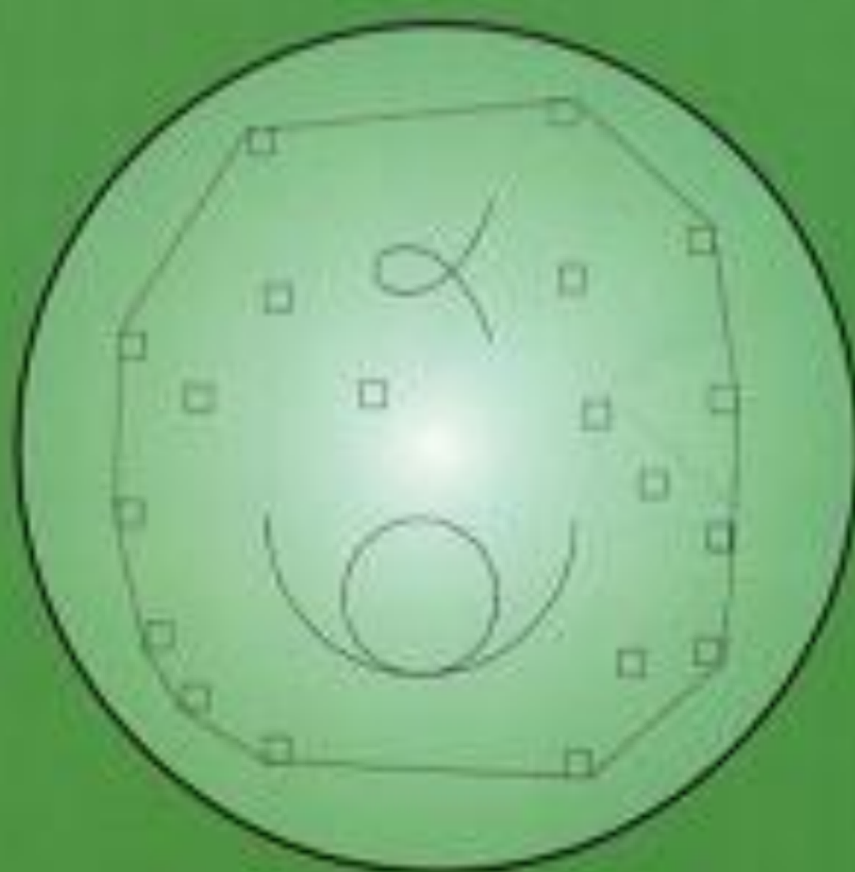
GEOMETRIC COMPUTATIONS WITH INTERVAL AND NEW ROBUST METHODS

Applications in Computer Graphics, GIS and Computational Geometry



H. RATSCHEK and JON ROKNE

Horwood Publishing Series in Computer Science

GEOMETRIC COMPUTATIONS WITH INTERVAL AND NEW ROBUST METHODS

APPLICATIONS IN COMPUTER GRAPHICS, GIS AND COMPUTATIONAL GEOMETRY

ABOUT OUR AUTHORS

Helmut Ratschek was born in Graz, Austria, in 1940. He studied mathematics, physics and philosophy at the University of Graz and obtained a PhD in mathematics in 1966. Since 1968, he has worked in the Department of Mathematics, University of Dusseldorf, Germany where he has been a professor since 1973. He has been a visiting professor at universities in Calgary, Canada, Arlington, Texas, USA and Rome, Italy. He is also an adjunct professor at the University of Calgary. His research is in interval analysis, global optimization and computational geometric algorithms.

Jon Rokne is a professor and the former Head of the Computer Science Department at the University of Calgary, Canada, where he has been a faculty member since 1970. His research has spanned the areas of interval analysis, global optimization and computer graphics and he has co-authored two books, several translations and a number of research papers. He has presented courses and papers at many conferences including Eurographics, Pacific Graphics and Computer Graphics International. He has been a visiting professor at the University of Canterbury, New Zealand (1978), University of Freiburg, Germany (1980), University of Grenoble, France (1983, 1984) and University of Karlsruhe, Germany (1984). He holds a PhD in mathematics from the University of Calgary (1969).

Geometric Computations with Interval and New Robust Methods

Applications in Computer Graphics, GIS and Computational Geometry

Helmut Ratschek,
Universität Düsseldorf, Germany
and
Jon Rokne,
University of Calgary, Alberta,
Canada

Horwood Publishing
Chichester

HORWOOD PUBLISHING LIMITED
International Publishers in Science and
Technology, Coll House, Westergate,
Chichester, West Sussex, PO20 3QL, England
First published in 2003

COPYRIGHT NOTICE
All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the permission of Horwood Publishing.

© Helmut Ratschek and Jon Rokne 2003

British Library Cataloguing in Publication Data
A catalogue record of this book is available from the British Library

ISBN 1-898563-97-7

Printed in Great Britain by Antony Rowe Ltd

Contents

1 Introduction 1

1.1 Errors in Numerical Computations 2
1.2 Geometric Computations 2
1.3 Problems in Geometric Computations Caused by Floating Point Computation 4
1.4 Approaches to Controlling Errors in Geometric Computations . 8
1.5 The Interval Analysis Approach 9
1.6 Global Interval Aspects 10
1.7 The Exact Sign of Sum Algorithm (ESSA) 10
1.8 Arithmetic Filters 11
1.9 Computer Implementations 12

2 Interval Analysis 13

2.1 Introduction 13
2.2 Motivation for Interval Arithmetic 14
2.3 Interval Arithmetic Operations 16
2.4 Implementing Interval Arithmetic 19
2.5 Further Notations 26
2.6 The Meaning of Inclusions for the Range 28
2.7 Inclusion Functions and Natural Interval Extensions 30
2.8 Combinatorial Aspects of Inclusions 32
2.9 Skelboe's Principle 35
2.10 Inner Approximations to the Range of Linear Functions 39
2.11 Interval Philosophy in Geometric Computations 42
2.12 Centered Forms and Other Inclusions 47
2.13 Subdivision for Range Estimation 54
2.14 Summary 59

3 Interval Newton Methods 61

3.1 Introduction 61
3.2 The Interval Newton Method 65
3.3 The Hansen-Sengupta Version 68
3.4 The Existence Test 75

4 The Exact Sign of Sum Algorithm (ESSA) 79
4.1 Introduction 79
4.2 The Need for Exact Geometric Computations 80
4.3 The Algorithm 83
4.4 Properties of ESSA 87
4.5 Numerical Results 92
4.6 Merging with Interval Methods, Applications 93
4.7 ESSA and Preprocessing Implementation in C 101

5 Intersection Tests 109
5.1 Introduction 109
5.2 Line Segment Intersections 114
5.3 Box-Plane Intersection Testing 123
5.3.1 Which of the 3 Approaches is the Best? 129
5.4 Rectangle-Triangle Intersection Testing 130
5.4.1 Interval Barycentric Coordinates 131
5.4.2 The Geometry of Test 1 133
5.4.3 An Algorithm for Test 1 138
5.4.4 Numerical Examples Using Test 1 140
5.4.5 An Alternative Test 142
5.5 Box-Tetrahedron Intersection Testing 147
5.5.1 Three-Dimensional Interval Barycentric Coordinates . . 150
5.5.2 The Algorithm 152
5.5.3 Examples 155
5.6 Ellipse-Rectangle Intersection Testing 157
5.6.1 Analytical Tools Needed 158
5.6.2 The Algorithm 160
5.6.3 Optional Inclusion Tests 163
5.6.4 Complexity and Rounding Errors 167
5.7 Intersection Between Rectangle and Explicitly Defined Curve . 168
5.8 Box-Sphere Intersection Test 174
5.8.1 Introduction 174
5.8.2 Midpoint and Radius as Sphere Parameters 175
5.8.3 Four Peripheral Points as Sphere Parameters 178
5.8.4 Algorithm 182

5.8.5 Numerical Examples 182

6 The SCCI-Hybrid Method for 2D-Curve Tracing 185

6.1 Introduction 185
6.2 The Parts of the SCCI-Hybrid Method 189
6.3 Examples 212

7 Interval Versions of Bernstein Polynomials, Bézier Curves and the de Casteljau Algorithm 219

7.1 Introduction 219
7.2 Plane Curves and Bernstein Polynomials 220
7.3 Interval Polynomials and Interval Bernstein Polynomials 226
7.4 Real and Interval Bézier Curves 230
7.5 Interval Version of the de Casteljau Algorithm 234

8 Robust Computations of Selected Discrete Problems 241

8.1 Introduction 241
8.2 Convex-Hull Computations in 2D 242
8.2.1 Introduction 242
8.2.2 A Prototype Graham Scan Version 244
8.2.3 The Exact and Optimal Convex Hull Algorithm 245
8.2.4 Numerical Examples 248
8.2.5 A More Practical Version of the Algorithm 259
8.3 Exact Computation of Delaunay and Power Triangulations . . 264
8.3.1 Introduction 264
8.3.2 Definitions and Methods for Computing Voronoi Diagrams 267
8.3.3 Methods for Constructing Delaunay and Power Triangulations 269
8.3.4 Exact Computation of the CCW Orientation Test . . . 272
8.3.5 Exact Computation of the INCIRCLE Test 273
8.3.6 Complexity Analysis for the Primitives 275
8.3.7 The Main Scheme of the Incremental Algorithm 276
8.3.8 Test Results 280
8.4 Exact and Robust Line Simplification 284
8 4.1 Introduction 284
8.4.2 The Ramer-Douglas-Peucker Line Simplification Algorithm for Polygons 287
8.4.3 Exact Computations of the Comparisons 292

List of Figures

1.1 Going-in and going-out transformations 6

2.1 Line-box intersection problem 45
2.2 Sub-box rejection 55

3.1 Solution set and inclusions 63

5.1 Intersection of two axis-parallel rectangles 113
5.2 Intersecting segments 116
5.3 The three cases of linesegment intersection 118
5.4 Close to degenerate configuration of segments 120
5.5 Dependence of wrong and inconclusive results on the perturbation 121
5.6 Dependence of wrong and inconclusive results on the perturbation 122
5.7 Box-plane intersection 126
5.8 The four possible rectangle-triangle configurations 130
5.9 Partitioning by one of the barycentric coordinates 132
5.10 A counterexample to intuition 133
5.11 The bounding rectangle possibilities 134
5.12 The geometry of the proof 138
5.13 Rectangle triangle intersection examples 141
5.14 The requirement of Step 5 143
5.15 The requirement of Step 6 143
5.16 k, k' and k'' are visible from r 144
5.17 The two cases for location of t in Step 7 144
5.18 Slope relationships 145
5.19 k above t and k below t 146
5.20 The bounding box and smallest axis-parallel box 148
5.21 Box-tetrahedron test 156
5.22 Rectangle 159
5.23 The initial easy cases of intersection[1] 163

5.24 The remaining cases when $mE \notin \mathcal{R}$ and $(x_0, y_0) \notin E$ 164
5.25 Cases for the $E \subseteq \mathcal{R}$ test 166
5.26 Intersection testing of curve and rectangle 168
5.27 Box-sphere configuration 176
5.28 Box partly intersecting sphere 183
5.29 Box outside sphere 184
5.30 Box included in sphere 184

6.1 Sample configurations 194
6.2 Impossible configurations 194
6.3 Possible configurations when $0 \in F_x(X,Y)$, $0 \notin F_y(X,Y)$ . . . 196
6.4 Impossible configurations when $0 \in F_x(X,Y)$, $0 \notin F_y(X,Y)$ . . 196
6.5 Possible configurations when $0 \in F_x(X,Y)$, $0 \in F_y(X,Y)$ . . . 200
6.6 Plotting cells $\tilde{Z}$ 201
6.7 Cone containing contour 202
6.8 Possible cones 204
6.9 The general case for a cone 205
6.10 Swapping for left-to-right trend 208
6.11 Two curve pieces 211
6.12 Two close co-centric circles 215
6.13 A cross with a singular point, small cell width 215
6.14 Plotting of a curve with a triple point 216
6.15 Plotting of a figure with an approximate triple point 217
6.16 Plotting of a curve crossing point 218
6.17 Plotting of a circle touching another circle 218

7.1 Bezier curve of degree 3 231
7.2 Changed Bezier curve of degree 3 233
7.3 Convex hull of control points 233
7.4 Convex combinations 235
7.5 Interval convex combinations 236
7.6 Two-dimensional interval 237
7.7 An interval curve 238

8.1 Graphics for Table 8.1 (O'Rourke's example) 250
8.2 Graphics for well-conditioned example with 20 points 251
8.3 Graphics for well-conditioned example with 100 points 252
8.4 Graphics for well-conditioned example with 1000 points 253
8.5 Graphics for random ill-conditioned example with 20 points . . 254
8.6 Graphics for random ill-conditioned example with 100 points . 255

8.7 Graphics for random ill-conditioned example with 1000 points . 256
8.8 Graphics for random ill-conditioned example with 20 not machine representable points 257
8.9 Graphics for random ill-conditioned example with 100 not machine representable points 258
8.10 Graphics for random ill-conditioned example with 1000 not machine representable points 259
8.11 Rectangle hull of S 262
8.12 The four subrectangles 262
8.13 Stair in R_1 263
8.14 The four stairs 263
8.15 Power diagram 270
8.16 Flipping the edges 279
8.17 Time ratio vs. perturbation 283
8.18 Percentage of wrong edges vs. perturbation 283
8.19 Time ratio vs. number of points 284
8.20 Percentage of wrong edges vs. number of points 285
8.21 A non-unique R-D-P simplification 288
8.22 An ϵ-strip 289
8.23 Line to be simplified 290
8.24 The three regions 294

List of Tables

2.1 Quotients of widths of inclusions 52

5.1 Test results for parallel segment configuration 121
5.2 Test results for perpendicular segment configuration 122

8.1 Data from computing with O'Rourke's example as input 249
8.2 Data from well-conditioned example with 20 points 251
8.3 Data from well-conditioned example with 100 points 252
8.4 Data from well-conditioned example with 1000 points 253
8.5 Data from random ill-conditioned example with 20 points . . . 254
8.6 Data from random ill-conditioned example with 100 points . . 255
8.7 Data from random ill-conditioned example with 1000 points . . 256
8.8 Data from random ill-conditioned example with 20 not machine representable points 257
8.9 Data from random ill-conditioned example with 100 not machine representable points 258
8.10 Data from random ill-conditioned example with 1000 not machine representable points 259
8.11 Total number of operations 276
8.12 Algorithm performance for perturbation values on grid 281
8.13 Algorithm performance for perturbation values on square . . . 282

Preface

This teaching and research text for undergraduate and postgraduate students and researchers will familiarize readers with interval arithmetic and related tools for gaining reliable and validated results and logically correct decisions for a variety of geometric computations.

The aim of this monograph is to make the reader familiar with interval arithmetic and related tools for gaining reliable and validated results and logically correct decisions for a variety of geometric computations.

In parallel it is demonstrated how interval arithmetic can be used to create simple constructions and structures in the mathematical treatment and computational preparation of the basic issues of geometrical computations.

The results are validated since computations executed with interval arithmetic enable an automatic control of rounding errors and to provide bounds for them. Hence, if the result of an interval computation says two given rectangles intersect then the interval doctrine guarantees that they in fact intersect, and that this result has not been falsified by rounding errors.

