), p, \text{succ}(p))\).

Finding the nearest neighbor
After updating the invariant, the nearest neighbor to the left of a data point \(p\) is found among
pred(p), succ(p) and the points that were deactivated by p and removed from the y-table. One can show that the remaining points in the y-table have a larger distance from p than succ(p) or pred(p). While updating the invariant we maintain the nearest neighbor to the left of p found so far.

3.3.1 Program structure

```plaintext
procedure transition(p); { Processes a transition event. }
begin
  if dataPoint(p) then
    insertY(p); nn := nil;
    updateY(p, nn, succY, -1); updateY(p, nn, predY, +1)
  elsif deactivate(p) then
    u := succY(p); v := predY(p); deleteY(p);
    testForDeactivate(predY(v), v, u);
    testForDeactivate(v, u, succY(u))
  end
end transition;

procedure updateY(p, nn, nextY, correctSign);
begin
  u := nextY(p); v := nextY(u);
  while (v ≠ nil) & (sign(A(p, u, v)) = correctSign) & (centerX(p, u, v) < p.x) do
    if d(p, u) < d(p, nn) then nn := u end;
  deleteY(u); u := v; v := nextY(u)
  if d(p, u) < d(p, nn) then nn := u end;
  testForDeactivate(p, u, v)
end updateY;
```

Conventions: ‘&’ denotes the conditional and operator, d(p, nil) = d(nil, p) = +∞, succY(nil) = predY(nil) = nil.

insertY(p) inserts p into the y-table. If p_y = q_y for q ∈ y-table then p is inserted as the successor of q.

deleteY(p) removes p from the y-table and a deactivation event for p from the x-queue if present.

testForDeactivate(u, v, w) schedules the leftmost deactivation event for v, if necessary, and inserts it into the x-queue. Any deactivation event for v further to the right is deleted from the x-queue.

3.3.2 An exact implementation in triple precision integer arithmetic

In this section we show how to handle degenerate cases and prove that triple precision integer arithmetic suffices to guarantee exact results.
Sequencing events in degenerate configurations

Applying definition 6 from section 2.4.1 to the above program we find the following three kinds of degeneracies.

1) Three collinear points \( p, q \) and \( r \). In such cases we do not compute a center of a circle and do not deactivate \( q \) which is correct since \( q \) might be a nearest neighbor to the left of some point \( t \). Essentially this degenerate case is treated as a non degenerate one.

2) Points \( p, q \) and \( r \) such that \( d(p, q) = d(p, r) \). This case is handled by updating the minimum distance found so far only if a strictly smaller value is encountered.

3) Events which coincide or have equal x-coordinates. They are handled by extending the order on events in the x-queue to the case of equal x- and / or y-coordinates. We define the order on events of the form \((x, y, \text{dataPoint})\) and \((x, \text{deactivation})\) as follows:

\[
\begin{align*}
(x_1, y_1, \text{dataPoint}) &< (x_2, y_2, \text{dataPoint}) \iff (x_1 < x_2) \lor (x_1 = x_2) \land (y_1 < y_2) \\
(x_1, y_1, \text{deactivation}) &< (x_2, \text{deactivation}) \iff x_1 \leq x_2 \\
(x_1, \text{deactivation}) &< (x_2, \text{dataPoint}) \iff x_1 < x_2 \\
(x_1, \text{deactivation}) &< (x_2, \text{deactivation}) \iff x_1 \leq x_2
\end{align*}
\]

The only effect of a deactivation event is the deletion of a point from the y-table. Hence deactivation events with equal x-coordinates can be processed in any order – we need not break the tie in this case.

Triple precision integer arithmetic suffices

In the following we assume that all input data points lie on a finite integer grid, i.e. there exists an integer \( M \) such that any data point has integer coordinates \((x, y)\) with \(|x| \leq M\) and \(|y| \leq M\). Straightforward application of rational arithmetic leads to a five fold precision implementation, since we must compare the x-coordinates \((n_i, d_i)\), \(i = 1, 2\), of two deactivation events. This is the most expensive computation with respect to precision and the formula in lemma 4 of section 2.1 shows that \( n_i \) needs three fold precision while \( d_i \) needs two fold precision.

We decrease the required precision by a technique similar to the topological argument presented in section 2.5. We implement the division operation in the computation of the x-coordinate of a deactivation event with the truncated division operator ‘\( \text{div} \)’. This does not change the order between data point events and deactivations events since data points have integer coordinates and are processed before deactivation events in the case of ties. We might change the order in which consecutive deactivation events are processed which poses no problem since any order is admissible.

The most expensive step is the computation of the x-coordinate of a deactivation event. Lemma 4 in section 2.1 gives us the numerator and denominator of this quantity while lemma 5 in section 2.1 allows us to bound them both. Under the assumption that the absolute value of all input data coordinates is bounded by \( M \) we obtain \( 4M^3 \) as a bound for
the numerator and $8M^2$ as a bound for the denominator.

Conclusion: Let $m$ be the number of bits needed to represent the input data coordinates, then integer arithmetic with $3m + 2$ bits yields an exact result.

### 3.4 The line segment intersection test in floating point arithmetic

In this section we apply the method from section 2.4 to solve the segment intersection problem using the plane sweep introduced in section 3.1. The following trace of an incorrect implementation demonstrates the numerical problems.

We consider the following configuration of three line segments and perform all computations with a floating point number system that has two decimal digits in the mantissa (i.e. all numbers are represented in the form $0.d_1d_2 \cdot 10^e$, $1 \leq d_1 \leq 9$, $0 \leq d_2 \leq 9$).

![Figure 3.6: Plane sweep might miss an intersection due to numerical errors](image)

Trace of the intersection plane sweep:

1) $x = 14$: Insert $R$ in the y-table

2) $x = 20$: Insert $S$ above $R$ in the y-table and test $S$ and $R$ for intersection

3) $x = 30$: Insert $T$ below $R$ in the y-table, since

   $$-66 = (13 - 16) \cdot (36 - 14) < (30 - 14) \cdot (12 - 16) = -64 \Rightarrow T_p \text{ below } R$$

   (Note that: $88 = (20 - 16) \cdot (36 - 14) > (34 - 14) \cdot (12 - 16) = -80 \Rightarrow T_q \text{ above } R$)

4) $x = 30$: Intersection test between $T$ and $R$

   The intersection between $T$ and $R$ is not reported because the computed intersection point is $(29, 13)$. Its x-coordinate is computed using the following formula:
\[
\frac{(T_x - T_y) \cdot (R_x - R_y) \cdot (R_y - T_y) + (R_x - R_y) \cdot (T_y - T_x) \cdot (T_x - T_y) \cdot (R_x - R_y) \cdot (T_y - T_x)}{(R_x - R_y) \cdot (T_y - T_x) - (R_y - R_y) \cdot (T_x - T_y)}
\]

5) \( x = 34 \): Remove \( T \) from the y-table

6) \( x = 36 \): Remove \( R \) from the y-table

7) \( x = 40 \): Remove \( S \) from the y-table

As a result we have missed the obvious intersection between \( S \) and \( T \), an intersection which would have been detected even in this low precision arithmetic if we had checked each pair of line segments for intersection. The reason for the error is, again, the incorrect treatment of nearly degenerate cases: The case where the start point \( T_p \) of segment \( T \) lies almost on segment \( R \) should have been reported as the first intersection.

As a solution we use the method from section 2.4 and explicitly detect nearly degenerate cases. We will see that we can stop in such a case because it can be viewed as an intersection. We formulate our result in the following theorem.

**Theorem 7** (robust intersection test for a set of line segments): Given a set \( S \) of \( n \) line segments in the plane where the coordinates of the end points are given in floating point arithmetic. Then there exists an optimal \( O(n \log n) \) algorithm which decides whether there exist two different segments in \( S \) that intersect. The decision ‘no’ is correct assuming that the floating point numbers defining the coordinates are exact and infinite precision arithmetic is used while for the decision ‘yes’ a \( k \cdot \varepsilon_p \)-perturbation of two segments must be performed to obtain an intersection. The constant \( k \), with \( k \geq 12 \), depends on the intersection routine used.

**Proof:** We show that the following sweep algorithm solves the problem.

```plaintext
type
    point = record x, y: float end;
    lineSegment = record p, q: point end; \{ \( p <_{lex} q \) \}
    event = record where: point; kind: (start, end); k: lineSegment end;

var
    S : array [1 .. n] of lineSegment;
    xQueue: priorityQueue of event;
    yTable: dictionary of lineSegment;
    found: boolean; \{ Whether an intersection has been found \}

begin \{ Of algorithm ‘FirstIntersect’ \}
    xQueue := \( \emptyset \); yTable := \( \emptyset \); found := false;
    for \( i := 1 \) to \( n \) do
        xQueue.insert([S[i].p, start, S[i]], \( \leq_{ev} \));
        xQueue.insert([S[i].q, end, S[i]], \( \leq_{ev} \))
end;
```
while (xQueue ≠ ∅) ∧ ¬found do
  e := xQueue.extractMinimum(<ev)
  case e.kind of
  start:
    yTable.insert(e.k, <ls);
    u := yTable.pred(e.k, <ls); v := yTable.succ(e.k, <ls);
    found := intersect(u.p, u.q, e.k.p, e.k.q) \lor
              intersect(e.k.p, e.k.q, v.p, v.q)
  case:
    u := yTable.pred(e.k, <ls); v := yTable.succ(e.k, <ls);
    found := intersect(u.p, u.q, v.p, v.q); yTable.delete(e.k)
  end { case }
end { while }
end;

function <ev(a, b: event): boolean;
begin
  <ev := (a.where <lex b.where) \lor ((a.where = b.where) ∧ (a.kind < b.kind))
end;

function <ls(k, l: lineSegment): boolean;
{ Precondition: The sweep line focus is at k.p. }
begin
  case SignWhichSideFloat(l.p, k.p, l.q) of
    -1: <ls := false
    0: found := true; exit(FirstIntersect)
    +1: <ls := true
  end
end;

function intersect(p, q, r, s: point): boolean;
{ Precondition: p <lex q ∧ r <lex s. Decides whether the segments pq and rs intersect. 'false'
  means no intersection assuming infinite precision arithmetic while 'true' means that there
  exists a perturbation by at most k-eps such that pq and rs intersect. We implement 'intersect'
  using similar ideas as in the implementation of 'SignWhichSideFloat'. An error analysis for
  a line intersection routine is in [MST89]. }

The primitive operation 'SignWhichSideFloat' must meet the following requirements.
1) SignWhichSideFloat(p, r, q) < 0 ⇒ r lies to the left of pq.
2) SignWhichSideFloat(p, r, q) = 0 ∧ p <lex r ∧ r <lex q ⇒ there exists a
   12-eps perturbation of at most two points in {p, q, r} such that r lies on pq.
3) SignWhichSideFloat(p, r, q) > 0 ⇒ r lies to the right of pq.

In section 2.4.8 we have shown how to implement this operation using the technique from
section 2.4.3. From the definition of 'SignWhichSideFloat' follows the correctness of the
plane sweep algorithm 'FirstIntersect' immediately for all cases where 'SignWhichSideFloat'
is nonzero. It remains to be shown that nearly degenerate cases are handled correctly.
As an invariant, we maintain the same order of the line segments as if we had infinite precision arithmetic. This is true because of the following two facts.

(I) We stop as soon as we find a pair of segments that intersect or that nearly intersect. The negative decision of the function 'intersect' is correct by definition.

(II) We stop as soon as SignWhichSideFloat(l.p, k.p, l.q) = 0 inside the function <ls. The function <ls is only invoked if \( l.p \leq_{\text{lex}} k.p \leq_{\text{lex}} l.q \) since the events are ordered lexicographically and segment \( l \) is already present in the y-table. Property 2) of 'SignWhichSideFloat' from above implies that we can create a real intersection by perturbing at most two of the points in \( \{l.p, k.p, l.q\} \). This justifies the result found := true.

3.5 Finding all intersections of line segments in the plane

The problem of finding all pairs of intersecting straight line segments in the plane is one of the fundamental problems in computational geometry. Although recently [CE 88] a theoretically time optimal algorithm was discovered, we focus on a variation of the well known plane sweep algorithm [BO 79, Me 84, PS 88] for solving the problem which generalizes the algorithm from the previous section. The plane sweep is of complexity \( O(n + k) \cdot \log n \) whereas the time optimal algorithm achieves \( O(n \log n + k) \) where \( k \) denotes the number of intersections reported.

We aim for two goals which are usually omitted from the literature, namely the ability to handle all degenerate cases and the property of needing a relatively low precision arithmetic for an exact implementation. Milenkovic states in [M 89] that four-fold precision arithmetic is needed for this problem. We lower this bound to three-fold precision.

The text book literature in computational geometry [Me 84, PS 88] does not state how to handle all degenerate cases. Moreover, the description of the plane sweep in the book of Preparata and Shamos [PS 88, p. 284] contains a minor bug: The test in line 21 'if \((s_3 \text{ intersects } s_2) \text{ then } ... \)' should be strengthened to 'if \((s_3 \text{ intersects } s_2 \text{ to the right of } p) \text{ then } ... \)' similarly to line 15 in order to prevent the sweep line from returning to an already processed event. The analogous problem exists in line 22.

In the following we show that the description of an algorithm handling all degenerate cases is only slightly longer but much more useful from the implementor's point of view.

3.5.1 Problem statement

We must state precisely what kind of result we want to compute, since many possibilities exist in the presence of degeneracies. We use the following declarations.

```pascal
3 45
```

```pascal
3.5 Finding all intersections of line segments in the plane

The problem of finding all pairs of intersecting straight line segments in the plane is one of the fundamental problems in computational geometry. Although recently [CE 88] a theoretically time optimal algorithm was discovered, we focus on a variation of the well known plane sweep algorithm [BO 79, Me 84, PS 88] for solving the problem which generalizes the algorithm from the previous section. The plane sweep is of complexity \( O(n + k) \cdot \log n \) whereas the time optimal algorithm achieves \( O(n \log n + k) \) where \( k \) denotes the number of intersections reported.

We aim for two goals which are usually omitted from the literature, namely the ability to handle all degenerate cases and the property of needing a relatively low precision arithmetic for an exact implementation. Milenkovic states in [M 89] that four-fold precision arithmetic is needed for this problem. We lower this bound to three-fold precision.

The text book literature in computational geometry [Me 84, PS 88] does not state how to handle all degenerate cases. Moreover, the description of the plane sweep in the book of Preparata and Shamos [PS 88, p. 284] contains a minor bug: The test in line 21 'if \((s_3 \text{ intersects } s_2) \text{ then } ... \)' should be strengthened to 'if \((s_3 \text{ intersects } s_2 \text{ to the right of } p) \text{ then } ... \)' similarly to line 15 in order to prevent the sweep line from returning to an already processed event. The analogous problem exists in line 22.

In the following we show that the description of an algorithm handling all degenerate cases is only slightly longer but much more useful from the implementor's point of view.

3.5.1 Problem statement

We must state precisely what kind of result we want to compute, since many possibilities exist in the presence of degeneracies. We use the following declarations.

```pascal
3 45
```

```pascal
3.5 Finding all intersections of line segments in the plane

The problem of finding all pairs of intersecting straight line segments in the plane is one of the fundamental problems in computational geometry. Although recently [CE 88] a theoretically time optimal algorithm was discovered, we focus on a variation of the well known plane sweep algorithm [BO 79, Me 84, PS 88] for solving the problem which generalizes the algorithm from the previous section. The plane sweep is of complexity \( O(n + k) \cdot \log n \) whereas the time optimal algorithm achieves \( O(n \log n + k) \) where \( k \) denotes the number of intersections reported.

We aim for two goals which are usually omitted from the literature, namely the ability to handle all degenerate cases and the property of needing a relatively low precision arithmetic for an exact implementation. Milenkovic states in [M 89] that four-fold precision arithmetic is needed for this problem. We lower this bound to three-fold precision.

The text book literature in computational geometry [Me 84, PS 88] does not state how to handle all degenerate cases. Moreover, the description of the plane sweep in the book of Preparata and Shamos [PS 88, p. 284] contains a minor bug: The test in line 21 'if \((s_3 \text{ intersects } s_2) \text{ then } ... \)' should be strengthened to 'if \((s_3 \text{ intersects } s_2 \text{ to the right of } p) \text{ then } ... \)' similarly to line 15 in order to prevent the sweep line from returning to an already processed event. The analogous problem exists in line 22.

In the following we show that the description of an algorithm handling all degenerate cases is only slightly longer but much more useful from the implementor's point of view.

3.5.1 Problem statement

We must state precisely what kind of result we want to compute, since many possibilities exist in the presence of degeneracies. We use the following declarations.

```pascal
3 45
```
Given \( n > 1 \) and the one dimensional array \( S \) of line segments compute the set \( I \) of intersecting pairs which is defined as \( I = \{ (i, j) : S[i] \text{ intersects } S[j] \text{ and } 1 \leq i < j \leq n \} \). Two straight line segments intersect iff their intersection as point sets in \( \mathbb{R}^2 \) is not empty. Note that under this definition overlapping or touching line segments do intersect. See lemma 2 in section 2.1 for a routine computing the intersection.

### 3.5.2 The plane sweep algorithm

We assume some familiarity with the fundamental plane sweep algorithm given in [BO 79, Mc 84, PS 88] since we concentrate on an implementation that handles degenerate cases. The following program solves the problem.

```plaintext
type
event = record where: point;
    kind: (start, intersection, end); k, l: lineSegment end;
intersection = [i, j: 1 .. n];

var
xQueue: priorityQueue of event;
yTable: dictionary of lineSegment;
I: dictionary of intersection;

begin  { Of algorithm 'AllIntersect' }  
xQueue := \emptyset ; yTable := \emptyset ; I := \emptyset ;
for i := 1 to n do
    xQueue.insert ([S[i].p, start, S[i], nil], <ev) ;
xQueue.insert ([S[i].q, end, S[i], nil], <ev)
end;
while xQueue \neq \emptyset do
    e := xQueue.extractMinimum(<ev)
    case e.kind of
        start:
            yTable.insert (e.k, <ls);
            p := yTable.pred(e.k, <ls); s := yTable.succ(e.k, <ls);
            xQueue.delete ([intersect[p, s], intersection, p, s]);
            checkIntersection(p, e.k); checkIntersection(e.k, s)
        intersection:
            p := yTable.pred(e.k, <ls); s := yTable.succ(e.l, <ls);
            xQueue.delete ([intersects(e.l, s), intersection, e.l, s]);
            xQueue.delete ([intersect(p, e.k), intersection, p, e.k]);
            checkIntersection(e.k, s); checkIntersection(p, e.l);
            yTable.swap(e.k, e.l);
            I.insert ([min(e.k.id, e.l.id), max(e.k.id, e.l.id)])
        end:
            p := yTable.pred(e.k, <ls); s := yTable.succ(e.k, <ls);
            checkIntersection(p, s); yTable.delete(e.k)
end { case } end { while } end; { 'AllIntersect' }
```
procedure checkIntersection(k, l: lineSegment);
begin
  if intersect(k.p, k.q, l.p, l.q) ≠ nil ∧ ¬I.member([min(k.id, l.id), max(k.id, l.id)]) then
    xQueue.insert([intersect(k.p, k.q, l.p, l.q), intersection, k, l]);
end;

function <ev(a, b: event): boolean;
begin
  <ev := (a.where <lex b.where) ∨ ((a.where = b.where) ∧ (a.kind < b.kind))
end;

function ws(p, q, r: point): number; { whichSide }
begin
  ws := (p.x - q.x) * (q.y - r.y) - (p.y - q.y) * (q.x - r.x)
end;

function <ls(k, l: lineSegment): boolean; { Precondition: The sweep line focus is at k.p. }
begin
  <ls := ws(l.p, k.p, l.q) > 0 ∨ ws(l.p, k.p, l.q) = 0 ∧ ws(l.p, k.q, l.q) < 0 ∨ ws(l.p, k.p, l.q) = 0 ∧ l.p <lex k.p
end;

function intersect(p, q, r, s: point): point; { Precondition: p <lex q ∧ r <lex s. Computes the lexicographically smallest intersection point of the segments pq and rs or 'nil', if none exists. }
begin
  prq := ws(p, r, q); psq := ws(p, s, q);
  rps := ws(r, p, s); rqs := ws(r, q, s);
  if (prq ≠ 0) ∧ (psq = 0) then { pq and rs lie on the same line. }
    if min(q, s, <lex) <lex max(p, r, <lex) then
      intersect := nil { pq and rs do not overlap. }
    else intersect := max(p, q, <lex)
  elsif (sign(prq) ≠ sign(psq)) ∧ (sign(rps) ≠ sign(rqs)) then
    intersect := [prq * s.x - psq * r.x, prq * s.y - psq * r.y] / (prq - psq)
  else intersect := nil
end;

This program description gives the necessary details for a correct implementation which handles all configurations. In the following we convince ourselves that all conceivable degenerate cases are handled correctly. The three major problems are vertical segments, coinciding events and overlapping segments.
3.5.3 Handling degenerate cases

Vertical segments

In order to process vertical segments we abandon the notion of a rigid vertical sweep line and replace it by a topological sweep line [EG 86]. Its location is specified by the position of the focus \( f \) with the meaning that all events \( e \) with \( e \prec_{\text{lex}} f \) have been processed.

![Figure 3.7: Sweeping with a topological sweep line with focus \( f \)](image)

Unfortunately the lexicographic order of events is not sufficient as the case of coinciding events, i.e. events with identical x- and y-coordinates, shows.

Coinciding events

The following case is the most general one: At some point \( p \), many segments start, many segments intersect and many segments end. We solve this problem by first processing the start events in any order, then by processing the intersection events according to the method described in section 2.5 (the topological argument) and finally by processing the end events again in any order. The order ‘start before intersection before end’ is justified by the observation that for any fixed single segment \( s \) its start event must come before its end event and an intersection event with \( s \) cannot occur before \( s \) has started or after \( s \) has ended. Coinciding start events can be processed in any order because the order of the y-table does not depend on the sequence of insertions but on the slope of the segments.

The method from section 2.5 guarantees that all intersecting pairs of segments are found and that the order in the y-table is properly reversed for the participating segments.

![Figure 3.8: Sweeping a multiple intersection point](image)

This is why the program from the previous section needs the dictionary \( I \) which contains the
intersecting pairs found so far. In the ordinary plane sweep program, the decision whether a detected intersection must be inserted into the x-queue or discarded is solely based on the x-coordinate of the intersection point: The intersection is inserted iff it lies to the right of the sweep line. In degenerate cases this test is too weak: We might process an intersection event and produce an intersection event at the same location and we could not tell by this criterion whether it should be discarded or not. Using the dictionary $I$ instead gives us the required information: An intersection event is generated iff this intersection is not present in $I$.

Since we have determined all intersecting pairs we can remove the segments ending at this common intersection point in any order. Removing a segment will generate an intersection test but no new intersection coinciding with the end point we currently process.

**Overlapping segments**

So far we have tacitly assumed that segments do not overlap. In this section we study the consequences of removing this assumption. The sweep invariant requires that the segments occur in the y-table in the same order in which they are intersected by the sweep line. Consequently, overlapping segments are always direct neighbors in the y-table although their relative order among each other changes when intersection events are processed.