A further advantage of interval arithmetic is the ability to represent and to deal not only with points, that is, real numbers, but also with sets of reals such as intervals, rectangles, parallelepipeds, balls, or simplices. This means that many geometric situations, that need some effort to express in the environment of reals, can be simply expressed in an interval arithmetic environment. For example, let A and B be two axes-parallel rectangles. Then, it is quite simple from a pure mathematician's point of view to investigate whether the two rectangles intersect or not. But if a programmer wants to write a code in some programming language like C, Fortran, or Pascal, he will discover that a good deal of thinking is necessary to find the right comparisons of all the components of the 8 vertices in order to get a consistent and complete procedure for this theoretically easy problem. In contrast to this situation, the same problem posed in interval arithmetic is very simple: The two rectangles intersect if and only if the null point 0 lies in the difference of the two rectangles, that is, $0 \in A - B$, where A and B are interpreted as two-dimensional intervals.

Another important helpful set-theoretic feature of interval analysis is the possibility to construct approximations of the range of a function in a computationally very simple manner. One just has to call up the function expression, for example, $f(x)$ and take the domain of the function, say X as variable. This results in an approximating interval $f(X)$ which contains the range of f over X. Since many of the function expressions which occur in geometrical computations are not too involved, the approximation frequently gives the exact range. The ability to determine approximations of the range is a valuable means for the development of search procedures and, connected with these, subdivision strategies for an arbitrarily precise localization of points or areas with a required property. Such techniques can be used at finding zeros of systems, extremum points of functions, intersection points of geometric figures,

etc.

It is known that complicated geometrical constructions can, in the main, be built upon simpler constructions. The simplest of these are usually called geometrical primitives. It is, therefore, in general, sufficient to study the application of interval arithmetic to such primitives. A famous example is the so-called left-turn test which decides whether a given point in the plane lies on the left, on the right or on a given directed line. Independent of how degenerate the situation is or how small the distances of the point to the line is or the distance of the points that define the line is (even if it is machine accuracy), a wrong answer is never given. However, due to the influence of rounding errors there are extreme situations where the interval arithmetic computation renders an inconclusive result (but never a wrong result, and most important, it can be clearly indicated by the program if an inconclusive result occurs). In order that the tests work also for these rare cases, that is, to make the programs complete and consistent, an exceptional algorithm called ESSA has been developed which always (!) determines the sign of a sum of machine numbers errorfree without using variable mantissa length or any hardware tools.

Summing up, our intention in writing this monograph was primarily to demonstrate a few numerical key points with a variety of possible applications in the fields of computational geometry, computer graphics and solid modelling. Hence we aim to stimulate activities in interval applications rather than to provide an exhaustive survey of all possible constructions.

As usual, the book begins with an introduction which covers the utility and helpfulness of interval analysis in dealing with stability and reliability problems in computational geometry. Then an extensive discussion of interval arithmetic and interval analysis as well as its implementation on a machine is initiated. No previous related experience is necessary. Emphasized are the two main directions, the error analysis and the set-theoretic use of interval arithmetic. The famous interval Newton method is also included which enables the proof of existence of zeros of systems of equations and localize them together with validated error bounds.

As already mentioned there is a small percentage of degenerate situations where interval tools are too crude for giving a decisive Boolean or arithmetic result. For these cases ESSA (Exact Sign of a Sum Algorithm) saves the completeness and correctness of many tests. It is developed in Ch. 4. ESSA is especially useful for degenerate constellations at left-turn tests, intersection tests with various geometric objects, geographic information systems, convex hull computations, etc.

Some basic issues relating to geometric computations are considered. Since the applications of ESSA combined with interval arithmetic are relatively new in this field, several proofs of facts are also added since these tools make it necessary to rethink and to reformulate various basically well-known geometrical facts, problems, constructive proofs and solutions. These basic themes consist

of a number of intersection tests (such as line segment vs. line segment, box vs. plane, rectangle vs. triangle, box vs. tetrahedron, ellipse vs. rectangle, rectangle vs. explicitly given curve, box vs. spheres) and a hybrid method for tracing implicitly defined 2D curves where this method particularly addresses curves around singularities, for instance, forks or ill-conditioned curve parts.

Then we added an issue of rather theoretical nature which has several interesting practical consequences. It is the interval representation of Bernstein polynomials, Bézier curves and the de Casteljau algorithm. One should not misunderstand this formulation and believe that the just mentioned representation means nothing more than the replacement of reals or real variables by intervals or interval variables. The crux of this section is to show that only certain reformulations of the specific definitions and theorems make it possible to make the step from reals to intervals successful.

Another main part of interval and ESSA applications is the investigation of complex algorithms for modelling and plotting convex hulls of arbitrary 2D finite point sets (we also mean arbitrarily ill-conditioned sets) and to establish simple methods to find the hulls errorfree if the points of the sets are machine numbers and to find the smallest machine-representable convex hull if the given set has points which are not machine representable.

The errorfree numerical execution of Delaunay and power triangulations, which is the next issue, will then already be a routine exercise involving already familiar primitives.

Finally, the monograph concludes with an exact and reproducible line simplification algorithm, which is frequently used in geographic information systems. Such an algorithm enables to simplify a polygon with a dense number of vertices (think of Norway's shore, for instance) in order to get a more visible polygon which is approximating the former one. The replacement shall take into account that the "geographic character" of the polygon is maintained, for example if the geographic map has to be shrunk.

Readership: Students and researchers in the fields of engineering, geography computer graphics, solid modelling, computer aided design and others interested in validated geometric computations.

Level of the readers' knowledge for an easy understanding: one year calculus of last year highschool or first year university level.

Personal comment on this book from R. E. Moore, the progenitor of interval arithmetic:

> It is always a pleasure to see a new book by Helmut Ratschek and Jon Rokne. Their two, widely cited, previous books Computer Methods for the Range of Functions (Ellis Horwood/John Wiley, 1984) and New Computer Methods for Global Optimization (Ellis Horwood/John Wiley, 1988) are recognized as classics, for their

content as well as their clarity of exposition.

Ramon Moore

Acknowledgements. Thanks are due to David Hankinson and Chris Bone, who assisted with the interval arithmetic implementation, to Karin Zacharias who implemented interval geometric primitives, to Georg Mackenbrock who wrote the codes and computed numerical examples for ESSA, the SCCI-hybrid method, and the convex hull algorithm, to Ania Lopez who also computed examples for the SCCI-hybrid method, to Pavol Federl who wrote the applet for the internet implementation of the convex hull algorithm, to Lynn Tetreault who wrote the applet for internet implementation of the line simplification algorithm, to Jennifer Walker who assisted with the Latex for the book and to the National Sciences and Research Council of Canada for financial support.

Chapter 1

Introduction

This monograph introduces the reader to the interval arithmetic and related tools for a variety of fields collectively grouped under the umbrella of geometric computations.

There are two main tools.

- The first tool is the interval arithmetic. On the one hand, it monitors and controls numerical errors so that the results of geometric computations are reliable and logically correct. On the other hand, it employs set theoretic properties of intervals together with subdivision techniques to simply represent the range of values of geometric functions over relatively large domains exactly or, at least to estimate it as sharp as necessary.
- The second tool is the development of ESSA. It is an algorithm which determines the sign of a sum of machine numbers errorfree. It is the background for executing several geometric primitives such as the left-turn-test errorfree.

As far as we know, the usefulness of interval arithmetic for computer graphics was first discovered by Mudur-Koparkar [174]. Following their work a number of researchers realized that it was a tool that could easily be applied to many problems encountered in computer graphics, CAD and other areas where robust geometric computations are required. For example, a third of the excellent book by Snyder [254] is devoted to interval techniques. ESSA was first presented in Ratschek-Rokne [218] and it had impressive applications to geometrical computations as shown in this book.

This introductory chapter first expands on the necessity for error control in floating point computations, especially in geometric computations. It then gives a survey of the two main tools, that is, interval arithmetic techniques and ESSA, how they fit together and supplement each other, and how they can be brought together and employed for the geometric computations described in

this book.

We remind the reader that our aim with this book is not to treat a special area of computer graphics or to establish the state of the art for interval applications to geometric computations. The intention is rather to demonstrate the variety of possible applications in the fields of computational geometry, computer graphics and solid modeling. Hence we aim to stimulate activities in interval applications for geometric computations as opposed to providing an exhaustive survey of all possible applications.

1.1 Errors in Numerical Computations

It has been long realized that in order to make reasonable statements about numerical computations it is also necessary to include an analysis of the numerical errors in the computations. These errors can arise from a variety of sources, such as input data, which are not known precisely or representable exactly on the machine in use, or from iterative or approximate methods (Newton algorithm, Simpson rule etc.), or from rounding errors, etc.

The latter type of error is focused on in this monograph. The effect of the individual elementary errors can build up and they can eventually overwhelm the desired result even in a perfectly well formed computational procedure. It is also possible that computations that are normally correct can produce completely erroneous values when the real result is close to (often unknown) singularities. The latter problem can occur with even very simple computational problems, such as solving for the roots of quadratic equation. Normal input data to a routine for solving the roots of a quadratic equations will generally result in calculations which are correct to as many figures as is possible to represent in the computational device. Particular input data can, however, generate results with arbitrarily few correct figures in the result [57, 58].

In this monograph we will apply interval arithmetic to bound and control the results of numerical computations, in particular the errors in numerical computations that occur in fields employing geometric information. Such an error control is an unavoidable preparation in order to be able to furnish reliable and logically guaranteed statements.

1.2 Geometric Computations

Over the last 25 years a number of areas of research and applications have developed where numerical computations which deal with geometric data are included and where results are interpreted as geometric objects or statements about geometric objects. As is normally the case, the algorithms for these numerical computations have been established with the assumption that both the input data and intermediate results are real quantities and that the operations

are those defined for real numbers. This assumption is reasonable in order to develop algorithms for geometric computations, but it is far from the truth when the algorithms are implemented. As stated by Forrest [56]:

> Geometric algorithms are notorious in practice for numerical instability. Many working modellers are far from robust for numerical reasons. It seems that implementors seldom remember any of the numerical analysis they have learned when it comes to writing geometric code, but the root causes of many of the problems lie in elementary numerical ill-conditioning.

We agree with the notion that some of the problems can be traced to elementary ill-conditioning, but there are also other fundamental problems such as cracking that are more difficult to deal with. (Cracking is a phenomenon which occurs when 2D surfaces are approximated by a collection of surfaces with geometric or mathematical simpler shape such as planar patches. If these surfaces approximate the surface optimally in a certain sense then the underlying mesh might not be uniform which means that the patches need not fit together at the edges. Under these conditions cracks might occur in the approximating surface.) Both types of problems will be discussed in this monograph and some solutions will be presented.

Computer graphics is probably the most common area using geometric computations. Here geometrical constructs are displayed on computer terminals, screens of workstations and other output devices. Two stages can in many cases be identified in this process. The first stage consists of the computations for the actual geometric constructs and the second stage consists of quantizing the result to the (often extremely coarse) output resolution. The second stage is dealt with extensively in textbooks in computer graphics, for example Foley [54]. Interval tools are rarely applied at this stage and we therefore only consider the first stage in this monograph where geometric constructs that include artificially generated three-dimensional scenes are manipulated, projected, rotated, scaled, and deformed.

An area closely related to computer graphics is *geometric modelling* where models of solids are displayed on computer terminals and other output devices and where they are rotated and scaled as required. Both geometric modelling and computer graphics are part of CAD/CAM (computer aided design / computer aided machining), cf. Zeid [280]). Another area is *computational geometry* which considers geometric problems in n dimensions (typically n is equal to 2 and 3) and where the aim is to develop efficient algorithms for the problems. Examples include finding the convex hull of a set of points, finding the intersection between two convex hulls, computing minimum distances between sets of points, and so on.

Other areas using geometric computations are for example geography (mainly in geographical information systems, abreviated as GIS), astronomy, civil and surveying engineering.

The commonality between these areas is that they deal with objects and point sets in 2, 3 and more dimensions. Furthermore, computations are executed that determine relationships between the points and the objects. In a very rough sense we can describe the commonality by saying that the input is generally continuous and the output is combinatorial.

Almost all computations on geometric objects is done using digital computing devices. These devices can not store and manipulate the set of real numbers, but rather a subset known as the floating point numbers. This means that a problem of discretization occurs, albeit at a very fine level. This problem is that if two real numbers are given such that there is no floating point number between them, then these two numbers must be represented by either the same floating point number or two adjacent floating point numbers. Since we know there is an infinite number of real numbers between any two real numbers, this means that an infinity of real numbers has to be represented by two floating point numbers. This causes problems even though a typical computing device can represent a relatively large number of floating point numbers.

This problem occurs at each stage of a computation. The data for the elementary operations in a computation are all floating point numbers, but the results are not which means that they have to be approximated by floating point numbers. The final result of a computation is therefore an approximation to the result that would have been computed if the computations were exact, i.e. performed with real numbers.

In [99] floating point computations are described with the following analogy:

> Doing floating-point computations is like moving piles of sand around. Every time you move a pile you loose a little sand and you pick up a little dirt.

One of the aims of this book is to discuss some tools for minimizing both the loss of sand and the accumulation of dirt.

1.3 Problems in Geometric Computations Caused by Floating Point Computation

A host of problems arise. when using floating point numbers in geometric computations. In the excellent article [107], the problems of accuracy in geometric computations with floating point numbers are discussed and related to the question of *robustness* of these computations. It is not quite simple to say what robustness is since several approaches to this concept can be found in the literature. (The same holds for other notations such as stable, well-posed, well-conditioned, etc. which are well-defined in numerical analysis, but which are used in different - and not always precise - manners in the areas covered by geometric computations.) Fortunately we do not need a precise definition,

since we do not perform a quantitative rounding error analysis. We only wish to connect some kind of well-behavior of an algorithm with the attribute robust and to hint at a short selection of explanations: For example by robust algorithms Li-Milenkovic [147] understand algorithms "whose correctness is not spoiled by round-off errors". Fortune [59] goes one step further: "An algorithm is robust if it always produces an output that is correct for some perturbation of its input, it is stable if the perturbation is small".

To quote [118]:

> Everyone who works in CAD/CAM, to say nothing of solid modelling knows that there is a problem with robustness. It is usually perceived to be a question of choosing tolerances carefully enough to make different parts of a system work reliably together. However, the problem is not just one of traditional numerical analysis, because of the strong coupling of geometry and topology. Topological data - what is connected to what - are discrete "yes" or "no" decisions. Geometrical data defining the physical locations of points, curves, and surfaces are almost universally treated as floating point numbers. Robustness requires that topological inferences drawn from possibly ill-conditioned numerical operations on the geometrical data must be internally consistent, in the sense that they describe an object that can exist as a solid in R^3.

For example, if three points are given in the plane, then the question can arise whether these points lie on a straight line or not. This seemingly simple problem can be the cause of some confusion, if the further computations depend on the truth or falsity of the answer to the question. Fixing two points and selecting a coordinate system allows us to compute the equation of a line. Since the computation is carried out within the set of floating point numbers the result is represented by floating point numbers. If the computed coordinates of the third point are far enough away from the line, and if the computations are carried out properly then it is reasonable to conclude that the three points are not on a straight line. If the coordinates of the third point are computed to be on the line or very close to the line, then the results are not conclusive due to the inaccuracies introduced by the floating point approximations.