The only difficulty with overlapping segments is the correct treatment of the corresponding start events, since after a correct y-table is established, intersection and end events maintain this invariant. Assume that $s_1, s_2, ..., s_m$ and $s$ are pairwise overlapping segments and that $s_i <_{ls} s_{i+1}$, $1 \leq i < m$, in the y-table. We now face the problem of correctly inserting segment $s$. In the case that the intersection between $s_i$ and $s_j$ for some $i$ and $j$, $i < j$, has already been processed we must require that either $s_i <_{ls} s$ and $s_j <_{ls} s$ or $s <_{ls} s_i$ and $s <_{ls} s_j$. Inserting $s$ between $s_i$ and $s_j$ misses either the intersection of $s$ with $s_i$ or the intersection of $s$ with $s_j$. Therefore we define $<_{ls}$ in such a way that a segment with a lexicographically larger starting point is inserted above (or below) all segments it overlaps. This concludes the treatment of all possible degeneracies which can occur in this plane sweep algorithm.

**3.5.4 Arithmetical requirements and how to decrease them**

So far we have assumed infinite precision arithmetic. In the case where the endpoints of the segments are described as pairs of $m$-bit integers, we implement infinite precision arithmetic as rational arithmetic since the depth of the arithmetical computation is bounded; in fact the most complicated computation determines the order in which two intersection events must be processed. According to lemma 5 in section 2.1 the numerator of the x- or y-coordinates of an intersection point needs $3m + 3$ bit integers while the denominator needs $2m + 2$ bit integers giving us a total requirement of $5m + 5$ bit integers for comparison purposes. This five fold precision agrees with Beretta’s analysis in [Be 84]. Also note that claims about the required precision are basically claims about the magnitude of intermediate expressions and not meant to be in a complexity theoretical sense.

In section 2.5 we have given a topological argument demonstrating that consecutive intersection events can be processed in any order as long as the following two requirements
are met: 1) They are not interrupted by start or end events, and 2) they are intersections of segments which are currently neighbors in the y-table. We meet the first requirement by placing the start and end points of the segments on an integer grid and the second requirement by using the algorithm from section 2.5 which we have already incorporated into our plane sweep algorithm. Assuming integer coordinates in the range from 0 to $2^m - 1$, the function ‘intersect’ is simplified as follows:

```plaintext
function intersect (p, q, r, s: point): point;
{ Precondition: $p < lex q \land r < lex s$. Computes the lexicographically smallest intersection point of the segments pq and rs or ‘nil’, if none exists. }
begin
    prq := ws (p, q, r); psq := ws (p, s, q);
    rps := ws (r, p, s); rqs := ws (r, q, s);
    if (prq = 0) a (psq = 0) then { pq and rs lie on the same line. }
        if min(q, s, <lex) <lex max(p, r, <lex) then
            intersect := nil { pq and rs do not overlap. }
        else intersect := max(p, q, <lex) end
    elsif (sign (prq) * sign (psq)) a (sign (rps) / sign (rqs)) then
        D := prq - psq; X := prq*s.x - psq*r.x; Y := prq*s.y - psq r.y;
        intersect := [(X div D, sign (X mod D)), (Y div D, sign (Y mod D))]
        { Coordinates are represented as pairs of integers $(a, b)$ where $a$ denotes the ‘floor’ of the exact value. $b = 0$ if the exact value is an integer and $b = 1$ otherwise. }
    else intersect := nil end
end;
```

We have replaced the exact division operator ‘/’ by the inexact ‘div’-operator. In addition we must use an extra bit in order to preserve the information whether the rounded result is exact or not. Numbers of the form $(a, b)$ with $a \in \{ 0 ... 2^m - 1 \}$ and $b \in \{ 0, 1 \}$ are compared lexicographically:

$$(a_1, b_1) < lex (a_2, b_2) \iff (a_1 < a_2) \lor ((a_1 = a_2) \land (b_1 < b_2))$$

This extra bit is necessary to guarantee that rounded intersection events are processed in their correct order, otherwise start and end events might interfere with them. A simple way to implement this additional bit is to require that all given coordinates are even numbers. Computed intersection points only have even coordinates if they are exact.

This rounding decreases the precision from five fold to three fold precision. We must still be able to compute the numerators of the coordinates of the intersection point. As a conclusion we have shown the following theorem:

**Theorem 8 (three fold precision arithmetic suffices for finding all intersections):**
Let $S = \{ s_1, ..., s_n \}$ be a set of $n$ straight line segments in the plane which do not degenerate into points. The coordinates of the end points are $m$-bit integers in the range from 0 to $2^m - 1$. Then the plane sweep algorithm ‘AllIntersect’ with the modified ‘intersect’ function computes the set $I = \{ (i, j): (1 \leq i < j \leq n) \land (s_i \cap s_j \neq \emptyset) \}$ in time $O((n + 1) \log n)$ using $3\cdot m + 3$ bit integer arithmetic in all (degenerate) cases.

**Proof:** Follows from the preceding discussion.
4 The XYZ GeoBench: A workbench for geometric computation

4.1 Goals of the project

We identify three major goals of the XYZ GeoBench.

A programming environment
We provide the implementor of geometric algorithms with the necessary infrastructure for rapid prototyping. The ingredients for such an environment are a rich set of geometric primitives, a fundamental set of basic geometric algorithms, a collection of abstract data types and the ability to perform universal operations, like input/output or scaling, on geometric objects.

An interactive testbed for experiments
Experimental data concerning the performance of geometric algorithms is rarely available. We wish to measure an implementation’s efficiency by comparing it to other programs solving the same problem. Furthermore we like to experiment with different implementations of abstract data structures and to try different models of arithmetic. We need to construct degenerate configurations as test cases and save them in a test suite.

Algorithm animation
Most geometric algorithms can be animated fairly easily since geometric objects have clear standard graphical representations. Algorithm animation is used for demonstration in the classroom and for debugging.

4.2 System structure and class hierarchy

This section gives an overview of the GeoBench system structure. We assume basic familiarity with the terminology of object oriented programming, namely the notions of class, abstract class, method, inheritance, instance variable and overriding which can be found for example in [Mey 87]. Our presentation uses Object Pascal, the implementation language of the GeoBench.

The backbone of the GeoBench is a common class hierarchy which defines all the necessary data types and describes the ‘is_a’ relationships among them. All algorithms are methods associated with the class on which they operate. For example the computation of the Voronoi diagram is a method in the class ‘pointVector’ yielding an object of type ‘voronoiDiagram’. Inheritance guarantees that methods for a certain class are also available for its descendants. Therefore the Voronoi diagram can also be computed for the vertices of a ‘polyLine’, of a ‘polygon’ and of a ‘convexPolygon’. The following figure shows the tree of the most important classes found in the GeoBench.
The purpose of abstract classes, that is classes only describing a common interface for derived classes, will be explained later (see section 4.3, 4.4 and 4.6).

The class hierarchy serves as the glue holding together the different components of the GeoBench: Universal operations, parametric arithmetic, geometric primitives, abstract data types and user interface. The following figure shows the relationship between the components: The arrows symbolize the 'is_based_on' relationship. The XYZ Library is implemented on top of the GeoBench and will be described in section 5.1.
The following sections describe the components in more detail. We explain and justify design decisions when appropriate and show solutions to implementation problems.

4.3 Universal operations

One of the benefits of object oriented programming is the ability to specify a common behavior for a large set of objects. For example, we define a method for all geometric objects to display themselves on the screen. We call operations like this 'universal operations'. They must be specified in the root class 'obj'. They exist for the following areas: Memory management, interactive input/output, file input/output, geometric transformations, type computations, random instances, class description and method execution.

4.3.1 Memory management

We provide methods for the creation, destruction and duplication of objects. When creating an object, the difficulty arises that for its proper initialization additional parameters might be necessary, e.g. the length of a dynamic array. In this case we specify that the parameterless initialization method 'init', which must always be called after an object is created, may use additional global variables as implicit parameters.

A second problem is the treatment of objects that contain other dependent objects either explicitly as instance variables or implicitly like in a list object. The effect of destruction or duplication of such an object is unclear since the dependent objects might be destroyed or duplicated or not. We solve this problem by explicitly stating at each object declaration for an object o whether the dependent objects of o belong to the internal state of o or not. Dependent objects belonging to the internal state are destroyed and copied just like regular instance variables whereas dependent objects that do not belong to the internal state are left intact. The following example shows the difference.
lineSegment = object (obj)  { Derived from the root class ‘obj’ }  

p, q: point  
{ ‘point’ is a class. ‘p’ and ‘q’ belong to the internal state of a ‘lineSegment’. }  
end;

vector = object (obj)  { Derived from the root class ‘obj’ }  

length:  0 .. maxLength;  { Length of the vector }  
elements: array [1 .. maxLength] of obj  
{ Elements of the vector do not belong to the internal state. }  
end;

The decision whether a dependent object belongs to the internal state or not is a purely pragmatic one and must be documented when declaring such an object in order to specify the semantics of destruction and duplication. In the case of ‘lineSegment’ we prefer completely independent copies whereas in the ‘vector’ example a (recursive) copy of the vector’s elements is usually undesirable. We provide additional methods for cases where the recursive destruction or duplication of all dependent object is required.

The interface is given by

type obj = object  { The root object }  
procedure init;  { Initializes the internal state. }  
procedure free;  { Reclaims all memory belonging to the internal state. }  
procedure freeAll;  
{ Reclaims all memory belonging to the internal state and recursively calls ‘freeAll’ for all dependent objects. }  
function duplicate: obj;  
{ Creates an object having a duplicate of the internal state. }  
function duplicateAll: obj;  
{ Creates an object having a duplicate of the internal state and calls ‘duplicateAll’ recursively for all dependent objects. }  
end;

As a result the following axioms hold:

\[ z := \text{freeMemory()}; \text{new(a); a.init; a.free}; \]  
\{ Postcondition: \( z = \text{freeMemory()} \) \}

\[ z := \text{freeMemory()}; \text{new(a); a.init;} \]  
\text{b := a.duplicate; b.free; a.free;}  
\{ Postcondition: \( z = \text{freeMemory()} \) \}

\[ z := \text{freeMemory()}; \text{new(a); a.init;} \]  
\text{b := a.duplicateAll; b.freeAll; a.free;}  
\{ Postcondition: \( z = \text{freeMemory()} \) \}

4.3.2 Interactive input/output

Interactive use of the GeoBench requires that all geometric objects can display themselves on the screen and that they can be created using input devices like a mouse. The following declarations specify the interactive input/output facilities.
type
  obj = object { The root object }
procedure construct (px, py, newX, newY: integer);
  { Constructs the object using the coordinates (px, py) of the first mouse click and
    the current mouse position (newX, newY). This works only for objects with at
    most four degrees of freedom. }
procedure windowOutput;
  { Draws the object on the screen. We assume XOR graphics, so that drawing and
    erasing are the same procedure. }
procedure windowInput (px, py: integer);
  { Like ‘init’ but the internal state is initialized using the mouse via dragging.
    (px, py) are the coordinates of the first mouse click, i.e. where the user initiated
    the construction process. This method is implemented in terms of ‘construct’ and
    ‘windowOutput’ and need not be overridden if ‘construct’ is implemented. }
end;

The method ‘windowInput’ serves as an example on how we abstractly implement the
concept of dragging. Dragging means that the user moves the mouse while a mouse button
remains pressed. The following procedure implements object input by dragging and is based
on ‘construct’ and ‘windowOutput’.

procedure obj.windowInput (px, py: integer);
var
  x, y, oldX, oldY: integer;
begin
  init;
  construct(px, py, px, py);  { Display initial instance. }
  windowOutput;  { Display initial instance. }
  oldX := px; oldY := py;  { Save current mouse position. }
  while Button do begin  { While mouse button is pressed }
    getMouse(x, y);  { Get new coordinates of the mouse. }
    if (x/oldX) or (y/oldY) then begin  { Mouse has moved. }
      windowOutput;  { Erase old instance (XOR graphics). }
      construct(px, py, x, y);  { Build new instance. }
      windowOutput;  { And display it. }
      oldX := x; oldY := y
    end end;
  windowOutput  { Input finished, erase drawing. }
end;

This example shows an advantage of object oriented programming where common code is
factored out if suitable primitives are introduced. Whenever ‘windowOutput’ and ‘construct’
are implemented according to their specifications, the method ‘windowInput’ is available.

As a typical example we give the implementations of ‘windowOutput’ and ‘construct’ for the
class ‘rectangle’. 
type rectangle = object (obj)
    left, bottom, right, top: integer;
    { Object invariant: (left, bottom) and (right, top) are opposite corners of the
      rectangle \( \text{left} \leq \text{right} \land \text{bottom} \leq \text{top} \). }
procedure windowOutput; override;
procedure construct (px, py, newX, newY: integer); override;
end;

procedure rectangle.windowOutput;
begin
    MoveTo(left, bottom);  { Moves the pen to (left, bottom) }
    LineTo(right, bottom); LineTo(right, top);
    LineTo(left, top);   LineTo(left, bottom)
end;

procedure rectangle.construct (px, py, newX, newY: integer);
begin
    if px < newX then left := px; right := newX
    else left := newX; right := px end;
    if py < newY then bottom := py; top := newY
    else bottom := newY; top := py end
end;

Besides the methods described, the GeoBench provides facilities for graphics in a different
coordinate system, for highlighting an object and for flashing an object. The latter two
methods are useful in algorithm animation and for visual feedback when an object is
selected.

4.3.3 File input/output

When processing geometric objects, we are interested in saving them permanently on
secondary storage. File input/output becomes therefore another area where operations
applicable to all objects must be provided. As file format we normally use a byte stream
which contains geometry data and some type information in a prefix format. The introduction
of lists and arrays makes this format recursive. A second file format provides a human
readable LISP like notation which is useful for debugging or for data exchange with other
programs. An EBNF definition is given in the appendix.

type obj = object
    procedure fileInput;
    { The complete object is read from the byte stream ‘currentStream’, the first byte
      on ‘currentStream’ is the first byte of the internal state. }
    procedure fileOutput;
    { The complete object is written as a stream of bytes to the byte stream
      ‘currentStream’. }
    procedure fileOutputTag;
    { A byte identifying the object’s type is written before calling ‘fileOutput’. }
procedure textInput;
procedure textOutput;
procedure textOutputTag;
{ text<method> is like <file>method, but a human readable format is used. }
end;

In cases where we know beforehand what type of object to expect on the byte stream, we just declare a variable of the appropriate type, allocate memory for it using 'new' and call 'fileInput'. A difficulty arises when we do not know the next object's type. In this case we require the next byte on the byte stream to be a type identifier, written by 'fileOutputTag'. For the transformation of the type identification byte into an object which 'understands' 'fileInput', we use a table called 'prototypes' which contains for each possible type identifier an object of the appropriate type. The following declarations and code fragments clarify this point.

type objectType = (... , point2dType, rectangleType, ..., objType);
{ An enumeration type having an entry for each class of the form <class>Type. }
var prototypes: array [objectType] of obj;
{ prototypes[<class>Type] is an object of type <class>. The entry for prototypes[<class>Type] is supplied by the initialization code in the module defining the class <class>. }

The function 'readNextObject' reads from 'currentStream' an object of unknown type.

function readNextObject: obj;
var b: byte; o: obj;
begin
readByte(b); { Read the next byte from 'currentStream'. }
o := prototypes[objectType(b)].copy;
{ Interpret 'b' as a type identifier and create an uninitialized object of the appropriate type. }
o.fileInput; { Initialize the object from 'currentStream'. }
readNextObject := o
end readNextObject;

A file created by the GeoBench additionally contains the coordinates of its display window and an identification number at the beginning of the file which distinguishes files containing geometry data from other data files.

4.3.4 Geometric transformations

We provide the following geometric transformations: Translation, rotation, scaling and mirroring of a geometric configuration with respect to a given line. The interface is given by the following declarations.

type obj = object

procedure translate (dx, dy: real);
{ Translates an object by the vector (dx, dy). }
procedure rotate (xOrigin, yOrigin, alpha: real);
  { Rotates the object by 'alpha' (in radian) around the point (xOrigin, yOrigin). }
procedure scale (lambda: real);
  { Scales the object by 'lambda'. }
procedure reflect (a, b, c: real);
  { Mirrors the object with respect to the line \( a \cdot x + b \cdot y = c \), \( a \neq 0 \lor b \neq 0 \). }
end;

4.3.5 Type computations

File input/output has shown the importance of an object’s type for permanent storage. We must also know an object’s type in order to decide which geometric operations are legal.

Object Pascal does not allow any computations with the types of objects besides the ‘member’ test which finds out whether an object is a member of a given class. As a solution, mentioned in section 4.3.3, we have introduced an enumeration type which orders the types in the class hierarchy and allows the necessary operations (e.g. testing whether two objects have the same type).

```pascal
type
  objectType = (... point2dType, rectangleType, ..., objType);
  { An enumeration type having an entry for each class of the form <class>Type. }

obj = object
  function memberType (o: obj): boolean;
  { Tests whether 'o' is a descendant of 'self'. By 'self' we mean the object that receives the method call. }
  function getType: objectType;
  { Get 'self's type. }
  function getTextType: str255;
  { Get a textual description of 'self's type. }
end;
```

A textual description of the type is used for debugging and for giving feedback to the interactive user about the kind of object created or selected.

4.3.6 Random instances

Generation of random instances is used for the rapid creation of test data. We provide a method that changes the internal state randomly, preserving the object invariant. The global variable ‘currentRandomConstraint’, usually a rectangle, specifies the boundaries in which the random change takes place. The interface is:

```pascal
type
  obj = object
    procedure randomChange;
end;
```
4.3.7 Class description and method execution

Each class should be able to describe itself to the user, meaning that given a list of arbitrary objects, a class should deliver all the methods it can execute on these objects in human readable form. Assuming a list of three points, the 'point2d' class should offer a method for creating a circle, the 'pointVector' class should offer a method for computing a closest pair and a Voronoi diagram and so on. After all classes have given such a description, it must be possible to execute a specific method of a specific class on the given list of objects.

```pascal
type obj = object
  procedure describe;
  { Computes the menu of operations applicable to the list of objects in the global variable 'currentArguments'. }
  procedure execute (item: integer);
  { Executes the selected method 'item' with arguments in 'currentArguments' and appends the result to the global variable 'currentResult'. }
end;
```

4.3.8 Miscellaneous

Certain universal operations do not fit into any of the categories mentioned so far:

1) Deciding whether a geometric object has been selected by a mouse click
2) Testing two objects for equality
3) Comparing two objects according to some canonical order.

We base 2) and 3) on the object's internal state and not on memory addresses. For example two points are considered equal whenever they have the same coordinate values, even if their corresponding objects occupy distinct memory locations. Operations 2) and 3) are typically used for detecting duplicate objects a user might have created.

```pascal
type obj = object
  function isSelected (where: rectangle): boolean;
  { Determines whether the object is selected by the given rectangle 'where'. }
  function equal (o: obj): boolean;
  { Determines whether the internal state of 'self' is identical to the internal state of 'o'. }
  function genericLessThan (o: obj): boolean;
  { Determines whether 'self' is less than to 'o' where 'o.getType = self.getType'. 'genericLessThan' specifies a total order among objects of the same type and is used to eliminate duplicates after sorting. }
end;
```

4.4 Interchangeable arithmetic and parameterized floating point arithmetic

We have seen in chapters 2 and 3 that the choice of arithmetic may have a significant impact
on how an implementation of an algorithm behaves, in particular for degenerate or nearly
degenerate configurations. Since we want to experiment with different arithmetics, easily
interchangeable arithmetic is needed. This means that in the best case no line of code of an
implementation must be modified in order to try out a different model of arithmetic. Since
points are a basic building block of geometry, we achieve this goal by defining an abstract
'point2d' class which has no instance variables for the coordinates but specifies an interface
with access procedures to the coordinates and various geometric primitives (see also the
following section 4.5).

```
point2d
   abstract, has no instance variables,
   only interface description

realPoint
longIntPoint
floatPoint
   concrete, has instance variables,
   implements required operations
```

Figure 4.3: The abstract class 'point2d' and its descendants

From this abstract 'point2d' we derive concrete point objects having instance variables and
implementing the geometric primitives in their respective arithmetic. Algorithms using only
the functions and procedures specified by the abstract type 'point2d' can be run in any of the
three kinds of arithmetic currently supported.

In order to study not only the built-in floating point arithmetic (as used in 'realPoint' where
the x- and y-coordinates are of type real) we have implemented a software floating point
package with arbitrary base (including odd bases) and precision which is used for the
coordinates of the object 'floatPoint'. The idea here is of course not to simulate high
precision floating point arithmetic but on the contrary low precision arithmetic in order to
make rounding errors and other problems of floating point arithmetic more pronounced.

4.5 Geometric primitives

The type 'point2d' (section 4.4) is the basic building block of our geometry. Therefore all
geometric primitives are methods in this class and are usually implemented in three different
ways taking advantage of the respective arithmetic. The overwhelming part of our library is
based on the following primitives.

```
function whichSide(p, q, r: point2d): (-1, 0, +1);
{ Determines on which side of the directed line segment pq the point r lies. }
function crossProduct (p, q, r, s: point2d): real;
{ Computes the cross product (p-q)x(s-r). }

function distance(p, q: point2d): real;
{ Computes the Euclidean distance between p and q. }
function squaredDistance (p, q: point2d): real; \( (p_x - q_x)^2 + (p_y - q_y)^2 \)
function squaredDx (p, q: point2d): real; \( (p_x - q_x)^2 \)
function squaredDy (p: point2d): real; \( (p_y - q_y)^2 \)

function circleCenter (p, q, r: point2d): point2d;
{ Computes the center of the circle through p, q and r. }
```
function intersectLineSegment(p, q, r, s: point2d):
    [point2d, point2d, boolean];
    { Tests whether the segments pq and rs intersect and computes the intersection segment. }

function xPlusY(p: point2d) : real; { px + py }
function xMinusY(p: point2d) : real; { px − py }

In total we need about 15 primitives for all geometric algorithms implemented so far and 'whichSide' turned out to be the most common one.

4.6 Abstract data types

Efficient geometric algorithms organize data in lists, dictionaries, queues, etc. We describe design decisions and implementation details for some abstract data types.

4.6.1 The abstract data type 'sequence'

The abstract class 'sequence', realizing collections of arbitrary objects, is implemented as a linked list structure and as a (dynamic) array. The list implementation is useful when no a priori bound on the number of elements is known and sequential processing of the elements is feasible. The array implementation provides much greater flexibility and functionality.