In this particular example the accuracy of the computation is related to the question of whether the computation that the three points are on a line is correct for a given set of input data. Clearly, this computation is not robust in floating point arithmetic since the accuracy of the computation is poor for points that lie on a line or which are close to lying on a line. Such points may indeed be computed as lying on the line or to the left of the line when in fact they are to the right (see also [204]). One learns from this example that errors in geometric computations can result in inconsistencies in the model analysis. Similarly, small perturbations of the input data may also lead to an inconsistency in the topological analysis which can occur if it is investigated

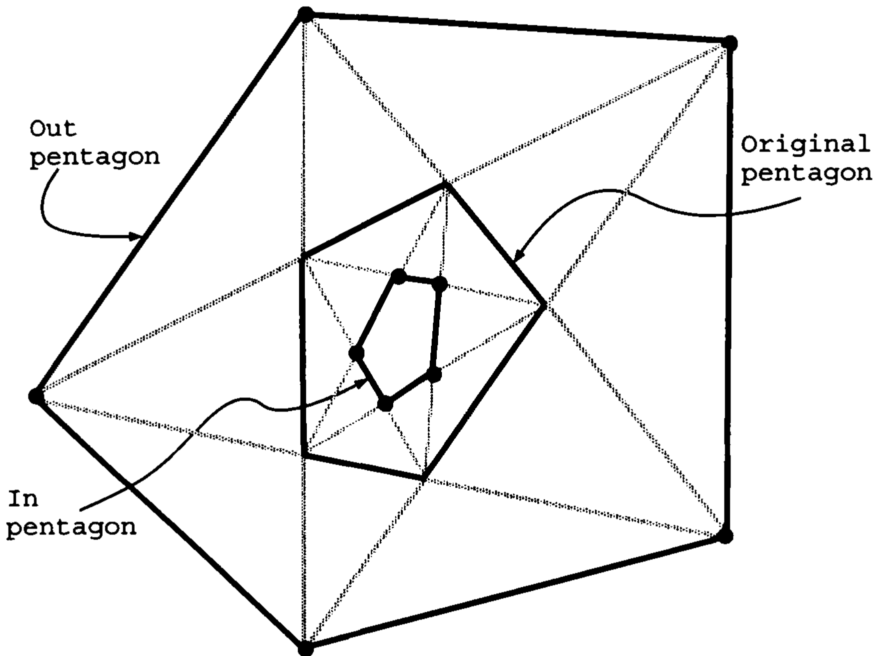


Figure 1.1: Going-in and going-out transformations

whether a line touches the edge of an object or not. For further such examples see Hoffmann [107].

A slightly more complicated example was given by Dobkin and Silver [32]. They suggested that one should choose a pentagon and then perform so-called going-in and going-out transformations. By the going-in transformation they meant the pentagon formed by the intersection of the chords between non-adjacent vertices of the pentagon. By the going-out transformation they meant the pentagon formed from the vertices that are the intersections of the lines containing the edges of the pentagon, see Figure 1.1. Note that the original pentagon is the going-in [respectively -out] transformation of the going-out [respectively -in] transformation of that pentagon.

In this example fairly large positional errors can be demonstrated even after a few going-in and going-out iterations if the computations are executed on a computer. This example is, of course, a little more complex than the previous examples since it involves a number of intersections of two lines. Such intersections require the solution of 2×2 systems of equations (see also [68]). The example of the bad behavior of the solution of a 2×2 system of equations was also considered in the article by Forsythe [58].

Robustness in geometric computations is also considered on the web page [277] with the title *Numerical non-robustness and geometric computations*. Def-

initions for some of the terms discussed above are provided in the following form:

- **Numerical non-robustness** is the informal property of computer programs to behave unpredictably, and often catastrophically, depending on the numerical values of its input parameters.

- In most numerical computation, numerical quantities are approximated and as such, **quantitative errors** are expected and usually benign. However, such quantitative errors can lead to drastic errors that are known as "catastrophic errors" or what we prefer to call **qualitative errors**. Such errors occur when a program enters some unanticipated state with no easy or graceful means of recovery. Colloquially, we say the program "crashed".

- Non-robustness is especially notorious in so-called "geometric" algorithms. But what makes a computation "geometric"? It is not the simple presence of numerical data alone. It turns out, an adequate explanation of "geometric computation" will also lead to an appropriate solution approach.

- We identify geometric computation as involving numerical data L that are intimately tied to combinatorial structures G under some **consistent constraints**. Informally then, a **geometric structure** is a pair (G, L) with such constraints.

- EXAMPLE: Suppose (G, L) represents the convex hull of a set of planar points. Here G may be viewed as a graph that represents a cycle, $G = (v_1, v_2, ..., v_n, v_1)$. L may be regarded as an assignment of the vertices v_i to points p_i in the plane. The consistency requirement here is that these points p_i must be in convex position, and be adjacent in the order indicated by the graph.

- Suppose L' is a perturbation of the true numerical data L. We may say that the perturbation is **only quantitative** as long as (G, L') remains consistent. Otherwise, the perturbation has introduced **qualitative errors** and (G, L') has become inconsistent. Non-robustness of geometric algorithms arises from inconsistencies because all algorithms implicitly rely on such consistency properties.

An example where the effect of errors can be disastrous is given by the rendering of animated sequences such as sunsets behind mountains at various times of the year. It might happen that the rendering algorithm works well for the initial frames whereas the rendering of later frames might encounter unexpected behavior. This might result in a complete re-rendering of the animation.

Even when rendering still-frames the problem can occur. Changes in the scene can give rise to undesired side-effects that necessitate code tuning and rewriting to restore the scene to an acceptable quality.

1.4 Approaches to Controlling Errors in Geometric Computations

Several methods have been suggested for dealing with the problems of errors in geometric computations. Some of these methods are discussed in the survey article by Hoffmann [107].

The first approach suggested by Hoffmann is to use ***symbolic representation.*** Here the computations are carried out as symbolic computations as far as possible. A number of rules are given for them; these are as follows (he uses the three-dimensional setting):

D1: All lines and points must be declared in advance as triples of variables in order that no two lines and no points so declared are equal.

D2: If a point P is incident to a line L, then this fact is explicitly stated as $L(P)$. If two lines L_1 and L_2 intersect in the declared point P, then this fact is expressed explicitly by the two incident statements, $L_1(P)$ and $L_2(P)$.

D3: No other incidences exist among the declared points and lines except those explicitly stated.

It is unfortunate that this approach leads to exponential complexity of computation as the number of input elements increase. Even for simple problems a large amount of computations have to be executed.

Another approach is a ***perturbation approach***, analogous to backward error analysis in numerical analysis (see [271]) where the approach to geometric computation consists of computing the exact result to a slightly perturbed input. The argument is that the input data is not necessarily exact to begin with, in which case altering it slightly might be appropriate. The monograph by Knuth [137] treats a number of computational geometry problems in this manner. This perturbation approach is also known as ***epsilon geometry*** (see for example the thesis by Salesin [237]). The disadvantage with this approach is that the results may still not be correct under all possible conditions. For example, if one has to deal with 4 coplanar points (such as 4 box corners of one side of a box), then the coplanarity is lost if one of the points is perturbed out of the plane defined by the other three points.

A further approach is to scale the input data so that they can be expressed as integers and then use integer arithmetic. Some of the problems inherent in this approach, mainly the rapid growth of the length of the integers, are discussed in [60].

1.5 The Interval Analysis Approach

The interval analysis approach to error control and management presented in this book has been used extensively in other numerically oriented fields (especially numerical analysis and global optimization).

The motivation for the use of this approach to numerical computations came from the desire to control the errors induced by the elementary floating-point operations, the basic arithmetic processing steps of a modern computer.

Some of the fundamental properties of interval computations were already known (Young [278], Warmus [270] and Sunaga [262]) when R. E. Moore developed the modern theory of interval analysis via his thesis [164] and his first monograph *Interval Analysis* [165]. Since then there have been a number of monographs on the subject, for example Alefeld-Herzberger [6], Bauch et al. [16], Hansen [91], Kalmykov et al. [121], Moore [166, 169], Neumaier [179], Ratschek-Rokne [212, 213]. Each of these books presents a different point of view of interval analysis. Many international conferences have also been held devoted to this subject. The first international journal devoted to this subject was "Interval Computations", founded in 1992 by V.M. Nesterov. The name of the journal was changed to "Reliable Computing" [226] in 1995 and it is now published by Kluwer Academic Publishers.

In investigating the interval analytic approach to numerical computations one is also led to develop algorithms tailored to interval spaces. These algorithms can be fundamentally different from the real-space algorithms for the same problem. Some of the developments in this book will be directed to taking advantage of these algorithms in the geometric context. We mention in particular the methods for outer approximations of the range. These methods, first developed by Moore [165] are surveyed in the monograph [212] and they can be very effective tools for a variety of geometric problems especially when they are combined with subdivision techniques.

The application of interval arithmetic and interval analysis will not cure all numerical and combinatorial ills that can occur in practical geometric computations. In general, the results will belong to one of the following two types:

> it is certain that the configuration is correct (the point is not on the line segment, the line intersects the region, etc.)

or

> within the precision of the underlying floating point arithmetic and the computational procedure it is not possible to decide whether the configuration is correct or not (the point may or may not be on the line segment, the line may or may not intersect the region, etc.).

In either of the above cases there is more information than what is provided by the standard floating point computations for the same problem since in the

first case the result is certain and in the second case an indication is given that the computation might not be correct.

1.6 Global Interval Aspects

The original intent of interval arithmetic and interval analysis was to monitor and control almost all kinds of errors that arose in numerical computations, especially those that arose in floating point computations as described in the previous sections. It turned out that interval analysis had important applications in the global sense where the tools could be used to derive properties of functions over large domains. These applications are based on outer estimates of the range (mentioned above) combined with subdivision methods for improving the estimates. In the monograph [213] this idea is applied to global optimization and in Snyder [254] we find some applications to solid modelling, mainly in curve and surface design. In this monograph these ideas are carried further, cf. Ch. 5. In many cases it is even possible to not only obtain an outer approximation of the range but the range itself, even if the function is multivariate. An interesting variety of geometric computation problems can be solved by cultivating the underlying techniques, becoming an important tool, especially when solving intersection problems, cf. Ch. 5. A few illustrative examples of this kind are also discussed extensively in Sec. 2.10. These examples are important since they open the door to an understanding of the interval philosophy. This book will open a new horizon for geometric computations if one has understood the philosophy.

1.7 The Exact Sign of Sum Algorithm (ESSA)

Many geometric algorithms are dependent on the sign of a simple expression like a finite sum. Examples of such algorithms are left-turn test, orientation questions, Boolean algorithms (point in circle vs. not in circle), etc. Implementing such algorithms in fixed length floating point arithmetic can lead to inaccurate or wrong geometric configurations due to falsification of the computation by rounding errors. Interval analysis techniques can remove some of the inaccuracies, however, some cases are left that have to be dealt with using exact techniques.

Because of this an algorithm called ESSA [221] which determines the sign of a sum of real quantities errorfree is discussed.

The algorithm is especially designed for computations involving geometry where rounding error free algorithms are particularly desirable due to the strong influence of rounding errors on logical decisions as mentioned above.

In order to meet the condition of being rounding-error-free, the algorithm is so constructed that it processes data that is already in a binary form. Con-

version errors are therefore avoided. The quantities dealt with are

> *normalized binary floating point numbers of a fixed mantissa length* (in short, *machine numbers*).

It should be noted that an extensive literature exists on the computation of the sum of a set of floating-point numbers and on the relationship of this computation to the stability of numerical and geometric computations [45, 104, 200]. Most of this literature does not mention the restricted problem of the determination of the sign of such a sum.

The algorithm will determine the sign of a sum of floating-point numbers exactly provided the input data consists of machine numbers.

The features of the algorithm are:

(i) Only computation with simple mantissa length is required.

(ii) No splitting of mantissas or other mantissa manipulations are required, one only needs to know the exponent part of the floating numbers.

(iii) It is almost never necessary to compute the sum (except when the sum is zero). It is only necessary to compute as many partial sums and arrangements as are required to decide what the sign is. Extensive test series with randomly generated summands indicate that already 5% of the summands are in most cases sufficient to decide the sign of the sum after some preprocessing.

(iv) If the summands are machine numbers, the computed sign is *always* the exact sign, since the computation is rounding-error-free.

(v) No variable length fixed point accumulator is necessary in order to cover the field of possible partial sums.

(vi) Exponential overflow and underflow control is not part of the algorithm.

1.8 Arithmetic Filters

The computational cost of any exact arithmetic computation tends to be high. One way of reducing this cost is provided by ESSA discussed above, where only the essential information is calculated when calculating the sign of a sum. Another way is to use error bounding techniques such as a Wilkinson type analysis (see [271]), interval arithmetic or other tools to estimate the error in a computation. If it can be shown that the estimated error in the computation using floating point arithmetic does not lead to incorrect geometric configurations in a computation then exact arithmetic can be avoided. The tools used to accomplish this are generically called arithmetic filters.

Since our main aim is to discuss the application of interval tools in geometric computations we focus on the use of interval arithmetic arithmetic filters. A more general view is found in the thesis by Pion [196].

In the ensuing chapters the interval arithmetic filters will be used extensively prior to ESSA due to the lower computational cost of the filters.

1.9 Computer Implementations

In the book we have included the basic interval routines and ESSA implemented in the Sun Sparc C++ . The C++ language was chosen due to the current popularity of C and its derivatives among computer graphics professionals and because it allowed us to define geometric constructs as objects in the language. The language also implements parameter overload thus simplifying the definition of the elementary interval arithmetic operations. Many good implementations can be found on the internet. See also a description of the interval library C-XSC in [132]. An interval realization in Pascal is given in [133]. For a Fortran 77 interval package called INTLIB see [131]. It can also be used in Fortran 90. Other Fortran 90 interval packages are [269], [128]. Interval libraries for C++ are PROFIL [135] and a library family in [111]. Another package with variable precision interval arithmetics is described in [43].

Chapter 2

Interval Analysis

2.1 Introduction

In this chapter the basic tools and techniques from interval analysis used in geometric computations are introduced. The global aspect, which was only touched in Sec. 1.6, is treated extensively. First, interval arithmetic is motivated and justified in Sec. 2.2. In Sec. 2.3 the interval arithmetic operations and some basic rules and properties are introduced including infinite interval arithmetic. The computer implementation of interval arithmetic is then discussed in detail, including C++ programs in Sec. 2.4. In Sec. 2.5 the notion of interval arithmetic is extended to matrix and vector interval arithmetic. Sec. 2.6 demonstrates the usefulness of computing inclusions to the range of a function over an interval. This leads into the concept of inclusion functions which are introduced in 2.7. Further details of inclusion functions are established in Sec. 2.7 to 2.12. In particular, centered forms are introduced as unsurpassable means for estimating the range of functions, and two special centered forms are recommended; these are the meanvalue forms and the Taylor forms. These forms can be understood and applied without too much theoretical background. In Sec. 2.11, a few important and typical examples are given to show how the interval concept and geometric computations fit together. They can be seen as a key for understanding the whole book. In Sec. 2.13 subdivision is introduced as a tool for improving the inclusions for the range. Sec. 2.14 summarizes the important recommendations of the chapter.

Although this chapter illuminates many important aspects of interval analysis it does not cover the whole area. Furthermore, proofs of most statements have been omitted. More thorough introductions to the area of interval analysis can therefore be found, for example, in Moore [169], Alefeld-Herzberger [6], Bauch et al. [16], Shokin [248], Nazarenko et al. [177], Kalmykov et al. [121], etc.

2.2 Motivation for Interval Arithmetic

There are two main reasons for using interval arithmetic in numerical computations. These are:

- **A.** all kinds of errors can be controlled, especially rounding errors, truncation errors, etc.
- **B.** infinite data sets can be processed.