As an implementation detail we mention the use of procedure parameters that are themselves procedures or functions. We consider two applications, the 'forAll' construct and sorting. Consider the following declarations:

type sequence = object(obj)
    length: seqIndexO; { Number of elements in the sequence. }
    procedure forAll (procedure whatToDo (x: obj));
        { Perform 'whatToDo' for all objects in the sequence. }
    procedure sort (function lessThan (p, q: obj) : boolean;
        1: seqIndex; r: seqIndex0);
        { Sorts the sequence between 'l' and 'r' in place using 'lessThan'. }
end;

We use 'forAll' mainly for broadcasting method calls to all elements of a sequence, as illustrated by the following example that translates a sequence by translating all its elements.

procedure sequence.translate (dx, dy: real);
    procedure translateP (x: obj);
    begin x.translate(dx, dy) end; { Access to parameters (dx, dy) is possible. }
begin forall(translateP) end;

A second application are comparison functions, like 'lessThan' in the previous example. They are more flexible than providing a universal 'lessThan' method, implemented in each class, since the same kind of objects often needs to be compared in different ways. Consider
the case of two points \( p \) and \( q \) where there are at least four different ways for defining 'lessThan':

\[
\begin{align*}
  p < q & \iff p_x < q_x & \{ \text{Compare only x-coordinates.} \} \\
  p < q & \iff p_y < q_y & \{ \text{Compare only y-coordinates.} \} \\
  p < q & \iff p_x < q_x \lor (p_x = q_x) \land (p_y < q_y) & \{ \text{Lexicographic order: x before y} \} \\
  p < q & \iff p_y < q_y \lor (p_y = q_y) \land (p_x < q_x) & \{ \text{Lexicographic order: y before x} \}
\end{align*}
\]

4.6.2 The reference concept

In the context of key-based data structures the reference concept helps us distinguish between operations that require a key and those that change a data structure without needing a key. A typical example in a dictionary is 'find' for the former kind of operation and 'swap' for the latter kind.

We motivate this distinction with a problem that occurs when implementing the plane sweep algorithm for finding all intersecting line segments.

![Figure 4.4: The difficulty of accessing line segments by their y-value](image)

The line segments form a total order along the front and their key value is essentially their y-value when evaluated at the position of the front. At position \( x_2 \) we must exchange \( s \) and \( t \), which is difficult using their key value: There are two objects with the same key value and we might access the same segment twice.

As a solution, we prohibit key based access in this case. Instead, we provide means for performing the required 'swap' without using key values. We associate with each object \( o \) in the data structure a unique reference that is supplied whenever \( o \) is known to participate in an operation. For easier understanding, imagine a reference to be a pointer into a data structure although the internal representation of a reference is hidden from the user. The idea of a reference resembles the notion of items introduced in LEDA [MN 89] as an abstraction of pointers and locations.

4.6.3 The abstract data type dictionary

In a dictionary based on the reference concept 'find' is the only key-based operation. The operation 'find' takes an object \( o \) to be found and a key comparison function and returns, if possible, a reference to an object with the same key value. If none is found, 'find' returns a
direction $d$ and a reference to an object $p$ such that $o$ can be inserted as $p$'s direct neighbor in direction $d$. Performing the insertion yields a reference for $o$. All other operations like 'delete' or 'swap' require references as arguments.

This separation between key-based operations and operations changing the data structure has two advantages: 1) Different objects with identical key values are easier to handle (see the example in 4.6.2) and 2) operations just changing the data structure can be implemented more efficiently.

If a dictionary is implemented as an AVL-tree, the delete operation takes constant time whenever rebalancing is not needed. In an ordinary implementation we would have to find the object first before we can delete it, making delete a $O(\log n)$ operation most of the time.

The following declarations specify the interface of the abstract data type 'dictionary'.

```plaintext
type directionType = (left, right);
relationObj = (lessThanObj, equalObj, greaterThanObj);
reference = 'integer; [Any pointer]
dictionary = object (obj)
numberOfElements: longInt;
function getObject (where: reference): obj;
{Retrieves the object referenced by 'where'. }
procedure setObject (where: reference; newValue: obj);
{Changes the object referenced by 'where' into 'newValue'. }
function find (x: obj;
function compare (a, b: obj) : relationObj;
var where: reference; var direction: directionType): boolean;
{If 'find' = 'true' then 'where' is the reference to some element 'e' with 'compare(e, x) = equalObj'. if 'find' = 'false' then either 'where' = 'nil' (i.e. the dictionary is empty) or 'x' is a direct neighbor of 'where' in direction 'direction'. }
function insert (x: obj; where: reference;
direction: directionType): reference;
{If 'where' = 'nil' then insert 'x' into the dictionary which must be empty. If 'where' # 'nil' then insert 'x' before ('direction' = left) or after ('direction' = right) 'where'. }
procedure delete (x: reference);
{Removes the object referenced by 'x' from the dictionary. }
procedure swap (p, q: reference);
{Exchanges the elements referenced by 'p' and 'q' in the dictionary. }
function next(x:reference; direction:directionType): reference;
{Finds the predecessor ('direction' = left) or the successor ('direction' = right) of 'x'. }
function extreme (direction: directionType): reference;
{Finds the leftmost ('direction' = left) or the rightmost ('direction' = right) value in the dictionary. }
function isEmpty: boolean;
{Determines whether the dictionary is empty. }
end;
```
The experience with plane sweep algorithms has shown that dictionaries based on the reference concept are simple to use and efficient (see experimental results in section 5.2.3).

The abstract data type dictionary is implemented as an AVL-tree, a sorted list, and a sorted vector. We report experimental performance results for different algorithms using different implementations of 'dictionary' in section 5.2.3.

In the AVL-tree implementation we augment the usual AVL-tree node declaration by a pointer to the father since otherwise we could not reconstruct the path to a given object in the delete operation. Hence we use the following declarations.

```pascal
type balanceType = (leftGreaterRight, leftEqualRight, leftLessRight);
avlNodeP = ^avlNode;
avlNode = record
    value: obj;
    balance: balanceType;
    sons: array [directionType] of avlNodeP;
    father: avlNodeP
end;
```

Son pointers are stored in an array indexed by 'left' and 'right', making it easier to exploit the symmetric structure of many operations necessary for maintaining the AVL condition (e.g. various rotations).

### 4.6.4 A heap with an efficient delete operation

This section shows how to implement a priority queue as a heap using the reference concept such that an efficient delete operation is supported. A heap is a partially ordered binary tree such that the value associated with each internal node is surpassed or reached by any of its children. The figure below shows a heap and its breadth first array representation.

![Figure 4.5: Tree and array representation of a heap](image)

Data structure invariant: \( h[i] \leq h[2-i] \land h[i] \leq h[2+i] \)

The textbook [AHU 83, p. 163] states that a heap does not allow an efficient delete operation and implementors of priority queues with delete have abandoned the heap and used more complicated data structures such as AVL-trees [B 81]. The reference concept is used to add an efficient delete operation to a heap.

The two key observations are 1) any element in a heap can efficiently be deleted if we know its position and 2) we can fix this position as soon as an element is inserted. The following figure shows how this works.
Data structure invariant:
\[ \text{value}[\text{heap}[i]] \leq \text{value}[\text{heap}[2-i]] \land \text{value}[\text{heap}[i]] \leq \text{value}[\text{heap}[2+i+1]] \land \text{ref}[\text{heap}[i]] = i. \]

When an element is inserted into the heap we use the free list of references to assign it a location in the array 'value'. This location is the reference and is never changed. What we change instead is the appropriate double arrow between 'ref' and 'heap' which determines the element's position in the heap. The operation for insertion ('sift' or 'pushDown') is basically the same as for an ordinary heap while the 'delete' operation is a simple generalization of the standard 'deleteMin' operation: The element to be deleted is exchanged with the last element in the heap which is a leaf node. Then we perform the 'pushDown' operation on this sub-heap followed by the 'pushUp' operation reestablishing the data structure invariant. Therefore the cost of delete is \(O(\log n)\) where \(n\) is the number of elements in the heap.

The code is simple. We denote the conditional and by '&'.

```pascal
type relation = (less, equal, greater);
seqIndex = 1 .. maxN; seqIndex0 = 0 .. maxN; reference = ^integer;
HPQ = object (obj) {Heap Priority Queue}
  length: seqIndex0; { Number of elements }
  freeList: seqIndex; { Points to first empty element in 'ref' }
  ref, heap: array [seqIndex] of seqIndex0;
  value: array [seqIndex] of obj;

procedure init; {Establishes the data structure invariant after creation.}
function insert(x: obj);
  function compare (a, b: obj): relation): reference;
procedure delete(x: reference);
  function compare (a, b: obj): relation);
procedure swap(p, q: seqIndex); {Private methods}
procedure pushUp(i: seqIndex);
  function compare (a, b: obj): relation);
procedure pushDown(i: seqIndex);
  function compare (a, b:obj): relation);
end;

procedure HPQ.init; begin
  length:=0; freeList:=1; ref[1]:=0
end;
```
procedure HPQ.swap(p, q: seqIndex); var pm, qm: seqIndex;
begin
  pm := heap[p]; qm := heap[q]; ref[pm] := q; ref[qm] := p; heap[p] := qm; heap[q] := pm
end;

procedure HPQ.pushUp(i: seqIndex; function compare(a, b: obj): relation);
begin
  while (i > 1) & (compare(value[heap[i]], value[heap[i div 2]]) = less) do
    begin swap(i, i div 2); i := i div 2 end
end;

procedure HPQ.pushDown(i: seqIndex; function compare(a, b: obj): relation);
var j, length2: seqIndex0; continue: boolean;
begin
  continue := true; length2 := length div 2;
  while continue and (i <= length2) do
  begin
    j := 2 * i;
    if (j > length) & (compare(value[heap[j]], value[heap[j + 1]]) = less) then
      j := j + 1;
    if compare(value[heap[i]], value[heap[j]]) = greater then
      begin swap(i, j); i := j end
    else continue := false
  end;
end;

function HPQ.insert(x: obj; function compare(a, b: obj): relation): reference;
var i: seqIndex0;
begin
  length := length + 1; { Overflow should be handled here. } i := freeList;
  if ref[freeList] = 0 then
    begin freeList := freeList + 1; ref[freeList] := 0 end
  else
    freeList := ref[freeList];
  insert := reference(i); ref[i] := length; value[i] := x;
  heap[length] := i; pushUp(length, compare)
end;

procedure HPQ.delete(x: reference; function compare(a, b: obj): relation);
var startPosition: seqIndex0;
begin
  startPosition := ref[seqIndex(x)]; swap(startPosition, length);
  ref[seqIndex(x)] := freeList; freeList := seqIndex(x); length := length - 1;
  if startPosition <= length then begin
    pushDown(startPosition, compare); pushUp(startPosition, compare) end
end;
More careful analysis shows that 'pushUp' and 'pushDown' can be implemented more efficiently by remembering the element common to consecutive swap operations. Section 5.2.3 presents experimental studies that confirm the superior efficiency of this implementation of a priority queue compared to an AVL-tree implementation.

A caveat as a final remark: This kind of priority queue cannot be used when the element to be deleted is only known by its key value. In this case we know no efficient way to locate the element in a heap. In our experience implementing plane sweep algorithms, however, this kind of priority queue was sufficient in all cases.

4.7 User interface and algorithm animation

GeoBench uses the Macintosh conventions for window- and menu-handling. The user finds an info window containing useful information, such as available memory, the coordinates of the cursor, time taken by the last operation and the type of the currently selected object. Selecting an object for input can either be done by using the palette attached to each geometry window or by using the objects menu that also allows the creation of a random instance of any of the currently directly accessible objects point, line segment, circle, rectangle, polyline, polygon, convex polygon and $d$-dimensional point.

Computation takes place in geometry windows: The user creates a new one, enters geometric objects, selects them and chooses the desired operation from the operations menu. The operations menu shows only operations which are legal for the selected objects. Performing an operation creates a new geometry window which contains the result of the operation already selected for a subsequent operation. For example one could enter some points using the mouse, select them, compute the Voronoi diagram, compute a Euclidean minimum spanning tree using the Voronoi diagram and finally compute a traveling salesman tour from the spanning tree.

The geometric transformation operations translate, rotate, scale and reflect can be found in the edit menu which also provides commands for changing the viewing transformation: Zoom in and zoom out.

In the animation menu the user selects which algorithms to animate while the arithmetic menu governs which kind of arithmetic to use for newly created objects. Since arithmetic is bound to objects and not to operations, various kinds of conversion operations are available through the operations menu.

Algorithm animation

Algorithm animation is used for demonstration and debugging. We have chosen a simple yet powerful approach to animation. There is only one version of an implementation into which code pertaining to the animation is included via conditional compilation. This code checks whether animation for this particular algorithm is turned on. If yes, it updates the currently visible state of the algorithm and waits for the user to let it proceed. In situations where speed is crucial, we avoid the slight overhead of repeatedly checking whether animation is
turned on by setting the appropriate compile time variable to 'false'.

Animation code has the following general structure.

```plaintext
...  
{ Geometric algorithm changing internal state }
{IFC myAlgAnim }
if animationFlag[myAlgAnim] then
  { Update graphical state information, usually draw some objects. }
  waitForClick(animationFlag[myAlgAnim]);
  { Update graphical state information, usually erase some objects. }
end;
{ENDC }
...
```

The procedure 'waitForClick' provides an interface between the user and the algorithm currently animated. It supports single step mode and a movie mode with user selectable speed (see the 'Animation' dialog box in the previous screen dump). Updating the visualization of the internal state is simplified by the convention that all drawing on the screen is done using XOR graphics which has the benefit that erasing is the same as drawing. Animating an algorithm consists of choosing a representation of the internal state (e.g. position of the sweep line, objects in the y-table, deactivated objects, etc.) and determining appropriate locations in the program where this information needs to be updated. Algorithm animation is implemented for all non-trivial geometric algorithms. Brown and Sedgewick present a more general system for algorithm animation in [BS 84].
4.8 The object oriented approach: Experience and evaluation

In this section we discuss the benefits and shortcomings of implementing a library for geometric computation in an object oriented way.

4.8.1 Dynamic binding and abstract classes

An abstract class is a class of objects with the property that no instances of the class itself are created, but only instances of derived classes. Dynamic binding means that the type of an object and the methods to be executed are determined at run time. Consider the example of the class 'point2d' which has descendants 'realPoint', 'floatPoint' and 'longIntPoint' (see also section 4.4). 'point2d' defines a common interface implemented in three different ways using three different kinds of arithmetic. Programs using only variables of the abstract class 'point2d' can be instantiated at run time to work with all three implementations.

We have used abstract classes and dynamic binding for two purposes: 1) Offering interchangeable arithmetic and interchangeable implementations of abstract data types and 2) factoring out common code. Factoring out common code is best illustrated by the 'windowInput' method from section 4.3.2 which realizes object input via dragging. An abstract class provides a method $M_0$ which is solely implemented in terms of other methods $M_1, M_2, \ldots, M_k$ of the abstract class. As soon as $M_1, M_2, \ldots, M_k$ are implemented in a derived class, a working implementation of $M_0$ is available. The abstract implementation of $M_0$ factors out the common code.

4.8.2 The class hierarchy and inheritance

Object oriented design allows us to take a class and derive new classes with enriched functionality from it, modeling the 'is_a' relation. This relationship is rare among geometric objects, causing the class hierarchy to be relatively flat in most places. The most prominent counterexample is the following: 'convexPolygon' is_a 'polygon' is_a 'polyLine' is_a 'pointVector'.

Inheritance goes hand in hand with derivation: A method $M$ defined in a class is automatically available for each of its subclasses and ideally needs not to be implemented in a different way ('overridden'). This is rarely the case in geometry: The best way for solving the same problem for a class and a derived class can be totally different as the trivial example of computing the convex hull of a polygon and a convex polygon shows.

Having a class hierarchy with a single root has the advantage that universal data types can be created. For example we need a dictionary that can hold any type of object. Even objects of different types should be admissible as long as we can define an order relation. The requirement that a dictionary can hold objects of the type of the root class achieves this goal.

As a conclusion, the concept of a class hierarchy is a useful tool for structuring the library and creating universal abstract data types. It is of less use for saving implementation effort.
4.8.3 Design concepts for classes

Deriving a new class from an existing one should model the ‘is_a’ relation since otherwise the structure of the class hierarchy would be confusing and the semantics of inherited operations could become unclear. This rule determines where to place a new class in the hierarchy. For example the class ‘lineSegment’ should not be derived from ‘point2d’ by adding another point because a line segment is not a point, but is composed of two points. In this case composition is the more appropriate construction principle.

During the evolution of a class hierarchy one often finds two classes which contain some similar methods. An example were the class for linked lists and the class for vectors for which various methods like displaying itself on the screen were similar. In this case we factor out common methods by introducing a common abstract ancestor class which realizes the common behavior.