These two reasons are now discussed in some detail:

A. Present-day computers mainly employ an arithmetic called fixed length floating point arithmetic or short, *floating point arithmetic* for calculations in engineering and the natural sciences. In this arithmetic real numbers are approximated by a subset of the real numbers called the *machine representable numbers* (abbreviated: *machine numbers* or *floating point numbers* when discussing implementation details). Because of this representation two types of errors are generated. The first type of error, which is frequently called the *conversion error*, occurs when a real valued input data item is approximated by a machine number and vice versa, when a machine number is transformed to a decimal number which is required for the output of a calculation. The second type of error called *rounding error* is caused by intermediate results being approximated by machine numbers. Both types of errors are often combined under the term rounding errors.

Interval arithmetic provides a tool for estimating and controlling these errors *automatically*. Instead of approximating a real value x by a machine number, the usually unknown real value x is approximated by an interval X having machine number upper and lower boundaries. The interval X contains the value x. The width of this interval may be used as measure for the quality of the approximation. The calculations therefore have to be executed using intervals instead of real numbers and hence the real arithmetic have to be replaced by interval arithmetic. When computing with the usual machine numbers $\tilde{x}$ there is no direct estimate of the error $|\tilde{x}-x|$. The computation *with including intervals*, however, provides the following estimate for the absolute error

$$|x - \text{mid } X| \leq w(X)/2$$

where mid X denotes the *midpoint* of the interval X and $w(X)$ denotes the *width* of X. An estimate for the relative error is,

$$\left|\frac{x - \text{mid } X}{x}\right| \leq \frac{w(X)}{2\min|X|} \text{ if } 0 \notin X,$$

where $|X| = \{|x| : x \in X\}$.

Let us consider an example. The real number $1/3$ cannot be represented by a machine number (unless the machine uses base 3, etc.). It may, however, be

enclosed in the machine representable interval $A = [0.33,\ 0.34]$ if we assume that the machine numbers are representable by two digit decimal numbers (without an exponent part). If we now want to multiply 1/3 by a real number b which we know lies in $B = [-0.01,\ 0.02]$ then we seek the smallest interval X which

(a) contains $b/3$,

(b) depends only on the intervals A and B, and does not depend on 1/3 and b,

(c) has machine numbers as boundaries.

These requirements are realized by two steps,

(i) operations for intervals are defined which satisfy (a) and (b),

(ii) the application of certain rounding procedures to these operations yields (c).

By (i), an interval arithmetic is defined, and by (ii) a machine interval arithmetic is defined.

B. An example of the second type where an intermediate result is approximated by a machine interval, mainly due to lack of information, is now considered.

We apply the meanvalue formula to obtain a local approximation of a continuously differentiable function $f : R \to R$ (R denotes the set of reals) near a point $x \in R$,

$$f(x+h) = f(x) + f'(\xi)h. \tag{2.1}$$

For simplicity, it is assumed that $h > 0$. Then $\xi \in [x,\ x+h] =: X$. How can the information given by (2.1) be represented on a computer? How can $f(x+h)$ be evaluated on the computer via the right side of (2.1) if x and h are given? Obviously, ξ is not assigned a numerical value which would be necessary if we wish to compute $f'(\xi)$ automatically on a computer. How can (2.1) be manipulated so that it can be used for further numerical manipulation? For example (2.1) might have to be multiplied by a number. The answer is quite simple: Use interval arithmetic and compute

$$F(x,h) := f(x) + f'(X)h$$

as will be defined in the sequel. Then $F(x,h)$ will be an interval, i.e. representable on the computer, and we will know that $f(x+h) \in F(x,h)$ where $f(x+h)$ is unknown and $F(x,h)$ is known.

Such principles have many interesting applications in numerical analysis and geometric computations. Examples are the computational verification of

the existence or the uniqueness of solutions of equations in compact domains, cf. Moore [167, 168], strategies for finding safe starting regions for iterative methods, cf. Moore-Jones [172], etc. One particular geometric application of such iterative methods is to the computation of the intersection of a ray and a surface, see [100]. The reader is especially encouraged to study the examples in Sec. 2.10. They develop the so-called global aspect of intervals, and they are a key to understanding large parts of this monograph.

A comfortable side effect of the use of interval arithmetic is that when a theoretical interval algorithm is implemented using machine intervals via the so-called outward rounding, the rounding errors are completely under control and cannot falsify the results, cf. Sec. 2.4. This means that geometric algorithms implemented using interval arithmetic will be robust in the sense discussed in the introduction.

2.3 Interval Arithmetic Operations

In this section the basic operations on intervals are defined and some of the properties of interval arithmetic are given. The differences between real arithmetic and interval arithmetic are emphasized and an arithmetic of infinite intervals is also introduced.

Let I be the set of real compact intervals $[a,\ b]$, $a, b \in R$ (these are the ones normally considered). Operations in I satisfying the requirements (a) and (b) of Sec. 2.2 are then defined by the expression

$$A * B = \{a * b : a \in A, b \in B\} \text{ for } A, B \in I \tag{2.2}$$

where the symbol $*$ stands for $+, -, \cdot$, and $/$, and where, for the moment, A/B is only defined if $0 \notin B$.

Definition (2.2) is motivated by the fact that the intervals A and B include some exact values, α respectively β, of the calculation. The values α and β are generally not known. The only information which is usually available consists of the including intervals A and B, i.e., $\alpha \in A, \beta \in B$. From (2.2) it follows that

$$\alpha * \beta \in A * B \tag{2.3}$$

which is called the ***inclusion principle of interval arithmetic*** This means that the (generally unknown) sum, difference, product, and quotient of the two reals is contained in the (known) sum, difference, product, respectively in the quotient of the including intervals. Moreover, $A * B$ is the ***smallest*** known set that contains the real number $\alpha * \beta$. Moore [164] proved that $A * B \in I$ if $0 \notin B$.

It is emphasized that the real and the corresponding interval operations are denoted by the same symbols. So-called ***point intervals***, that is intervals consisting of exactly one point, $[a,\ a]$, are denoted by a. Expressions like aA,

$a + A$, A/a, $(-1)A$, etc. for $a \in R$, $A \in I$ are therefore defined. The expression $(-1)A$ is written as $-A$.

Definition (2.2) is useless in practical calculations since it involves infinite sets. Moore [164] proved that (2.2) is equivalent to the following constructive ***rules***,

$$\begin{aligned} [a,\ b] + [c,\ d] &= [a + c,\ b + d], \\ [a,\ b] - [c,\ d] &= [a - d,\ b - c], \\ [a,\ b] \cdot [c,\ d] &= [\min(ac, ad, bc, bd),\ \ \max(ac, ad, bc, bd)], \\ [a,\ b]\ /\ [c,\ d] &= [a,\ b] \cdot [1/d,\ 1/c] \text{ if } 0 \notin [c,\ d]. \end{aligned} \tag{2.4}$$

Note that (2.4) shows that ***subtraction and division in I are not the inverse operations*** of addition and multiplication respectively as is the case in R. For example,

$$\begin{aligned} [0,\ 1] - [0,\ 1] &= [-1,\ 1], \\ [1,\ 2]/[1,\ 2] &= [1/2,\ 2]. \end{aligned}$$

This property is one of the main differences between interval arithmetic and real arithmetic. Another main difference is given by the fact that the distributive law of real arithmetic does not carry over to interval arithmetic in general. Only the so-called ***subdistributive law***,

$$A(B + C) \subseteq AB + AC \text{ for } A, B, C \in I \tag{2.5}$$

holds in I. For example,

$$\begin{aligned} &[0,\ 1][1 - 1] = 0, \\ &[0,\ 1]1 - [0,\ 1]1 = [-1,\ 1]. \end{aligned}$$

It follows from this that the order of operations in a formula or an expression becomes important. It can be thought of as an extension of the fact that the numerical properties of different expressions for a given function can vary widely, see for example [57, 58].

The above two deviations from real arithmetic are the main reasons why real algorithms can not be translated directly into interval arithmetic. Instead new algorithms and new ideas have to be provided in order to take full advantage of interval arithmetic and interval analysis.

The ***distributive law*** is valid in some special cases, for example,

$$a(B + C) = aB + aC \text{ if } a \in R \text{ and } B, C \in I.$$

The following properties follow directly from (2.2): Let $A, B, C, D, \in I$ and $*$ be any interval operation then

$$\begin{aligned} &A + B = B + A, \\ &A + (B + C) = (A + B) + C, \\ &AB = BA, \\ &A(BC) = (AB)C, \\ &A \subseteq B, C \subseteq D \text{ implies } A * C \subseteq B * D \text{ (if } B * D \text{ is defined).} \end{aligned} \tag{2.6}$$

The last property of (2.6) is the very important *inclusion isotonicity* of interval operations. This property is essential both for the efficient implementation of interval arithmetic on a computer (see Sec. 2.4) and for the used of interval arithmetic together with subdivisions to obtain convergent algorithms.

An extension of the interval arithmetic operations defined above to *unbounded intervals* is used in this chapter and it is therefore defined here. Alefeld [4] was the first to use infinite intervals in Newton methods. The following formulas are due to Hansen [90]:

Let $0 \in [c,\ d]$ and $c < d$, then

$$[a,\ b]/[c,\ d] = \begin{cases} [b/c,\ +\infty) & \text{if } b \le 0 \text{ and } d = 0, \\ (-\infty,\ b/d] \cup [b/c,\ +\infty) & \text{if } b \le 0, c < 0, \text{ and } d > 0, \\ (-\infty,\ b/d] & \text{if } b \le 0 \text{ and } c = 0, \\ (-\infty,\ a/c] & \text{if } a \ge 0 \text{ and } d = 0, \\ (-\infty,\ a/c] \cup [a/d,\ +\infty) & \text{if } a > 0, c < 0, \text{ and } d > 0, \\ [a/d,\ +\infty) & \text{if } a \ge 0 \text{ and } c = 0, \\ (-\infty,\ +\infty) & \text{if } a < 0 \text{ and } b > 0, \end{cases} \tag{2.7}$$

and furthermore $[a,\ b]/0 = (-\infty,\ \infty)$.

These formulas are not applicable to every problem, but they are appropriate for solving linear equations in connection with the interval Newton method. There is also no need for implementing the formulas (2.7) explicitly on the machine since they are finally intersected with a bounded interval such that the result is always either a bounded interval, a pair of bounded intervals, or the empty set.

It is hardly possible to deal with larger interval arithmetic calculations unless formulas and rules are available for common properties of intervals. For a good collection of such formulas and their proofs the reader is referred to Alefeld-Herzberger [6] and Neumaier [179]. Examples of such formulas are, where $w([c,\ d]) = d - c$ and mid $([c,\ d]) = (c+d)/2$:

$$\begin{aligned} w(aA \pm bB) &= |a|\,w(A) + |b|\,w(B), \\ (aA \pm bB) &= \text{mid}\,(A)a \pm \text{mid}\,(B)b \end{aligned} \tag{2.8}$$

for $a, b \in R$, $A, B \in I$. If A is *symmetric*, that is, $A = [-a,\ a]$ for some $a \ge 0$, and if $B = [c,\ d]$ then

$$\left.\begin{aligned} AB &= A\max(|c|, |d|), \\ A/B &= \begin{cases} A/c \text{ if } c > 0, \\ A/d \text{ if } d < 0, \end{cases} \\ w(AB) &= 2a\max(|c|, |d|), \end{aligned}\right\} \tag{2.9}$$

etc.

We also need symbols $a \vee b$, $a \vee b \vee c$, $a \vee A$, or $A \vee B$ for $a, b, c \in R$ and $A, B \in I$ to denote the smallest interval that contains a, b resp. a, b, c resp. a, A, resp. A, B.

2.4 Implementing Interval Arithmetic

Let us return to the requirements (c) or (ii) of Sec. 2.3, that is, that the endpoints of our intervals must be machine numbers (i.e. floating point numbers). This leads to a special topic called *machine interval arithmetic*. It can be considered as an approximation to interval arithmetic on computer systems leading to the practical use of interval arithmetic.

Machine interval arithmetic is based on the inclusion isotonicity of the interval operations in the following manner: Let us again assume that α, β are the unknown exact values at any stage of the calculation, and that only including intervals are known, $\alpha \in A$, $\beta \in B$. Then A, B might not be representable on the machine. Therefore A and B are replaced by the *smallest machine intervals* that *contain* A and B,

$$A \subseteq A_M, \quad B \subseteq B_M$$

where we denote the replacement of A by A_M and where a machine interval is an interval which has left and right endpoints that are machine numbers. From (2.6) it follows that

$$A * B \subseteq A_M * B_M.$$

The interval $A_M * B_M$ need not be a machine interval and it is therefore approximated by $(A_M * B_M)_M$ which is representable on the machine. This leads to the *inclusion principle of machine interval arithmetic*:

$$\alpha \in A, \beta \in B \text{ implies } \alpha * \beta \in (A_M * B_M)_M. \tag{2.10}$$

Thus, the basic principle of interval arithmetic is retained in machine interval arithmetic, that is, the exact unknown result is contained in the corresponding known interval, and *rounding errors are under control.*

We *sum up*: When a concrete problem has to be solved then our procedure is as follows:

the theory is done in interval arithmetic;

the calculation is done in machine interval arithmetic;

the inclusion principle (2.10) provides the transition from interval arithmetic to machine interval arithmetic.

The practical implementation of a machine interval arithmetic is now discussed. We provide sample programs that facilitate the implementation of interval arithmetic on some common computer systems used extensively in geometric computations. These sample programs may also be modified relatively easily for the use on other computer systems with other languages.

Almost all modern computers use floating point numbers of the form

$$mb^e$$

where m, b and e are the mantissa, base and exponent, respectively, and where both m and e are represented by a fixed number of digits in the base b. This representation is a consequence of the fixed size of bits, bytes and words in a computer, with words commonly being the smallest addressable unit.

Although there are already two meanings of the word "rounding" in common usage, cf. Sec. 2.2, we have to accept a third meaning when considering specific machine behavior. We already mentioned that the exact arithmetic operations are performed on floating point numbers the results are in general not floating point numbers due to the fixed mantissa length restriction. The result therefore has to be assigned to a number representable as a floating point number. If the closest floating point number is selected then the process is also called *rounding* (in the proper sense). If the floating point number is selected by dropping excess lower order digits then the process is called *truncation*. Most common floating point implementations allow a choice between several modes of assigning floating point results. In the sequel we assume that the method of assignments follow the IEEE standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std. 754-1985 [112], see also [113]) as implemented for the Sparc series C compilers via the system function

```
int ieee_flags(action,mode,in,out)
char  *action, *mode, *in, **out
```

The function call

```
ieee_flags("set","direction","tozero", &out)
```

will set the rounding mode to "tozero", that is, the truncation mode, which is the preferred choice for interval arithmetic implementations as will be discussed below.

In order to gain a better understanding of the implementation of A_M when A is given we briefly compare rounding to truncating. Let c be a real number. If c is represented as a decimal number and then truncated to a floating point number, a number c_t is obtained. If the nearest floating point number is chosen then the rounded result c_r is obtained. Clearly

$$\begin{aligned} c - c_t &\leq \epsilon \text{ if } c > 0, \\ c_t - c &\leq \epsilon \text{ if } c \leq 0, \\ |c_r - c| &\leq \epsilon/2 \end{aligned}$$

where $\epsilon = 1\,ulp = 1\,ulp(c)$ is one unit in the last place of c_t. It can also easily be shown that $|c_r - c| \leq |c_t - c|$ which means that rounding is preferable to truncation in numerical computations. In implementing interval arithmetic,

however, where all intervals A has to be replaced by machine intervals A_M, rounding in the proper sense is useless, since the condition $A \subseteq A_M$ has to be met. The transition from A to A_M is also called ***outward rounding*** which can be executed by directed rounding or by utilizing the truncation mode. We use the latter way in developing the machine interval arithmetic.

In the following the skeleton of an implementation of a double precision (64 bit) class interval is given in C++. The routines are described and presented in a natural order, with the basic components being introduced first followed by the interval class and interval arithmetic.

It should also be noted that some of the examples in the latter sections were executed using single precision (32 bit) floating point numbers in order that the effect of the floating point errors would be more easily displayed. The use of single precision interval arithmetic will be explicitly mentioned in each case.

The following C++ code adds a one to the last bit of a double floating point number which implements the *next_floating_point_number* operation required to implement the machine interval arithmetic.

```
/*

This routine adds 1 to the last digit of a double data
item which is here assumed to have 8 bytes where

seeeeeee eeeemmmm mmmmmmmm mmmmmmmm (4 leading bytes)
mmmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm (4 trailing bytes)

s=sign of number
e=exponent of number
(note that the exponent 01111111111 is an exponent
of the representation of 0)
m=mantissa bits with assumed leading 1

The algorithm works by adding one to the last two bytes
of the mantissa (due to the integer arithmetic used).
If the result of this addition is zero then it must
overflow to the next two bytes.
If the mantissa bits are exhausted then the exponent
will be automatically incremented and the mantissa will
be set to all zeros.

No check is made to see if the exponent overflows. Negative
numbers will turn into the next larger negative number.

This routine assumes that there are 8 bytes in a double
data item and that there are 4 bytes in an unsigned integer.
```

```
*/

double add_one(double bound)
 /* routine to add 1 to the last digit of a floating point number */
{
unsigned int *msip, *lsip;
/* most and least significant parts of the number (msip and lsip) */

msip = (unsigned int *)&bound;
/* the address of msip */

lsip = msip + 1;
/* the address of lsip */

*lsip += 1;
/* add one to the least significant portion of the number */

if (*lsip == 0) *msip += 1;
/* if lsip overflows then add 1 to msip */

return(bound);
}
```

For simplicity a routine

`setmode_`

is included:

```
void setmode_()
{
char *out;

ieee_flags("set", "direction","tozero",&out);
}
```

Together with the statement

```
extern "C" {
int ieee_flags(char *, char *, char *in, char **);
```

```
}
```

the routine sets the mode to "truncation".

The basic interval operations defined by equation 2.4 are now implemented in the class interval as given below.

```
/*
The basic arithmetic operations are embedded in the class
interval. In each case the pair of double data items
representing the interval [lo,hi] is checked via the routine
check_bounds. This routine accesses add_one whenever
  lo<0
or
  hi>0.
In both cases the next double representable item is
calculated in add_one and the result is returned to the
calling operator. In this manner it is guaranteed that the
machine interval result will contain the interval result
that would have been computed using infinite (real)
interval arithmetic, i.e. inclusion
isotonicity\index{inclusion isotonicity}
is maintained.
*/

/*defining the class interval*/

class interval {
double lo,hi;


public:

/*accessing the bound of an interval*/

interval(double lo = 0, double hi = 0) {
this->lo = lo;
this->hi = hi;
};

/*an interval printing utility */

void print() {
```

```
printf("[ %31x, %31x ]\n", lo, hi);
};

/*checking lower and upper bounds adding 1 if necessary*/

friend interval check_bounds(interval a) {
    if(a.lo < 0 ) {
    a.lo = add_one(a.lo);
  }
  if(a.hi > 0 ){
    a.hi = add_one(a.hi);
  }
  return(a);
}

/*interval addition*/

friend interval operator+(interval a, interval b){
return(check_bounds(interval(a.lo+b.lo,a.hi+b.hi)));
};

 /*interval subtraction*/

friend interval operator-(interval a, interval b){
return(check_bounds(interval(a.lo-b.hi,a.hi-b.lo)));
};

/*interval multiplication*/

friend interval operator*(interval a, interval b){
double ac,ad,bc,bd;
ac=a.lo * b.lo;
ad=a.lo * b.hi;
bc=a.hi * b.lo;
bd=a.hi * b.hi;
return(check_bounds(interval(MIN(ac,ad,bc,bd),
      MAX(ac,ad,bc,bd))));
};

/*interval division*/

friend interval operator/(interval a, interval b){
double ac,ad,bc,bd;
if( b.lo == 0.0 || b.hi == 0.0 ){
```

```
cout << form("\n bad interval for division");
 }
 ac=a.lo / b.lo;
 ad=a.lo / b.hi;
 bc=a.hi / b.lo;
 bd=a.hi / b.hi;
 return(check_bounds(interval(MIN(ac,ad,bc,bd),
       MAX(ac,ad,bc,bd))));
 };

 /*extract lower and upper bound utilities*/

 friend double lower_bound(interval a) {
   return(a.lo);
 };
 friend double upper_bound(interval a){
   return(a.hi);
 };
 };
```

A few system calls and some definitions are needed to complete the interval package:

```
#define min(a, b) (((a) < (b)) ? (a) : (b))
#define MIN(a, b, c, d) min(min(min(a, b), c), d)
#define max(a, b) (((a) > (b)) ? (a) : (b))
#define MAX(a, b, c, d) max(max(max(a, b), c), d)

#include <stdio.h>
#include <stream.h>
#include <sys/ieeefp.h>
```

The routines for interval multiplication and interval division can be made more efficient by considering 9 cases for each depending on the signs of the result. For both interval multiplication and interval division 8 of the cases only require 2 multiplications resp. divisions. Only one case requires the full 4 multiplications resp. divisions. These cases are described in for example [169]. In the routines presented above this was not done for the sake of clarity.

In [254] and in [241], pp. 160-161, it is suggested that the round-to+∞ mode be used for the upper bound and round-to$-\infty$ mode be used to compute the lower bound if the arithmetic conforms to the IEEE standards for floating

point arithmetic discussed above. This would in general result in a higher operation count unless the above division into cases is implemented. It is also conceptually easier to deal with only one truncation mode for a given numerical (i.e. interval) computation.

Further increases in the execution speed of the interval routines would be achieved if the routines were implemented at the chip or microcode level, see also [261].

There are several commercial and public domain software systems and software packages in which machine interval arithmetic is implemented, for example TRIPLEX-ALGOL-60, ALGOL 68, C-XSC, PASCAL-SC, PASCAL-XSC, FORTRAN-SC, FORTRAN-XSC, MODULA-SC or ACRITH for some IBM computers, ARITHMOS for some Siemens computers, etc. For a basic discussion of such languages see Kulisch-Miranker [145].

The advantage of scientific programming languages, i.e. those that have a letter string "sc" in their name, is that not only the arithmetic operations are executed via maximum accuracy, also called *machine exact*, but also vector and matrix operations. An interesting report about computer systems that provide the "scientific" concept on chips or in microcode is given in Kulisch [144].

2.5 Further Notations

Interval arithmetic can be extended to most spaces considered in numerical computations. For each extension new results have to be derived and new and interesting properties might be found. In this section the idea of interval arithmetic is extended to m-dimensional vectors and matrices in a natural manner. The question of the solution of linear interval equations in m-space is left to the next chapter.

If $A \in I$ then we also write $A = [\text{lb}A, \text{ ub}A]$ denoting the lower and upper boundaries of A by lb$A$ and ub$A$. If $D \subseteq R$ then $I(D) = \{Y : Y \in I, Y \subseteq D\}$. If $B \in I$ then $A \leq B$ and $A < B$ means that ub$A \leq$ lb$B$ resp. ub$A <$lb$B$.

The set of real m-dimensional vectors is denoted by R^m and the set of m-dimensional interval vectors by I^m.

If $A = (A_1, \ldots, A_m) \in I^m$ then A is commonly interpreted as a right parallelepiped $A_1 \times A_2 \times \ldots \times A_m$. The vector of left endpoints of A is denoted by lb$A$ = (lb$A_1, \ldots,$ lbA_m) and the vector of right endpoints is denoted by ub$A$ = (ub$A_1, \ldots,$ ubA_m). Interval vectors are also called intervals when it is clear from the context whether real intervals or interval vectors are intended. A *box* is also a frequently used synonym for an m-dimensional interval vector, a particularly appropriate notation in 3D computer graphics.

If $A = (A_1, \ldots, A_m) \in I^m$ then the *width* of A is defined to be

$$w(A) = \max \{w(A_i) : i = 1, \ldots, m\}$$

and the *midpoint* of A to be

$$\text{mid } A = (\text{mid } A_1, \ldots, \text{mid } A_m).$$

The set of $n \times m$ real matrices is denoted by $R^{n\times m}$ and the set of $n \times m$ interval matrices by $I^{n\times m}$. If $A = (A_{ij}) \in I^{n\times m}$ then

$$\text{mid } A = (\text{mid } A_{ij}) \in R^{n\times m}$$

is the *midpoint* of A.

If $A, B \in I^m$ or $A, B \in I^{n\times m}$ then $A \subseteq B$ means that $A_i \subseteq B_i$ for $i = 1, \ldots, m$ or $A_{ij} \subseteq B_{ij}$ for $i = 1, \ldots, n$; $j = 1, \ldots, m$. Similarly, if $x \in R^m$, $A \in I^m$, or if $x \in R^{n\times m}, A \in I^{n\times m}$ then

$$x \in A$$

means $x_i \in A_i$ for $i = 1, \ldots, m$ or $x_{ij} \in A_{ij}$ for $i = 1, \ldots, n$; $j = 1, \ldots, m$. For instance, mid $A \in A$ holds.

Similarly, if $A, B \in I^m$ then

$$A \leq B \quad \text{or} \quad A < B$$

shall mean $A_i \leq B_i$ for $i = 1, \ldots, m$, or $A_i < B_i$ for $i = 1, \ldots, m$, respectively. Note that $A \leq B$ does *not* mean that $A = B$ or $A < B$ holds as is the case with inequalities in R.

The interval arithmetic operations are extended to *interval vector* and *interval matrix operations* in the usual manner:

$$\begin{aligned} a(A_{ij}) &= (aA_{ij}) \text{ for } a \in R, \ (A_{ij}) \in I^{n\times m}, \\ (A_{ij}) \pm (B_{ij}) &= (A_{ij} \pm B_{ij}) \text{ for } (A_{ij}), (B_{ij}) \in I^{n\times m}, \\ (A_{ij})(B_{ij}) &= (\sum_{l=1}^{k} A_{il}B_{lj}) \text{ for } (A_{ij}) \in I^{n\times k}, \ (B_{ij}) \in I^{k\times m}. \end{aligned}$$

This definition includes the arithmetic for interval vectors (rows as well as columns) by setting $n = 1$ or $m = 1$.

If A, B, C, D are interval vectors or interval matrices and if $*$ denotes one of the operations $+, -$ or $\cdot$ then

$$A \subseteq C, \ \ B \subseteq D \text{ implies } A * B \subseteq C * D. \tag{2.11}$$

Property (2.11) is the extended form of the *inclusion isotonicity* of the interval arithmetic operations.

For $D \subseteq R^m$ we denote by $I(D)$ the set of all boxes $Y \in I^m$ with $Y \subseteq D$. For example, if $X \in I^m$, and thus $X \subseteq R^m$, the set of all subboxes Y of

X is just $I(X)$. In this connection we also say that Y is an *interval variable* over $I(X)$ which shall mean that Y can take each box of $I(X)$ as value. This terminology is mainly used when functions $F : I(X) \to I$ etc. are considered.

An interesting functional is $\chi : I \to [-1,\ 1]$ defined by $\chi[0,\ 0] = -1$ and, if $[a,\ b] \neq 0$, by

$$\chi[a,\ b] = \begin{cases} a/b & \text{if } |a| \leq |b|, \\ b/a & \text{otherwise.} \end{cases}$$

χ characterizes the degree of symmetry of intervals. For instance, $\chi A = -1$ means that A is symmetric, i.e. $A = -A$, and $\chi A = 1$ means that A is a nonzero point interval and hence completely nonsymmetric. Thus, χ admits the geometric interpretation that, for intervals A and B,

$$A \text{ is more symmetric than } B \text{ iff } \chi A \leq \chi B.$$

χ is an indispensable tool for dealing with interval products. For example, if $A, B \in I$ is given then there exists an $X \in I$ with $AX = B$ iff $\chi A \leq \chi B$ (Ratschek [207]). Or, if $\delta A = \text{mid } \{|a| : a \in A\}$ and $A_1, \ldots, A_n \in I$ then (Ratschek-Rokne [217])

$$\left.\begin{aligned} w(A_1 \cdots A_n) &= |A_1| \cdots |A_n| - (\delta A_1) \cdots (\delta A_n) \\ &\quad \text{if } \chi A_i \geq 0\ (i = 1, \ldots, n), \\ w(A_1 \cdots A_n) &= |A_1| \cdots |A_{n-1}| w A_n \\ &\quad \text{if } \chi A_n \leq \chi A_i\ (i = 1, \ldots, n-1) \text{ and } \chi A_n \leq 0. \end{aligned}\right\} \tag{2.12}$$

(Note that all possible cases for $A_1, \ldots, A_n$ are exhausted by the two formulas.)

The χ-functional also allows a splitting of intervals which can be convenient for product considerations:

Let $\sigma A = 1$ if mid $(A) \geq 0$, otherwise set $\sigma A = -1$. Then each interval A can be represented by

$$A = (\sigma A)|A|[\chi A,\ 1] \tag{2.13}$$

where the "signum" of A, the modulus of A, and the symmetry character of A are involved. More important is that the product of two intervals take on an easy form with (2.13), i.e.

$$\left.\begin{aligned} AB &= (\sigma A)(\sigma B)|A||B|[(\chi A)(\chi B),\ 1] \text{ if } \notin A, 0 \notin B, \\ &= (\sigma A)(\sigma B)|A||B|[\min(\chi A, \chi B),\ 1] \text{ otherwise}. \end{aligned}\right\} \tag{2.14}$$

This formula is needed in Sec. 2.8.

2.6 The Meaning of Inclusions for the Range

Computing the range of a function is important for many numerical problems. Exact computations of the range are not possible in general, however, important

information can be gained from computations of inclusions. In this and the following section efficient methods for the computation of an inclusion for a function over a closed compact domain is therefore discussed as well as some applications of the methods to geometric computations.

The main application of outer estimates of the range of a function over a particular domain is to test whether a particular constraint is satisfied for the range or not. Suppose a function $f(x)$ and a region (i.e. interval or box) X is given and suppose the constraint is $f(x) \neq 0$ over X. If we compute an inclusion $F(X)$ such that $f(x) \in F(X)$ for any $x \in X$ with the property that $0 \notin F(X)$ then it follows that $f(x) \neq 0$ for any $x \in X$. If, however, $0 \in F(X)$ then the result is inconclusive, i.e. we can not tell whether $f(x) \neq 0$ for any $x \in X$ or not. This discussion illustrates two points for inclusion functions:

- If a property for sets S of the form "S does not contain a certain object" required for the range is satisfied by an inclusion for the range then the inclusion computation provides a positive answer to the test for the property, that is, the range does not contain this object,
- if the property required is not satisfied by the inclusion then the result is uncertain.

It is therefore important to calculate inclusions for the range of a function that have a small overestimation of the range in order to increase the likelihood of obtaining a positive answer to a test for such a property.

There are three main methods for improving an inclusion for the range of a function:

- Devise improved inclusions by algebraic and analytic means. Typical examples are meanvalue forms, centered forms and Taylor forms, use of monotonicity knowledge of the function,
- subdivide the domain then compute inclusions for the range over each subdomain and compute the bounds of the union,
- compute the global maximum and minimum of the function by optimization methods.