4.8.4 Implementing plane sweep in an object oriented fashion

Another advantage of object oriented design is the uniform treatment of plane sweep algorithms. We model events as objects, derived from an abstract event class that provides an execute method. The typical structure is given by the following declarations.

```pascal
var xQueue: priorityQueue;
yTable: dictionary;
type event = object
  procedure execute { Abstract method, not implemented here }
end;

event1 = object (event) { Derive the interface from the abstract class. }
  procedure execute; override { Concrete implementation }
end;

event2 = object (event) { Derive the interface from the abstract class. }
  procedure execute; override { Concrete implementation }
end;

{ Other event types }

begin
  initialize xQueue; initialize yTable;
  while not xQueue.isEmpty do
    xQueue.extractMin.execute { Let the event process itself. }
  end;

This structure has the advantage that all code for handling a certain event is concentrated in one place, making the implementation clearer and easier to understand. Furthermore the main program needs no change in the case new events become necessary, e.g. when we transform the sweep that tests for existence of an intersection to the sweep that finds all intersections.
4.8.5 Desirable features of an object oriented language

Although Object Pascal is a small extension to Pascal and supports only the basics of object oriented programming languages, we have experienced the lack of only two relatively minor features.

Treating classes as objects themselves would have facilitated the creation of the persistent object store. We have solved the problem by introducing an enumeration type with an entry for each class and an array with a prototype of each class. For more details see section 4.3.3.

The second problem concerns the lack of a feature similar to the 'like' declaration in Eiffel [Mey 87]. As an example consider the 'duplicate' method in the class 'pointVector' which should create a copy of the object itself. Ideally this duplication also works for classes derived from 'pointVector' that is, we require 'pointVector.duplicate' to create objects of a class completely unknown to 'pointVector'. In Eiffel one can specify that the result is of the same type as 'self' (like 'self') but Object Pascal does not provide this feature. Again we make use of the enumeration type for all classes mentioned in the preceding paragraph: We can enquire the type of an object and create an object of a prescribed type which suffices to simulate the effect of 'like'.

4.8.6 Helpful hints for increasing the reliability of a geometric workbench

In order to create a reliable system we have tried to catch programming errors early. We recommend the following methods which were used successfully.

Show the dynamic memory allocated. The 'info' window contains the number of bytes currently allocated on the heap for dynamic variables (mostly objects). This number gives a first hint whether memory management is working correctly. A constantly growing number of allocated bytes is usually something to worry about.

Use assertions. Especially geometric algorithms often depend on the truth of certain assertions (e.g. the value of some denominator should be different from zero) and a defensive programmer introduces checks (assertions) at appropriate places which give warnings when the assertion fails. Even errors in algorithms can be detected earlier this way.

Data structures should have a method checking their invariant. This is especially helpful while testing non trivial data structures like AVL-trees or even a heap. One tests the data structure by different operations, and after each operation the invariant is checked which increases the confidence in its correctness. Having program code actually check the invariant (as opposed to doing it by hand) also makes automatic testing feasible: One could randomly insert or delete a random element in a dictionary and check the invariant after each operation.

Write and keep test programs for the central data structures. We have written test programs to test the abstract data type 'dictionary' which has the additional advantage that it can be used for all concrete implementations of the type 'dictionary'. The alternative to test programs is to actually use the data structure in an algorithm which has the two disadvantages that 1) in the case of an error one does not know precisely the source of the error and 2) one algorithm
rarely uses all methods offered by a data structure designed for universal applicability.

Create geometric test data and keep it for reference in a test suite. Creating good test data is not trivial and the results of this effort should be kept. A new program solving the same problem might be written, or more often the current implementation changes. In both cases one increases the confidence in the reliability by checking with the test suite. It should contain configurations ranging from no degeneracies at all to multiple ones.

Use algorithm animation for geometric algorithms. Algorithm animation should be used to present an algorithm's essential state information in a graphical way. In the debugging phase, this facilitates the detection of inconsistencies before an incorrect result occurs.

Do not remove the code used for debugging. The code used for debugging should be left in the programs since future modifications might benefit from it. Using conditional compilation removes any run time overhead if necessary.
The XYZ Library: Algorithms and experimental results

We describe the XYZ Library, a library of basic geometric algorithms based on the GeoBench and report the result of various experiments performed with the library programs.

5.1 The XYZ Library

5.1.1 Choice of algorithms

There is no shortage of algorithms for inclusion in a program library for geometric computation. The problem is one of selection, whereby we emphasize the following criteria.

Robustness: A library routine must yield meaningful results for any geometric configuration, including highly degenerate ones. Unlike random data where degenerate configurations are rare, many practical applications generate highly degenerate configurations — degeneracy comes from the regularity inherent in man-made artifacts. We use the methods from section 2, special purpose code for handling degeneracies, high precision arithmetic or we explicitly state an implementation's limitations.

Practical efficiency: We strive for programs that are efficient in practice, that is, outperform competing programs on realistic input data. Example: An optimal algorithm can often be modified to run faster on a battery of test data, even though worst-case optimality is no longer guaranteed. This may occur, for example, by replacing a balanced tree implementation of a dictionary by an array implementation. Thanks to object oriented programming we can postpone such choices and leave them to the user.

Standard problems of geometric computation: A program library is never comprehensive enough to solve most users' problems directly. We limit ourselves to basic problems that serve as building blocks for advanced geometric programs.

Well understood and elegant algorithms: We select algorithms that stand out by virtue of their elegant simplicity and can be implemented in a straightforward manner. Even if they are not asymptotically optimal, these tend to do better than their complicated counterparts with respect to robustness and practical efficiency. Some 'optimal' algorithms are just too complicated for a reliable, robust implementation.

Start with 2-d geometry: The difficulties posed by 2-d problems must be solved completely before venturing into higher dimensions, so we have concentrated on accumulating a sufficient number of representative 2-d algorithms. Just to show that the the structure of the GeoBench is not restricted to low-dimensional space we have implemented an algorithm that computes the minimal area disk enclosing a set of points in d-space.

5.1.2 Implemented algorithms

We structure the collection of implemented algorithms according to the type of objects on which they operate. This conforms to the object oriented, data driven design of the whole
system. Conventionally, a geometric program is a function method that does not modify the object which receives the method call. For the following classes we only state methods that are newly defined for this class; inherited methods are not mentioned explicitly.

**Multiset of points (‘pointVector’)**
- Convex hull (Graham’s scan [G 72], divide and conquer [PH 77])
- Closest pair (sweep line [HNS 88], simplified sweep line [Sc 90a], probabilistic [Ra 76])
- All nearest neighbors (sweep line [HNS 90], simplified sweep line [Sc 90a], extraction from Voronoi diagram [PS 88])
- All nearest neighbors in sector
- Voronoi diagram (sweep line [F 87], divide and conquer [PS 88])
- Euclidean minimum spanning tree [PS 88]
- Traveling salesman (nearest neighbor heuristic, Euclidean minimum spanning tree heuristic [PS 88], convex hull heuristic, tour optimizer)

**Splinegon**
- Convex hull of a splinegon

**Convex polygon**
- Diameter of a convex polygon [PS 88]
- Intersection of two convex polygons [PS 88]
- Tangents common to two convex polygons [PH 77]

**Polygon**
- Spiral intersection
- Winding number [PS 88]
- Boolean operations (union, intersection, difference) on polygons [NP 82]

**Multiset of rectangles (‘rectangleVector’)**
- Contour of rectangles [PS 88]

**Multiset of line segments (‘lineSegmentVector’)**
- Intersection of line segments (Sweep line for 1) existence, 2) reporting 3) horizontal and vertical line segments [PS 88])

**Multiset of $d$-dimensional points (‘dDimPointVector’)**
- Minimal area disk (randomized incremental [W 90], heuristic)

### 5.2 Experimental results

We present the most important types of experiments that are facilitated by the XYZ GeoBench together with experimental results. We show how the efficiency of different algorithms can be compared, how the robustness of algorithms can be assessed using easily interchangeable arithmetic, how implementations for abstract data types can be exchanged and how we support the generation of random and degenerate test data.
5.2.1 Efficiency measurements

The most obvious factor for comparing two different algorithms is their time consumption for specific problems. The asymptotic complexity with its well known O-notation does not provide enough information in order to assess an algorithm's practical efficiency, since the implementation-dependent constant factors are ignored. This often means that competing algorithms must be implemented and compared. The GeoBench times each operation executed automatically in the ‘info’ window.

Using a common platform has the additional advantage that influences common to both implementations get factored out, making timing results more reliable. For example, two algorithms for computing the convex hull will most certainly use some means for determining on which side of a directed line a given point lies. Since this primitive will be implemented the same way in both programs, timing does not depend on some peculiarities of this primitive.

Unless otherwise noted the input data for the following experiments are random points, uniformly distributed in a square. All timing results are in seconds and measured on a Macintosh IIfx. Most figures use logarithmic scales.

Example 1: Sweep or incremental algorithms vs. divide and conquer

We compute the Voronoi diagram by Fortune’s sweep [F 87] and using the divide and conquer algorithm described in [PS 88]. The sweep is about three times faster, easier to implement and uses less memory.

A second example compares the Graham scan [G 72] with Preparata and Hong’s [PH 77] divide and conquer algorithm for computing the convex hull of a set of points. The scan is about two times faster and its source code is about 40% shorter than its divide and conquer counterpart. The handling of degeneracies is more difficult for the divide and conquer algorithm.

\[\begin{array}{|c|c|c|}
\hline
n & \text{Sweep} & \text{D & C} \\
\hline
64 & .733 & 1.767 \\
128 & 1.533 & 4.317 \\
256 & 3.2 & 10.85 \\
512 & 6.733 & 25.383 \\
1024 & 14.35 & 54.933 \\
2048 & 30.283 & 114.133 \\
4096 & 63.85 & \\
8192 & 142.183 & \\
\hline
\end{array}\]
Incremental algorithms, in particular sweeps, are usually superior to divide and conquer algorithms. Divide and conquer algorithms often suffer from two drawbacks not shared by incremental algorithms: 1) They compute too much information that is not part of the final solution and 2) they need more memory, especially in the last merging phase where the final result is obtained from combining two large objects which can often not be done in place.

The conceptual simplicity of divide and conquer rarely leads to an algorithm efficient in practice. The computation of the lower envelope of a set of line segments in the plane [HS 86] seems to be an exception since we do not know an efficient incremental algorithm.

Example 2: The all-nearest-neighbors problem

We compare the sweep algorithm introduced in section 3.3, the sweep for the Voronoi diagram and the projection method mentioned in section 3.2.5. The simple and accurate projection method with its complexity of $\Theta(n^{3/2})$ is superior for point sets up to a few thousand points. For larger sets the direct plane sweep algorithm outperforms both the projection method and the Voronoi diagram method.
Example 3: The closest-pair problem

In this example we compare three different approaches for solving the closest-pair problem: The sweep algorithm from section 3.2, a simplified version of the sweep where we omit the y-table and the probabilistic linear expected time algorithm by Rabin [Ra 76]. Quite unexpectedly, the linear time algorithm is slower, even for moderately large random point sets.

![Figure 5.4: Efficiency comparison of three algorithms for the closest-pair problem](image)

The simplified sweep is always faster than the standard sweep on random point sets. Golin shows in [Go 90] that the simplified sweep on random data takes linear expected time after sorting. We will see in section 5.2.4 that degenerate configurations may reverse this situation, making the probabilistic algorithm more attractive.

From an implementor's point of view the sweep algorithm is by far the simplest. It needs 108 lines of code whereas the probabilistic algorithm needs 188 lines.

Example 4: How simple heuristics can speed up efficient algorithms

For efficient algorithms it pays to look for heuristics to improve efficiency further. We consider two examples: Computing the convex hull and computing the smallest circle (in \(d\) dimensions) that encloses a given point set.

**Computing the convex hull:** In the case of the convex hull, we observe that the standard Graham scan spends most of its time sorting the point set and then incrementally constructing the convex hull. By eliminating points known to be inside the convex hull, we can speed up the process considerably if this test is cheap. In [GS 88] a simple method is given that computes in linear time a rectangle that lies inside the convex hull. Then the points inside the rectangle are eliminated. Performance is greatly enhanced for data sets that are uniformly distributed random points in a square.
In cases where the elimination step does not reduce the work we pay a slight overhead. The following figure shows an overhead of about 20% for the worst case where all input data points end up as convex hull vertices.

---

Computing the minimal area disk: The minimal area disk problem consists of computing the smallest $d$-dimensional disk that encloses a given set of $d$-dimensional data points. Welzl [W 90] gives an elegant, randomized, linear expected time algorithm for solving this problem. In its basic form the algorithm starts with an arbitrary disk having two given points on its perimeter and proceeds by either adding a point, if the point falls inside the disk constructed so far, or by recursively constructing a new disk containing all the points considered so far. Instead of taking an arbitrary disk as the initial one we propose as a heuristic to take the disk through the two points $p$ and $q$ that maximizes $|p_x - q_x|^{d}$ for all dimensions $d$. Intuitively this disk is already a good guess at the final result and will contain many of the remaining data points, reducing the number of recursive calls for the creation of larger disks.
MDHeu is the average number of recursive calls in the method with heuristic improvement whereas MDStand gives the number of recursive calls in the standard randomized algorithm. We experience a slight decrease in recursive calls and time used.

Complexity analysis of the previous minimal area disk algorithm shows that it is linear in the number of points but the constant is exponential in the dimension. The following algorithm gives an approximate solution to the minimal area disk problem in time $O(dn)$. We start with an arbitrary disk and continue by either adding a point if the point falls inside the disk constructed so far, or by creating the smallest disk that contains the disk computed so far and the new data point. The following figure shows the necessary geometry for computing the enlarged disk $E$ from an original disk $D$ and an outside point $p$.

The corresponding algorithm is

Input: $S = \{ p_1, ..., p_n \}$, a set of $n > 1$ $d$-dimensional points.

Output: $D = (r, c)$ a $d$-dimensional disk with center $D_c$ and radius $D_r$ such that $D_r \leq 2D_{min}$, where $D_{min}$ is the minimal area disk containing $S$. 

Figure 5.7: Efficiency comparison of two algorithms computing the minimal area disk of 128 points

Figure 5.8: Computing the smallest disk that encloses a given disk $D$ and a given point $p$
\[ D := \frac{1}{2} \cdot (p_1 + p_2, d(p_1, p_2)); \] (* d is the Euclidean distance. *)

for \( i := 3 \) to \( n \) do
  if \( p_i \notin D \)
    then \[ D := \frac{1}{2} \cdot \left( D_c + p_i + \frac{D_c}{d(D_c, p_i)} \cdot (D_c - p_i), D_c + d(D_c, p_i) \right) \]
end

Lemma 8: Using the notation from above: \( D_r \leq 2 \cdot D_{min} \).

Proof: When the algorithm terminates we have the following situation:

\[ D_{min} \subseteq D, \text{ since } D_{min} \text{ is a convex combination of some points in } S \text{ and ConvexHull}(S) \subseteq D. \text{ Point } p \in S \text{ is the last point to cause a change of } D \text{'s radius. The lemma follows if we show that } D_c \in D_{min} \text{ since } D_r = d(p, D_c) \leq d(p, D_{min}) + d(D_{min}, D) \leq 2 \cdot D_{min}. \text{ Since } \text{ConvexHull}(S) \subseteq D_{min}, \text{ it suffices to show } D_c \in \text{ConvexHull}(S) \text{ by induction on } n. \text{ } n = 2 \text{ is clear since } D_c \text{ is a convex combination of } p_1 \text{ and } p_2. \text{ For the induction step we need only consider the non trivial case } p_1 \notin D. \text{ In this case the new center lies on the straight line segment from } D_c \text{ to } p_1 \text{ which lies inside the enlarged convex hull because } D_c \text{ is inside the old convex hull by induction hypothesis, and } p_1 \text{ is a hull vertex of the enlarged convex hull. } \]

The performance improvement is shown in the following figure and table.

![Figure 5.9: Situation after termination](image)

![Figure 5.10: Efficiency comparison of the exact method and the approximate method for the minimal area disk problem](image)

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Precise</th>
<th>Approximate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>.097</td>
<td>.04</td>
</tr>
<tr>
<td>3</td>
<td>.175</td>
<td>.043</td>
</tr>
<tr>
<td>4</td>
<td>.305</td>
<td>.045</td>
</tr>
<tr>
<td>5</td>
<td>.474</td>
<td>.048</td>
</tr>
<tr>
<td>6</td>
<td>.959</td>
<td>.05</td>
</tr>
<tr>
<td>7</td>
<td>1.675</td>
<td>.053</td>
</tr>
<tr>
<td>8</td>
<td>3.057</td>
<td>.056</td>
</tr>
<tr>
<td>9</td>
<td>14.181</td>
<td>.058</td>
</tr>
</tbody>
</table>
In our experiments we have often experienced errors that were only a few percent off the correct radius, especially in lower dimensions. For the initial disk we also use the aforementioned heuristic.

5.2.2 Parametric arithmetic

The GeoBench allows us to study the effect of various arithmetics on the outcome of a computation since algorithms can be written in an arithmetic independent way. As an example we take the all-nearest-neighbors plane sweep from section 3.3.

![Figure 5.11: Computing with different models of arithmetic](image)

In the leftmost window we have created a random configuration of 12 points whose coordinates are floating point numbers in base 10 with a two digit mantissa. The next window shows the tree of the nearest neighbor to the left relation computed in this floating point arithmetic whereas the third window shows the result computed with 32-bit reals. We compare the two results by superimposing them on each other which cancels the common line segments since we are using XOR graphics. We interpret the result of the comparison in the fourth window such that sometimes wrong nearest neighbors are identified but the distances are not too far off.

5.2.3 Exchangeable implementations for abstract data types and instrumentation

Using the GeoBench, we study the effects of different implementations of abstract data types. A dialog box enables the user to choose the appropriate implementation at run time. It can be used by any program that only uses the abstract classes ‘priorityQueue’ and ‘dictionary’.
The 'Statistics' option provides the user with the maximal number of elements in the data structure and the number of insertions and deletions performed.

**Example: Data structures in the all-nearest-neighbors plane sweep**

In the all-nearest-neighbors sweep from section 3.3 we study the influence of different implementations of the x-queue, a priority queue, and the y-table, a dictionary. As input data we use 8192 random points, uniformly distributed in a square. The times reported are only for computing all-nearest-neighbors-to-the-left. The modified heap described in section 4.6.4 is superior to a priority queue implemented as an AVL-tree. Implementing the y-table as an AVL-tree beats list and array implementations of a dictionary.

![Figure 5.13: Efficiency comparison of different implementations of abstract data types (8192 random points)](image)

The same experiment with only 64 random points yields the same result: The theoretically optimal data structures should be favored.
Figure 5.14: Efficiency comparison of different implementations of abstract data types
(64 random points)

Example: Data structures in the closest-pair plane sweep

We study the influence of the implementation of the y-table on the performance of the closest-pair plane sweep introduced in section 3.2.

Figure 5.15: Efficiency comparison of different implementations of abstract data types in the closest-pair sweep

The array implementation of the y-table dictionary is superior to both competing implementations because the maximum number of elements in the data structure is too small for a sophisticated data structure to be worthwhile.

5.2.4 Test data generation: Random and degenerate configurations

We can create random instances of 2-dimensional points, d-dimensional points, line segments, poly lines, polygons, convex polygons, circles and rectangles. The user specifies the number of objects (for points, d-dimensional points, line segments, circles and rectangles) or the number of vertices (for poly lines, polygons and convex polygons). Random d-dimensional points can be uniformly distributed in a d-dimensional square or a d-dimensional circle. We do not specify the precise meaning of 'random' for complex objects like polygons or even convex polygons because of the difficulties involved.
We can generate the most common degenerate configurations: More than two collinear points, more than three points on a circle, more than two line segments with a common intersection point, horizontal and vertical line segments, point sets that lie on a rectangular grid, point sets with equal x- or y-coordinate, coinciding objects. In integer arithmetic, most of these degeneracies are exact, i.e. the corresponding test polynomials evaluate to zero whereas in the case of floating point arithmetic the test polynomials either evaluate to zero or to a value close to zero making the configuration nearly degenerate.

**Example: The creation of a set of line segments that intersect in one point**

First, we create a set of random points $S$, then we choose the common intersection point $p$ and rotate by 180 degrees a copy $S'$ of $S$ around $p$. Finally we connect the points of $S$ with their respective points in $S'$ yielding a bushel of line segments that intersect in $p$.

![Figure 5.16: Test data generation of a set of line segments that intersect in a common point](image)

**Example: Approximating Voronoi diagrams of more complex objects**

GeoBench can cover line segments, rectangles and circles with evenly spaced points. The resulting configurations are degenerate and are used to approximate Voronoi diagrams of more complex objects as the following figure shows.
Example: How do degenerate configurations influence the performance?

This example illustrates questions like, what is a difficult configuration for an algorithm, how are algorithms influenced by different kinds of configurations, or is a configuration of uniformly distributed random points a typical configuration.

The following diagram shows how the all-nearest-neighbors plane sweep behaves on $n$ points that lie on an $n$-gon, on a vertical line, on a horizontal line or are random points uniformly distributed in a square. Points on an $n$-gon are almost as difficult as random points whereas vertical or horizontal data points are particularly fast to process.

<table>
<thead>
<tr>
<th>$n$</th>
<th>n-gon</th>
<th>Vert.</th>
<th>Horiz.</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>.283</td>
<td>.133</td>
<td>.149</td>
<td>.249</td>
</tr>
<tr>
<td>128</td>
<td>.584</td>
<td>.299</td>
<td>.316</td>
<td>.616</td>
</tr>
<tr>
<td>256</td>
<td>1.266</td>
<td>.584</td>
<td>.617</td>
<td>1.317</td>
</tr>
<tr>
<td>512</td>
<td>2.667</td>
<td>1.267</td>
<td>1.333</td>
<td>2.933</td>
</tr>
<tr>
<td>1024</td>
<td>5.551</td>
<td>2.700</td>
<td>2.833</td>
<td>6.334</td>
</tr>
<tr>
<td>2048</td>
<td>11.616</td>
<td>5.634</td>
<td>5.900</td>
<td>13.217</td>
</tr>
<tr>
<td>4096</td>
<td>24.483</td>
<td>11.950</td>
<td>12.499</td>
<td>27.317</td>
</tr>
<tr>
<td>8192</td>
<td>55.099</td>
<td>25.017</td>
<td>26.417</td>
<td>54.567</td>
</tr>
</tbody>
</table>

The plane sweep for computing the Voronoi diagram shows that uniformly distributed random points are not always the hardest configuration.
The computation for random points is fastest for larger \( n \) while all degenerate configurations take longer. For small \( n \), the Voronoi diagram of horizontal or vertical point sets is computed faster than for random points. The choice of sweep direction, in our implementation the y-axis, causes the difference between horizontal and vertical point sets.

Another aspect concerns the question of how different algorithms are affected by certain types of configurations. As an example we take the closest-pair problem. The sweep algorithm from section 3.2 performs as follows.

The case where all points lie on a vertical line needs more than twice the time of the case with all points on a horizontal. The reason is that for vertical points we essentially sort the data with an AVL-tree whereas for horizontal points we do not need the y-table at all. The case of random points, uniformly distributed in a square, lies between these extremes but is closer to the case of horizontal points, since the size of the y-table quickly decreases as the sweep proceeds.

Rabin's probabilistic algorithm [Ra 76] does not exhibit a similar behavior: Due to randomization, there is essentially no difference between the three types of configurations.
As a final example we show that there might be a difference between random points uniformly distributed in a square versus random points uniformly distributed in a circle. We consider Welzl's randomized algorithm [W 90] for computing the smallest d-dimensional circle enclosing a given set of d-dimensional points. We take 128 points uniformly distributed in a square and in a circle.

We encounter a substantial difference which increases with higher dimensions. The reason is that in the case of random points in a circle, more points lie close to the boundary of the final minimal area disk. Therefore, the in-circle-test fails more often which in turn causes expensive recursive calls. For random points chosen uniformly from a square, relatively few points will lie close to the boundary of the final minimal area disk.
6 Conclusion

6.1 Summary

The XYZ GeoBench, the XYZ Library and three new mathematical techniques for the construction of robust programs are our answer to the problem of how to write good software for geometric computation. The GeoBench is a programming environment for rapid prototyping of geometric programs and supports experimentation extensively. In the algorithms implemented in the XYZ Library we have strived for robustness. New techniques have been used to solve some of the problems of degeneracy, accuracy and robustness.

6.2 Future directions

Our mathematical techniques work well whenever they are applicable. Unfortunately they do not seem to be able to handle all geometric algorithms e.g. the Voronoi diagram. The problem of implementing geometric algorithms with rounded arithmetic is still not completely solved for all practical problems, and additional research in this direction is needed.

We plan to extend the XYZ GeoBench and the XYZ Library in various ways which is simplified by the extensible, object oriented design.

1) Currently we are working on layered objects that approximate 3-dimensional objects in a simple way. Future versions of the GeoBench should expand the 3-d part even more.

2) The grid-file, a multi-dimensional symmetric data structure, will provide access to geometric data on secondary storage. We plan to investigate geometric algorithms that process large amounts of data that do not fit into central memory.

3) Some areas of 2-d geometric computation need additional attention: geometric search structures, aligned (axis-parallel) polygons, triangulations.
Appendix: Syntax of the textual I/O format

In order to exchange geometric data between the GeoBench and other programs, the GeoBench supports a human readable, LISP like notation for most geometric objects. The GeoBench can read and write files according to the EBNF syntax specified below. Axiom of the following grammar is 'List', i.e. the GeoBench expects a list of objects as textual input. For example the string ‘(LST 2 (RPT 50 50) (RPT 100 50))’ is valid input data and represents two points.

Digit = '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'
Natural = Digit {Digit }
Integer = ['-' ] Natural
Real = Integer ['.' Natural] ['E' ['+' '-'] Natural]
Float = Natural Natural Real (* base precision value *)
String = "" Char {Char } "" (* Char & "")

Circle = ('CIR' Real Real Real ')' (* x y radius *)
ConvexPolygon = ('CPL' PointList ')
ColorQuickDrawPicture = ('CQP' Real Real Real Real ')
DGMatrix = 'DCT' Real Real Real
DDimCircle = 'DCI' CoordinateList Real ')' (* coordinates radius *)
DDimPoint = 'DPT' CoordinateList ')
DDimPointVector = 'DPV' DDimPointList ')
DirectedGraph = ('DGR' Vector HomogenVector ')
FloatPoint = ('FPT' Float Float ')' (* x y *)
GraphEdge = ('GED' Integer Integer ')
HomogenVector = ('HVC' HomogenObjectList ')
Int = ('INT' Integer ')
IntegerPoint = ('IPT' Integer Integer ')
Layer = ('LAY' Real Real Vector ')
LayerVector = ('LAV' Real Vector ')
LineSegment = ('LSG' Point Point ')
LineSegmentVector = ('LVC' LineSegmentList ')
List = ('LST' ObjectList ')
MarkedRealPoint = ('MRP' Real Real Str ')
Polygon = ('POL' PointList ')
PolyLayer = ('PLA' Real Vector ')
PolyLayerVector = ('PLV' Real Vector ')
PolyLine = ('PLI' PointList ')
PointVector = ('PVC' PointList ')
QuickDrawPicture = ('QDP' Real Real Real ')
RealPoint = ('RPT' Real Real ')
Rectangle = ('REC' Real Real Real ')
(* left bottom right top *)

(*) startVertex endVertex
(* x y *)
(* x y mark *)
(* dz objects *)
(* z thickness objects *)
RectangleVector = ('RVC' RectangleList)
SimpleUndirectedGraph = ('SUG' Vector HomogenVector)
SpanningTree = ('SPT' Vector HomogenVector)
Spline2 = ('SP2' RealPoint RealPoint)
SplineGon = ('SPL' ObjectList)
Str = ('STR' String)
StraightEdge = ('SED' RealPoint RealPoint)
UndirectedGraph = ('UDG' Vector HomogenVector)
Vector = ('VEC' ObjectList)
VoronoiEdge = ('VED' Integer Point Point [ Point ] [ Point ]
* edgeType p1 p2 [ v1 ] [ v2 ]
* points edges neighbors)
VoronoiDiagram = ('VDG' Point Vector List List)

CoordinateList = Integer { Integer } (* dimension coordinates *)
DDimPointList = Integer { DDimPoint } (* length elements *)
HomogenObjectList = Integer { Object } (* length elements *)
LineSegmentList = Integer { LineSegment } (* length elements *)
ObjectList = Integer { Object } (* length elements *)
Point = RealPoint | FloatPoint | IntegerPoint
PointList = Integer ( { RealPoint } | { FloatPoint } | { IntegerPoint } )
* length elements *)
RectangleList = Integer { Rectangle } (* length elements *)

Object = Circle | ConvexPolygon | ColorQuickDrawPicture | DDimCircle | DDimPoint | DDimPointVector | DirectedGraph | FloatPoint | GraphEdge | Homogen Vector | Int | IntegerPoint | Layer | LayerVector | LineSegment | LineSegmentVector | List | MarkedRealPoint | Polygon | PolyLayer | PolyLayerVector | PolyLine | PointVector | QuickDrawPicture | RealPoint | Rectangle | RectangleVector | SimpleUndirectedGraph | SpanningTree | Spline2 | SplineGon | Str | StraightEdge | UndirectedGraph | Vector | VoronoiEdge | VoronoiDiagram
References


Curriculum Vitae

I was born on August 19, 1961 in Kaufbeuren, Federal Republic of Germany where I also received my high school diploma as head of my class in 1980. In the last four years of high school I won a total of seven prizes in the national mathematics competition.

From 1980 to 1986 I studied Computer Science with a minor in Mathematics at the TU München with a two semester leave of absence where I studied Computer Science at the ETH Zürich. From 1980 to 1986 I received a fellowship from the State of Bavaria and from 1982 to 1986 a fellowship from the 'Studienstiftung des Deutschen Volkes'. In 1986 I finished as best of class with a diploma (‘Diplom Informatiker’) in Computer Science.

On various occasions I worked as a teaching and research assistant and, for two months, I was a consultant for Siemens in Munich. I also developed a Prolog system for microcomputers which I marketed successfully.

In 1986 I joined Prof. Nievergelt’s group at the University of North Carolina and enrolled in the graduate program which I finished with a Ms. Sc. in Computer Science in 1988. Since 1989 I have been working as a research assistant at ETH Zürich.