In the next section we provide methods for obtaining improved inclusions for the range using algebraic and analytic techniques. In the last section the very important idea of subdivision to improve the inclusions is discussed. The computation of the maximum and mimimum value over the domain of a function will provide the best inclusion. This computation is expensive and it is part of the topic of ***global optimization*** which was dealt with in [213].

The importance of the knowledge of inclusions of the range in interval analysis was first discussed in Moore [165] in 1966. He also introduced the idea of the centered form which will be developed later. In the ensuing two decades a

number of techniques and algorithms based on the centered form were developed. They were collected and presented in the monograph [212]. More recent work on centered forms is found in for example [2, 235, 14, 149].

In computer graphics the application of inclusions for the range of functions occur in contour and surface tracing, ray tracing and generally intersection of surfaces defined parametrically as well as in proximity tests.

2.7 Inclusion Functions and Natural Interval Extensions

The main tool in the treatment of geometric problems using interval arithmetic are inclusion functions. In this section the concept of an inclusion function is therefore introduced together with the idea of the natural interval extension. Some further properties are also given.

Let $D \subseteq R^m$ and $f : D \to R$. Let furthermore $\Box f(Y) = \{f(x) : x \in Y\}$ be the *range* of f over $Y \in I(D)$. A function $F : I(D) \to I$ is called an *inclusion function* for f if

$$\Box f(Y) \subseteq F(Y) \quad \text{for any } Y \in I(D). \tag{2.15}$$

Inclusion functions for vector-valued or matrix-valued functions are defined analogously. The inclusion condition (2.15) must in this case be satisfied componentwise.

It turns out that interval analysis provides a natural framework for constructing inclusion functions recursively for a large class of functions.

In order to outline this class of functions it is assumed that some fundamental functions are available for which inclusion functions are already known. This assumption is verified by existing computer languages for interval computations. These languages have pre-declared functions g (examples are sin, cos, etc.) available. For these functions it is also assumed that *pre-declared inclusion functions* G satisfying the above conditions (2.15) are given. If the functions G are not given then they are easy to construct since their monotonicity intervals are generally known, such that even $G(Y) = \Box g(Y)$ will hold, in general. It is also easy to realize these inclusion functions G on a computer such that (2.15) is not violated. In this case the influence of rounding errors is kept under control by computing

$$(G(Y_M))_M \text{ instead of } G(Y),$$

cf. Sec. 2.5.

Let $f : D \to R,\ \ D \subseteq R^m$ be a *programmable* function, that is, a function which may be described as an explicit expression without use of logical or conditional statements (such as "if ... then", "while", etc.) in the following manner: Each function value $f(x), x \in D$, can be written down as an expression

(also denoted by $f(x)$) which is independent of the value of x and such that this expression consists only of

(1) the variable or arguments x (or their components $x_1, \ldots, x_m$),

(2) real numbers (coefficients, constants),

(3) the four arithmetic operations in R,

(4) the pre-declared functions g,

(5) auxiliary symbols (parentheses, brackets, commas, etc.).

Let $Y \in I(D)$ then the ***natural interval extension*** of f to Y is defined as that expression which is obtained from the expression $f(x)$ by replacing each occurrence of the variable x by the box Y, the arithmetic operations of R by the corresponding interval arithmetic operations, and each occurrence of a pre-declared function g by the corresponding inclusion function G. This definition is due to Moore [165]. The natural interval extension of $f(x)$ to Y is denoted by $f(Y)$ and is understood as an expression, that is, a string of some specified symbols. The function value which is obtained by evaluating this expression is also denoted by $f(Y)$.

It follows from the definition of an expression, from the inclusion isotonicity of the interval operations, (2.6), and from the properties of the pre-declared inclusions, i.e. the $G's$, to be inclusion functions (see (2.15)) that

$$x \in Y \text{ implies } f(x) \in f(Y). \tag{2.16}$$

Since property (2.16) is the key to almost all interval arithmetic applications and results, it is called the ***fundamental property of interval arithmetic.***

If $f : D \to R, D \subseteq R^m$ is programmable and can be described by a function expression as characterized above then the interval function $F : I(D) \to I$ defined by $F(Y) = f(Y)$ is an inclusion function for f. More importantly we have here an effective constructive means to find an inclusion function F for a real programmable function f using the tool of natural interval extensions.

The reader preferring a more precise presentation is referred to Ratschek–Rokne [212].

Example. If $f(x) = x_1 \sin(x_2) - x_3$ for $x \in R^3$ and if SIN is the pre-declared interval function for sin then $f(Y) = Y_1 \ \mathrm{SIN}(Y_2) - Y_3$ is the natural interval extension of f to $Y \in I^3$.

It is one of the large curiosities of interval arithmetic that different expressions for one and the same function f lead to interval expressions which are also different as functions:

Example. If $f_1(x) = x - x^2$ and $f_2(x) = x(1-x)$ then $f_1(x)$ and $f_2(x)$ are different as expressions, but equal as functions. Further, $f_1(Y)$ and $f_2(Y)$ are also different as functions, i.e., if $Y = [0,\ 1]$ then $f_1(Y) = Y - Y^2 = [-1,\ 1], f_2(Y) = Y(1-Y) = [0,\ 1]$. For comparison, $\Box f(Y) = [0,\ 1/4]$.

It is therefore a very important and challenging problem to find expressions for a given function that lead to the best possible natural interval extensions, that is, $f(Y)$ shall approximate $\Box f(Y)$ as well as possible. Part of the solution to this problem can be found in Ratschek-Rokne [212].

For construction of inclusion functions of programmable functions containing logical connectives see Ratschek-Rokne [213]

A measure of the quality of an inclusion function F for f is the *excess-width*,

$$w(F(Y)) - w(\Box f(Y)) \quad \text{for } Y \in I(D),$$

introduced by Moore [165]. A measure for the asymptotic decrease of the excess-width as $w(Y)$ decreases is the so-called order (also: convergence order) of F, due to Moore [165]: An inclusion function F of $f : D \to R, D \subseteq R^m$ is called of *(convergence) order* $\alpha > 0$ if

$$w(F(Y)) - w(\Box f(Y)) = \mathcal{O}(w(Y)^\alpha)$$

as $w(Y) \to 0$, that is, if there exists a constant $c \geq 0$ such that

$$w(F(Y)) - w(\Box f(Y)) \leq cw(Y)^\alpha \quad \text{for } Y \in I(D).$$

In order to obtain fast computational results it is important to choose inclusion functions having as high an order α as possible. A detailed investigation of the order of inclusion functions is given in Ratschek-Rokne [212]. A similarly looking concept, which is however independent of the order, is the idea of a Lipschitz function. Let $D \subseteq R^m$ be bounded and $F : I(D) \to I^k$. Then F is called *Lipschitz* if there exists a real number K *(Lipschitz constant)* such that

$$w(F(Y)) \leq Kw(Y) \quad \text{for } Y \in I(D).$$

The Lipschitz property delivers us a frequently used criterion for the meanvalue form which is a special inclusion function being of convergence order 2, cf. Sec. 2.9.

2.8 Combinatorial Aspects of Inclusions

It was noted in the previous section that if $f(x)$ was an expression for a programmable function f in the variables $x = (x_1, \ldots, x_n)$ then the natural interval extension was $f(X)$ which resulted from replacing all occurrences of the real variables by interval variables, all occurrences of transcendental functions by

inclusions and all operations by interval operations. A very important property of $f(X)$ was the fundamental property of interval arithmetic

$$\Box f(X) \subseteq f(X). \tag{2.17}$$

A further property was *inclusion isotonicity* expressed as

$$A \subseteq B \text{ implies } f(A) \subseteq f(B).$$

It was also noted earlier that interval arithmetic differs from real arithmetic in two important aspects

1. only the subdistributive law holds, i.e.

$$A(B+C) \subseteq AB + AC \text{ for } A, B, C \in I, \tag{2.18}$$

2. subtraction and division are not the inverse operations of addition and multiplication. For example

$$[0,1] - [0,1] = [-1,1],$$

$$[1,2]/[1,2] = [1/2,2].$$

Because of these differences it follows that the order of operations is important for interval arithmetic. As an example, we have the following 2×2 determinant which will be evaluated in three algebraically equivalent ways:

$$D_1(a_1,a_2,b_1,b_2) = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1 b_2 - a_2 b_1, \tag{2.19}$$

$$D_2(a_1,a_2,b_1,b_2) = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1(b_2 - b_1) + b_1(a_1 - a_2), \tag{2.20}$$

$$D_3(a_1,a_2,b_1,b_2) = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = b_2(a_1 - a_2) + a_2(b_2 - b_1). \tag{2.21}$$

The natural interval extensions of these formulas are different. Let $A_1 = [-1,1]$, $A_2 = [0,2]$, $B_1 = [0,1]$ and $B_2 = [-1,1]$. When A_i, B_i, $i = 1,2$ replaces a_i, b_i, $i = 1,2$ in the above formulas we obtain

$$\begin{aligned} D_1([-1,1],[0,2],[0,1],[-1,1]) &= [-3,1], \\ D_2([-1,1],[0,2],[0,1],[-1,1]) &= [-5,5], \\ D_3([-1,1],[0,2],[0,1],[-1,1]) &= [-7,5]. \end{aligned}$$

Another example, which is a typical feature of the set theoretic definition of interval equations and narrowly connected with the determination of inclusion functions for polynomials, is the interval arithmetic power evaluation. Interval arithmetic distinguishes between the simple and the extended power evaluation.

The *simple* version of the power evaluation (also called power evaluation by *simple arithmetic*) is defined by

$$A^0 = 1 \text{ and } A^n = A \cdot \ldots \cdot A(n \text{ times}) \text{ if } \geq 1$$

for intervals A. The *extended* version of the power evaluation (also called power evaluation by *extended arithmetic*) is defined by

$$\tilde{A}^n = \{a^n : a \in A\} \text{ if } n \geq 0.$$

We also get explicit formulas for the simple as well as for the extended power evaluation.

Let $A = [a,\ b]$ and $n \geq 1$. Then

$$\tilde{A}^n = \begin{cases} a^n \vee b^n \text{ if } 0 \notin A, \\ 0 \vee a^n \vee b^n \text{ if } 0 \in A. \end{cases} \tag{2.22}$$

The formula follows directly from the definition of the extended power evaluation and the monotonicity of the power function on the positive and negative real halfaxes.

In case of the simple power evaluation we get

$$A^n = \begin{cases} a^n \vee b^n \text{ if } 0 \notin A, \\ (\sigma A)^{n-1}|A|^{n-1}A \text{ if } 0 \in A. \end{cases} \tag{2.23}$$

The formula in case $0 \notin A$ follows directly from the multiplication rules for intervals applied recursively. The formula in case $0 \in A$ follows from (2.13).

For instance, one can see from (2.23) that if $A = [a,\ b]$, then

$$\begin{aligned} A^n &= Ab^{n-1} \text{ if } 0 \in A, \sigma A = 1, \\ A^n &= Aa^{n-1} \text{ if } 0 \in A, \sigma A = -1. \end{aligned}$$

Or, if A is symmetric, i.e. $A = [-a,\ a]$, then

$$\begin{aligned} A^n &= [-a^n,\ a^n], \\ \tilde{A}^n &= \begin{cases} [-a^n,\ a^n] & \text{if } n \text{ is odd}, \\ [0,\ a^n] & \text{if } n \text{ is even}. \end{cases} \end{aligned}$$

The following two examples explain the need for two kinds of power evaluation: If $x, y \in A$ then A^2 is the smallest interval that contains xy (w.r.t. the information given, that is $x, y \in A$) If $x \in A$ then $\tilde{A}^2$ is the smallest interval that contains x^2 (w.r.t. the information given, i.e. $x \in A$). Hence, if we for example need inclusions for the polynomial

$$p(x) = x_1^2(2 + x_2x_3) + x_2^2(2 + x_1x_3) + x_3^2(2 + x_1x_2),\ x \in R^3$$

for $x_1, x_2, x_3 \in A$, $A \in I$, then the natural interval inclusion

$$p(A) = 3A^2(2 + A^2)$$

is one possible inclusion. But, in general, a narrower inclusion will be gained with extended arithmetic, that is, by the inclusion

$$P(A) = 3\tilde{A}^2(2 + A).$$

If, for instance, $A = [-1,\ 2]$, then $A^2 = [-2,\ 4]$, $\tilde{A}^2 = [0,\ 4]$, and one obtains

$$p(A) = [-36,\ 72] \text{ and } P(A) = [0,\ 72],$$

where, accidentally, $P(A)$ is already the range of p over the box $A \times A \times A$, that is, $\Box p(A) = P(A)$.

These few examples show the importance of choosing a good formula for the evaluation of an inclusion for the range. Theoretical investigations in this area are found in [235, 14, 149, 217]. The selection of the form that results in the narrowest inclusion is a difficult problem since there is an infinite number of possible forms. In this monograph we focus on the practical aspects of the computation of the forms as they apply to geometric computations.

2.9 Skelboe's Principle

If a programmable function is given and if it is possible to find an expression for the function in a well defined form or if some monotonicity properties are known, then there are ways of determining the range more or less directly. This is, in short, Skelboe's principle, which we will present here in a very developed form. Since the principle can be applied to several problems in geometric computations we provide a complete proof of the principle here.

The notation $a \vee b$ is again used for the interval with endpoints a and b (especially if one does not know whether $a \le b$ or $b \le a$) for $a, b \in R$. Similarly, if $A, B \in I$, we write $A \vee B$ for the smallest interval that contains A and B.

Let us first consider monotone continuous functions f. If f is a function in only one variable then the result is evident, i.e. the range is found immediately as

$$\Box f([a,\ b]) = f(a) \vee f(b)$$

if defined. If a function $f(x_1, \ldots, x_m)$ is defined for $x_i \in X_i \in I$ $(i = 1, \ldots, m)$ we say that f is monotone in the variable x_k if for each choice of values $c_i \in X_i$ $(i = 1, \ldots, k-1, k+1, \ldots m)$ the function

$$g(x_k) = f(c_1, \ldots c_{k-1}, x_k, c_{k+1}, \ldots, c_m)$$

is monotone. The basis for using monotonicity properties is found in the following theorem which is due to Skelboe [250].

THEOREM 1 *Let the continuous function $f(x_1, \dots, x_m)$ be defined for $x_i \in X_i$ $(i = 1, \dots, m)$ and monotone in x_1 (without restricting the generality). If $X_1 = [a,\ b]$ and if the function g_c is defined for any $c \in X_1$ by $g_c(x_2, \dots, x_m) = f(c, x_2, \dots, x_m)$ for $x_i \in X_i$ $(i = 2, \dots, m)$, then*

$$\square f(X_1, \dots, X_m) = \square g_a(X_2, \dots, X_m) \vee \square g_b(X_2, \dots, X_m).$$

Proof. Assume $y = f(c_1, \dots, c_m)$ for some $c_i \in X_i$ $(i = 1, \dots, m)$. Since $a \leq c_1 \leq b$ and using the monotonicity we either obtain

$$g_a(c_2, \dots, c_m) \leq f(c_1, \dots, c_m) \leq g_b(c_2, \dots, c_m)$$

or the opposite chain (the inequalities reversed), and the value y lies in the interval hulls $g_a(c_2, \dots, c_m) \vee g_b(c_2, \dots, c_m) \subseteq \tilde{g}_a(X_2, \dots, X_m) \vee \tilde{g}_b(X_2, \dots, X_m)$. The continuity of f is used to find a term $f(c_1, \dots, c_m)$ which lies between the given values $f(a, d_2, \dots, d_m)$ and $f(b, e_1, \dots, e_m)$ in order to prove the inclusion in the opposite direction. □

If this theorem is applied repeatedly then it can be generalized to functions that are monotone in several variables.

Therefore, if it is required to determine the range of the function $e^x/(x^2+y)$ for $x \in X = [-1,\ 1]$ and $y \in [3,\ 4]$, then it is sufficient to determine the interval hull of the ranges of the functions $g_3(x) = e^x/(x^2+3)$ and $g_4(x) = e^x/(x^2+4)$ for $x \in X$. The ranges $\square g_3(X)$ and $\square g_4(X)$ can easily be calculated since g_3 and g_4 are monotone (their derivatives are positive!).

THEOREM 2 *If a continuous function f is representable in the form*

$$f(x_1, \dots, x_m) = g(x_1, \dots, x_{m-1}) * h(x_m) \tag{2.24}$$

for $x_i \in X_i$ $(i = 1, \dots, m)$ and $* \in \{+, -, \cdot, /\}$, where h is a monotone function, then

$$\square f(X_1, \dots, X_m) = \square g(X_1, \dots, X_{m-1}) * \square h(X_m).$$

Proof. Since x_m is separated out in (2.24), the interval hull given in Theorem 1 has the desired form. □

The advantage of the two theorems is that the range determination is reduced to a range determination for a function with fewer variables plus the trivial range determination of the monotone function h. The theorems can also be used if one can only find inclusions for g_a or g. Then the formulas can be applied to obtain inclusions for f.

Since each occurrence of a variable in which the function is monotone enables the reduction of the number of variables in the range computation, it is suggested to manipulate the expression for f in such manner that Theorem 2 can be used as often as possible, cf. the following example from Skelboe [250], Moore [166]. Let

$$f(x, y, z) = \frac{x+y}{x-y} z \text{ for } x \in X = [1,\ 2], y \in Y = [5,\ 10], \text{ and } z \in Z = [2,\ 3],$$

then an optimal arrangement of f is

$$f(x,y,z) = \left(1 + \frac{2}{(x/y) - 1}\right) z$$

such that the range is determined directly by

$$\Box f(X,Y,Z) = \left(1 + \frac{2}{(X/Y) - 1}\right) Z.$$

If f is a function in the two variables (for simplicity) $x \in X$ and $y \in Y$ then it may happen that f is monotone in y but that f is not representable in the form (2.24), for example the function $f(x,y) = e^{(x-1)y}(x+1)$. Then the use of Theorem 1 is again recommended.

Another classical theorem (Skelboe [250]) involves arithmetic expressions in which each variable occurs at most once and of power at most 1. Hence, $(x_1+x_2)/x_3$ is such an expression, but $(x_1x_2+x_3)/(x_1-x_4)$ or $(1+x_1^2)$ are not. It is obvious that such expressions are monotone with respect to each variable.

THEOREM 3 *Let $f(x_1,\ldots,x_m)$ be an arithmetic expression in which each variable occurs only once and of power at most 1. Then*

$$\Box f(X_1,\ldots,X_m) = f(X_1,\ldots,X_m) \textit{ for } X_1,\ldots,X_m \in I$$

if defined.

Proof. One only has to show that $f(X_1,\ldots,X_m) \subseteq \Box f(X_1,\ldots,X_m)$. Since each variable occurs only once, there exist reals $c_i \in X_i$ such that

$$y = f(c_1,\ldots,c_m) \in \Box f(X_1,\ldots,X_m). \Box$$

We can generalize this principle essentially by admitting power operations and (continuous) transcendental functions. The variables are only allowed to occur once. Two further conditions must be satisfied when determining the natural interval extension:

(i) Each occurring power is to be evaluated with extended interval arithmetic,

(ii) the natural interval extensions of the transcendental functions must be equal to their range (related to the current domain).

Condition (ii) is always satisfiable for the common standard transitive functions such as sqrt, log, ln, sin, cos, exp, etc.

THEOREM 4 *Let an expression $f(x)$, $x \in R^m$ be given for a programmable function f which may contain arithmetic operations, continuous transcendental functions and powers. If the box $X \in I^m$ lies in the domain of f and if the above mentioned conditions (i), (ii) are satisfied then*

$$\Box f(X_1,\ldots,X_m) = f(X_1,\ldots,X_m).$$

Proof. The proof is the same as for the previous theorem. The existence of the argument $c \in X$ follows from the property that a programmable function is defined recursively (such that mathematical induction can be applied) and that for each step of the recursion the natural interval extension gives the range. It then follows that the current range is obtained from requirements (i) and (ii) for powers and transcendental functions and from the definitions of the interval arithmetic operations. □

Example 1. The range of the function

$$f(x, y, z) = \cos^3 \sqrt{|x^3 z + \cos(1/\exp(y^4 - 1))|}$$

over arbitrary intervals $X, Y, Z \in I$ is obtained by the natural interval extension

$$f(X, Y, Z) = \tilde{\cos}^3 \sqrt{|\tilde{X}^3 Z + \cos(1/\exp(\tilde{Y}^4 - 1))|}$$

where condition (ii) must be considered.

Example 2. If $f(x, y, z) = e^z(x + y)/(x - y)$ with $X = [1,\ 2], Y = [5,\ 10]$, and $Z = [\log 2,\ \log 3]$ then $f(X, Y, Z) = [-108,\ -12]/9$. If f is represented by the expression

$$f_1(x, y, z) = e^z(1 + \frac{2}{x/y - 1})$$

then Theorem 4 can again be used and it yields

$$\Box f(X, Y, Z) = f_1(X, Y, Z) = [-63,\ -22]/9.$$

Example 3. $f(x, y, z) = xye^{x-y} \sin z$ and $X = Y = Z = [1, 10]$ and if f is represented as $f(x, y, z) = (xe^x)(ye^{-y}) \sin z$, then the first two parenthesised factors are monotone over $X = Y$ and Theorem 2 can be applied twice yielding $\Box f(X, Y, Z) = [e, 10e^{10}][10e^{-10}, e^{-1}][-1, 1] = [-10e^9, 10e^9]$.

It is often assumed that the numerical costs of using derivatives are high. Derivative based methods are therefore avoided if possible. If, however, automatic differentiation or Krawczyk's slope arithmetic, see for example Krawczyk [140], Rall [201], Ratschek-Rokne [212] or Griewank-Corliss [83] is available then the number of arithmetic operations in the evaluation of the derivative is $\mathcal{O}(n)$. The next theorem requires the partial derivative $D_i f$ of f w.r.t. x_i to obtain an inclusion which can be derived directly from Theorem 1.

THEOREM 5 *(Monotonicity). If $0 \notin D_i f(X)$ for some i and $X_i = [\mathrm{lb}X_i, \mathrm{ub}X_i$ then f is strictly monotone in the i-th variable and*

$$\Box f(X) \subset f(X_1, \ldots, \mathrm{lb}X_i, \ldots, X_n) \vee f(X_1, \ldots, \mathrm{ub}X_i, \ldots, X_n). \Box$$

If the boxes are getting smaller, one has frequently monotonicity in all variables (think of contour tracing, for instance), which can easily be discovered by checking the natural interval extension of the partial derivatives. If, for example, $n = 3$ and $D_1 f(X) > 0$, $D_2 f(Y) < 0$, and $D_3 f(Z) > 0$, then

$$\Box f(X, Y, Z) = [f(\text{lb}X, \text{ub}Y, \text{lb}Z),\ f(\text{ub}X, \text{lb}Y, \text{ub}Z)].$$

2.10 Inner Approximations to the Range of Linear Functions

With a few exceptions, almost the whole rounding error related research in interval arithmetic is based on outward rounding. It is as good as self-evident that each interval software package uses outward rounding for the interval arithmetic operations. The reason is clear historically since the computed interval result is required to contain the unknown exact result, cf. Sec. 2.2. The outward rounding has almost become a doctrine among the users of interval mathematics. Hence, as one began to develop approximations of the range of functions, one arrived at outer approximations. This was, on the one hand, manifested by the meanvalue form, where when starting at the classical meanvalue formula

$$f(x) = f(c) + (x - c)f'(\xi) \text{ with } \xi \in x \vee c$$

the inclusion

$$f(x) \in f(c) + (x - c)f'(X)$$

was obtained and, for the range $\Box f(X)$, the outer approximation

$$f(c) + (X - c)f'(X),$$

cf. Sec, 2.2 and 2.11 was computed. On the other hand, there were early applications of outer approximations such as

$$\Box f(X) \subseteq f(c) + (X - c)f'(X)$$

or

$$\Box f(X) \subseteq f(X)$$

to subdivision or exhaustion methods, cf. Moore [165], where areas are removed that cannot contain function values. In the next section we will see that inner approximations of the range are almost more important than outer approximations. The reason for this is the following standard conclusion which we will find several times in the sequel.

Let A be an inner approximation of some range $\Box f(X)$ and B an outer approximation. Then clearly

$$A \subseteq \Box f(X) \subseteq B.$$

Imagine that $\square f(X)$ is not known. What can A and B contribute to explore $\square f(X)$? The information gained from B is the implication

$$\text{if } w \notin B \text{ then } w \notin \square f(X),$$

and the information gained from A is

$$\text{if } w \in A \text{ then } w \in \square f(X).$$

We see that B is related to the complement of $\square f(X)$ while A is important for the affirmative case that a number w is, in fact, a function value.

How can we get inner inclusions of the range when working with an interval arithmetic which is outwardly oriented? There is no simple answer which would cover the general case: One has first to elaborate formulas for the inner inclusion, that covers the mathematical background of the approximation, cf. Krawczyk [139], Markov [163]. Secondly, one needs an inward rounding, which certainly could be constructed with directed rounding devices as is the case in scientific computation languages, for instance, C-XSC. The average user interested mainly in geometrical computation might not be prepared to implement it.

It is already troublesome to find inner inclusions for the simple case of a linear function where the exact range is available. We therefore include this separate section which provides formulas whose evaluation approximates the range of linear functions from the inside.

Let $\varphi(x,y) = ax+by+c$ be a linear function with coefficients $a, b, c \in R$. We derive formulas for an inner approximation of the range of φ over a rectangle $X \times Y \in I^2$. Since conversion errors can occur, we also allow the coefficients to vary over intervals. This means that the subsequent formulas can also be applied if a, b and c result from preceding computations as will frequently be the case (for example, in Sec. 5.8, "Box-Sphere Intersection"). The intervals X and Y are already assumed to be machine intervals. If this is not the case, then they are shrunk to machine intervals when the input procedure of the data set is executed.

Let $X = [x_1, x_2]$, $Y = [y_1, y_2]$, rad $X = w(X)/2$, rad $Y = w(Y)/2$ and $\alpha = \varphi(\text{ mid } X, \text{ mid } Y)$. Since each variable occurs only once in the expression defining $\varphi(x,y)$, the range can be obtained as natural interval extension,

$$\square\varphi(X,Y) = \varphi(X,Y) = aX + bY + c.$$

A meanvalue development around the midpoint of $X \times Y$ gives

$$\begin{aligned} \square\varphi(X,Y) &= \alpha + \varphi(X,Y) - \alpha \\ &= \alpha + \varphi(X - \text{ mid } X, Y - \text{ mid } Y) \\ &= \alpha + a(X - \text{ mid } X) + b(Y - \text{ mid } Y) \\ &= \alpha + [-|a| \text{ rad } X - |b| \text{ rad } Y, \\ &\quad\; |a| \text{ rad } X + |b| \text{ rad } Y]. \end{aligned} \tag{2.25}$$

The influence of rounding errors and the consideration of inward rounding can be investigated easily in formula (2.25):

Instead of the coefficients, we allow including machine intervals as parameters,

$$a \in [a_1, a_2], \quad b \in [b_1, b_2] \text{ and } c \in [c_1, c_2].$$

The following algorithm computes an inner approximation of the range of φ over $X \times Y$ if a standard machine interval arithmetic is used for the execution of the steps.

ALGORITHM 1 *(For functions in 2 variables)*

Step 1. *Compute* $[\alpha_1, \alpha_2] := [a_1, a_2]$ *mid* $X + [b_1, b_2]$ *mid* $Y + [c_1, c_2]$.

Step 2. *Compute* $[u_1, u_2] := \delta([a_1, a_2])$ *rad* $X + \delta([b_1, b_2])$ *rad* Y.

Step 3. *Compute*

$$\begin{aligned} [\tilde{w}_1, w_1] &:= \alpha_2 - u_1, \\ [w_2, \tilde{w}_2] &:= \alpha_1 + u_1. \end{aligned}$$

Step 4. *If* $w_1 \leq w_2$ *then* $[w_1, w_2] \subseteq \Box\varphi(X, Y)$. *Otherwise no inner approximation is available.*

Note that mid X, rad $X, \cdots$ will in general be intervals.

The correctness of this algorithm becomes evident if one realizes that an inner approximation of $[\alpha_1, \alpha_2]$ and an inner approximation of the interval part in (2.25) are necessary in order to get one of $\Box\varphi(X, Y)$. Since it is unlikely that an inner approximation of $[\alpha_1, \alpha_2]$ is found, α_1 is used as left approximation of α_2, and α_2 as right approximation of α_1. This has the same effect provided $w_1 \leq w_2$. In order to get an inner approximation of the interval part in (2.25), one has to choose values out of the intervals $[a_1, a_2]$ and $[b_1, b_2]$ so, that the interval width is as small as possible. This is achieved with $\delta([a_1, a_2])$ and $\delta([b_1, b_2])$.

For later applications, we present the algorithm for functions in 3 variables as well.

Let $\varphi(x, y, z) = ax + by + cz + d$ and let the coefficients vary in intervals,

$$a \in [a_1, a_2], \; b \in [b_1, b_2], \; c \in [c_1, c_2] \text{ and } d \in [d_1, d_2]$$

and let $X \times Y \times Z \in I^3$ be a box. The half width of an interval is again denoted by "rad".

The following algorithm computes an inner approximation of $\Box\varphi(X, Y, Z)$ provided X, Y, Z are machine intervals and the arithmetic operations are executed with a usual machine interval arithmetic. If they are not, let the box shrink to the largest machine representable box.

ALGORITHM 2 *(For functions in 3 variables)*

Step 1. *Compute*

$$[\alpha_1, \alpha_2] := [a_1, a_2] \; mid \; X + [b_1, b_2] \; mid \; Y + [c_1, c_2] \; mid \; Z + [d_1, d_2].$$

Step 2. *Compute*

$$[u_1, u_2] := \delta([a_1, a_2]) \; rad \; X + \delta([b_1, b_2]) \; rad \; Y + \delta([c_1, c_2]) \; rad \; Z.$$

Step 3. *Compute*

$$\begin{aligned} [\tilde{w}_1, w_1] &:= \alpha_2 - u_1, \\ [w_2, \tilde{w}_2] &:= \alpha_1 + u_1. \end{aligned}$$

Step 4. *If $w_1 \leq w_2$ then $[w_1, w_2] \subseteq \Box\varphi(X, Y, Z)$. Otherwise no inner approximation is available.*

Remark. One could imagine another conceptually very simple way to determine an inner approximation of the range. It would consist of evaluating the functions at the 4 corners of the rectangle (or 8 corners of the box) with machine interval arithmetic, and to derive the inner approximation from this information. An algorithmic description of such a procedure is, already in the case of 2 variables, more involved than the algorithm. The reason is that the computation might have to be split up into cases in order to figure out which of the corners are to be used for the inner approximation, especially, if a or b contains zero.

2.11 Interval Philosophy in Geometric Computations

In this section, we apply the results of 2.9 to a few simple standard principles of geometric computations in order to demonstrate how the global interval aspect and geometrical ideas fit together. One will see that a slightly unusual kind of thinking in terms of set theoretic concepts is necessary for understanding the interval ideas as applied to geometric concepts. This type of thinking is applicable to geometrical objects that can be represented by intervals. (Not only intervals, rectangles and boxes, but also balls and simplices belong to this class.) The set theoretic aspects are now implemented by point-like computations. The term "point-like" indicates that it does not matter whether a continuous function $f(x, y)$ is called up at a point $(a, b) \in R^2$ yielding a point $f(a, b)$ or at an interval pair $(A, B) \in I^2$ yielding an interval $f(A, B)$. This means that there is no real computational difference between invoking $f(a, b)$

or $f(A, B)$ if an interval software package is used, since (a, b) as well as (A, B) are both arguments for f. This relationship between $f(a, b)$ and $f(A, B)$ opens up new vistas for geometrical thinking, which has nothing in common with the interval arithmetic rounding error control philosophy. In order to introduce this to the reader who might not be familiar with this kind of thinking, we give a thorough and broad discussion of only a few but important examples in this section. However, these examples already express the crux of the philosophy. Let us turn to the first example.

Example 1. In Sec. 2.8 we considered a simple determinant

$$D(a_1, a_2, b_1, b_2) = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$$

and we compared three different expressions for the expansion of the determinant which led to natural interval extensions of different widths. The simplest expression was

$$D_1(a_1, a_2, b_1, b_2) = a_1 b_2 - a_2 b_1 .$$

Later we need the collection of values of D for all $a_1 \in A_1$, $a_2 \in A_2$, $b_1 \in B_1$, $b_2 \in B_2$, that is, just the range of D over $A_1 \times A_2 \times B_1 \times B_2$. In order to get this range we apply the theorem to D_1 and interpret the elements a_i, b_i as variables over the domains A_i and B_i, respectively (i=1,2). Since each variable occurs once, the natural interval extension of D_1 gives the range

$$\Box D(A_1, A_2, B_1, B_2) = D_1(A_1, A_2, B_1, B_2)$$

by Skelboe's principle.

Example 2. A simple example in the plane is given by the computation to test whether the line defined by the two points $P_1 = (a_1, b_1)$ and $P_2 = (a_2, b_2)$ passes or does not pass through the rectangle $\mathcal{R} = (X, Y)$ with $X, Y \in I$.

Again the solution to this problem can be found by a range computation in the following manner: For each point $Q = (x, y) \in R^2$ three possibilities exist: Q lies on one side of the line through P_1, P_2 or Q lies to the other side of the line, or Q lies on the line. Consider the oriented triangle $\mathcal{T}$ with vertices P_1, P_2, Q (the orientation being decided by this order). Then the oriented area of $\mathcal{T}$ is defined as

$$D(P_1, P_2, Q) = \frac{1}{2} \begin{vmatrix} a_1 & b_1 & 1 \\ a_2 & b_2 & 1 \\ x & y & 1 \end{vmatrix} .$$

Note that $|D(P_1, P_2, Q)|$ coincides with the well-known formula for the (unoriented) area of $\mathcal{T}$ as can be found in any formula collection. Now, for all points $Q \in R^2$ lying on one side of the line, D has the same sign, for all Q lying on

the other side of the line D has the opposite sign, and D is zero if Q lies on the line. This gives rise to the following criterion:

If a $Q \in \mathcal{R}$ exists such that $D(P_1, P_2, Q) = 0$ (saying that Q is on the line) then the line meets the rectangle, and the converse. Equivalently, the line passes through the rectangle iff $0 \in \Box D(P_1, P_2, \mathcal{R})$. Note that $\Box D(P_1, P_2, \mathcal{R})$ is the range of the determinant function over a continuum, and that the question posed can be completely answered just by checking whether zero is in the range or not!

In order to determine the range, one just has to find an appropriate expression such that the theory can be applied. First of all one has to find out what the "variables" are. Obviously, in this case they are x and y since they vary over X and Y, respectively. The other symbols, a_i and b_i, are constants in the expression, since they are the coordinates which fixes the line.

It is clear that the usual expression for the determinant,

$$a_1 b_2 + b_1 x + a_2 y - b_2 x - b_1 a_2 - a_1 y$$

does not satisfy Skelboe's principles directly. However, we can write down a different expression such that each variable occurs just once, namely

$$D_4(P_1, P_2, Q) = (y(a_2 - a_1) + x(b_1 - b_2) + a_1 b_2 - a_2 b_1)/2. \tag{2.26}$$

By Theorem 3, the natural interval extension of D_4 gives the range,

$$\Box D(P_1, P_2, \mathcal{R}) = D_4(P_1, P_2, \mathcal{R}). \tag{2.27}$$

We note that we have used exact arithmetic and exact interval arithmetic to describe and to solve the given geometric situation. It is usual to do this when one thinks geometrically and mathematically. If, however, the computations are executed on a computer then due to the outward rounding, one gets an outer approximation

$$D_M = D_4(P_1, P_2, \mathcal{R})_M$$

of $\Box D(P_1, P_2, \mathcal{R})$, i.e. $D_M \supseteq \Box D$. Depending on the purpose of the computation, one can frequently identify the cases $0 \in \Box D$ and $0 \notin \Box D$ with $0 \in D_M$ and $0 \notin D_M$, respectively. But if one wants guaranteed answers to the intersection question, one only has the conclusion

$$0 \notin D_M \text{ imples } 0 \notin \Box D$$

that is, the line does not pass through the rectangle. The case $0 \in D_M$ does not allow any guaranteed decision since

$$0 \in \Box D \text{ as well as } 0 \notin \Box D$$

are consistent with $0 \in D_M$. The only way out is the construction of an inner approximation D_I of $\Box D$ as described in Sec. 2.10. Now one can differentiate between the relations

$$0 \in \Box D_I \text{ and } 0 \notin \Box D_I.$$

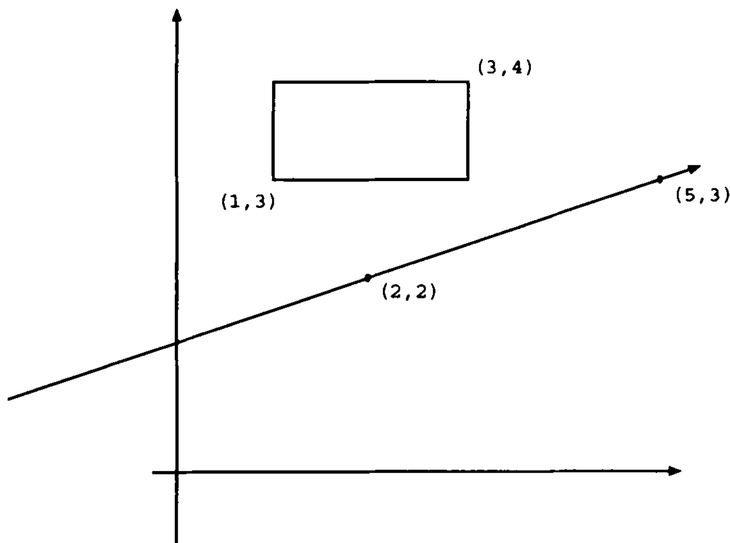


Figure 2.1: Line-box intersection problem

In the first case, $0 \in \square D$ is guaranteed because of $D_I \subseteq \square D$. In the second case, no guaranteed decision is possible as long as one keeps the current accuracy and the methods chosen for getting D_I and D_M. One need not worry too much about this uncertainty since the the area $D_M \setminus D_I$ will be very tiny in general, such that the probability is small that the case $0 \in D_M \setminus D_I$ happens. If one, however, relies on a decision, one can repeat the calculation with double or multiple precision, but still has a small chance of remaining an unknown area. One also can change the method applied completely and switch to a couple of left-turn tests (cf. the next example), that can be executed exactly with the high performance method ESSA, cf. Ch. 4. The test had then the following form: If the four corners of the rectangle lie strictly to the left of the line directed from P_1 to P_2 or if the four corners lie strictly to the right of this line, then the line does not pass through the rectangle.

A particular example is given by the values shown in Figure 2.1 where $X = [1,3]$ and $Y = [3,4]$. This means that $2D_4(P_1, P_2, \mathcal{R}) = [3,4](5-2) + [1,3](2-3) + 2 \times 3 - 5 \times 2 = [2,8]$. Since $0 \notin D_4(P_1, P_2, \mathcal{R})$ it follows that the line through P_1 and P_2 does not intersect $\mathcal{R}$.

Example 3. The determinant $D(P_1, P_2, Q)$ is closely connected with the so-called *left-turn test*, which is one of the most important primitives in 2D computational geometry. It is, for instance, needed in algorithms for determining the convex hull of a finite number of points in the plane, cf. Sec. 8.2.

The left-turn test decides, for 3 given points in the plane, say P_1, $P_2 \neq P_1$

and Q, whether Q lies left of the line through P_1 and P_2, where the line is oriented from P_1 to P_2 (that is, one looks from P_1 in direction P_2, which makes the property "left of" meaningful). Now, Q lies to the left of the line from P_1 to P_2 iff the triangle, say $\mathcal{T}$, with vertices P_1, P_2 and Q is surrounded counter-clockwise when passing through the vertices in the order P_1, P_2, Q, P_1. In this case, the triangle is said to be positively oriented (by convention) and the area of $\mathcal{T}$ is positive, i.e.

$$D(P_1, P_2, Q) > 0.$$

Clearly, if Q is on the line, the 3 points P_1, P_2 and Q are colinear, and the determinant becomes zero. If Q is to the right of the line, then P_2 is to the left of the line from P_1 to Q, and hence, $D(P_1, Q, P_2) > 0$. Swapping the 2 lines in the determinant shows that $D(P_1, P_2, Q) = -D(P_1, Q, P_2) > 0$ Thus, the left-turn test can also be used to check, whether the point Q lies on the line or to the right of the line.

Instead of the point Q we can again consider a rectangle $\mathcal{R}$ arguing as in Example 2, which means that we have again derived that,

$\mathcal{R}$ lies completely to the left to the line through P_1 and P_2 when the line is oriented from P_1 to P_2, iff

$$\square D(P_1, P_2, \mathcal{R}) = D_4(P_1, P_2, \mathcal{R}) > 0.$$

This criterion is confirmed by Fig. 2.1 reflecting the example and the above discussion.

Example 4. It is interesting to note that the frame of the 2D test, whether a line passes through a rectangle or not (Example 2), cannot be transformed to a 3D test, whether a line in R^3 passes through a box or not. The reason is that an appropriate determinant that could be the base of some range determination as in Example 2, is missing.

It is, however, possible to reduce the 3D case to several 2D tests, cf. Haines [99]. The line is defined as a parametric ray

$$P(t) = P_0 + tP_d$$

with $P_0 = (x_0, y_0, z_0)$ and $P_d = (x_d, y_d, z_d)$. The box is defined to be $B = (X, Y, Z)$ with $X, Y, Z \in I$. The procedure is then to check the 2D line-rectangle intersections using (2.26) with the generating points $P(0)$ and $P(1)$ in the projections into the xy-plane, the yz-plane and the zx-plane. The ray does not intersect the box iff any of the three following 2D tests fails: The computations reduce to checking if

$$\begin{aligned} 0 &\in 2(x_d(-y_d - Y) + y_d(x_0 - X)) && \text{in the } xy\text{-plane,} \\ 0 &\in 2(y_d(-z_d - Z) + z_d(y_0 - Y)) && \text{in the } yz\text{-plane,} \\ 0 &\in 2(z_d(-x_d - X) + x_d(z_0 - Z)) && \text{in the } zx\text{-plane.} \end{aligned}$$

A numerical example given by Haines [99] is

$$P_0 = (0, 4, 2), \quad P_d = (0.218, -0.436, 0.873)$$

with the box being $B = ([-1, 3], [2, 3], [1, 3])$. The results are

in the xy-plane	$[-1.744, 2.180]$	contains 0,
in the yz-plane	$[0.874, 4.364]$	does not contain 0,
in the zx-plane	$[-2.182, 5.674]$	contains 0.

The computation could be terminated after having verified that 0 was not in the projection of the box in the yz-plane.

2.12 Centered Forms and Other Inclusions

Among the many ways to construct inclusion functions the so-called centered forms play an important role since they are inclusion functions of order 2. The idea of a centered form was first given in Moore's original book *Interval Analysis* which appeared in 1966 [165]. He noted that if functions were developed in a certain manner then the result gave in general narrower inclusions than many other possible inclusions. Explicit formulas were found for polynomials by Hansen [88], for rational functions [209] and for multivariate rational functions [210]. A general definition of centered forms was given by Krawczyk-Nickel [142] in 1982.

When Moore [165] computed inclusions for the range of example functions using the natural interval extensions of various algebraically equivalent formulations of a given function he also noticed that he often got narrower results if the function was developed in a certain manner around the midpoint of the domain interval. Moore [165] discussed this phenomenon in some detail and he also compared the results with other methods for including the range.

In 1973 Goldstein-Richman [74] confirmed the results of Moore's [165] experiments.

Historically, the basic feature of any centered form for $f(x)$ was that $f(x)$ was rewritten as

$$f(x) = f(c) + s(x)$$

for a given c, which was most often the center of the domain in which f was to be developed. If $s(x)$ is further developed as

$$s(x) = (x - c)g(x)$$

then the forms have a certain second order convergence property as will be made clear later. The approach was therefore to write $f(x)$ as

$$f(x) = f(c) + (x - c)g(x)$$

and then to evaluate the natural interval extension of the resulting expression over X as

$$F(X) = f(c) + (X - c)g(X).$$

From the fundamental principle of interval analysis it followed immediately that

$$\Box f(X) \subseteq F(X).$$

It took a surprising amount of time before explicit definitions of g were found for functions other than polynomials (Ratschek [210]). It also turned out that the explicit definitions that were found were too complicated to be useful. For instance, if f was a rational function, $f = p/q$, and if the maximum degree of the the polynomials p and q was n, then all partial derivatives of p and q up to the order n were required for an explicit definition of g.

The so-called meanvalue form was commonly used as an inclusion function of order two. This is nothing more than a natural interval extension of the well-known meanvalue formula. More general Taylor forms were introduced by [202]. These forms were also inclusion functions of order two being the interval extension of the Taylor expansion of the function f.

Both the meanvalue form and the Taylor form did not fit into the historical setting by Moore discussed above.

It was left to Krawczyk-Nickel [142] to find an ingenious and precise definition of a general centered form, which not only covered Moore's historical form, but also the meanvalue form, the Taylor form and various other inclusion functions of order two. The general definition is, however, too complicated to be included in this monograph. Hence we treat the meanvalue form and the Taylor form separately and we refer the interested reader to Ratschek-Rokne [212].

We therefore introduce:

1. The meanvalue form.

2. The Taylor form of second order.

We first give a formal definition of the meanvalue form. Let $f : D \to R, D \subseteq R^m$ (as before m is usually 2 or 3 in geometric computations) be differentiable and let $F' : \mathbf{I}(D) \to I^m$ be an inclusion function for the gradient, f'. Then $T_1 : I(D) \to I$ defined by

$$T_1(Y) = f(c) + (Y - c)^T F'(Y) \text{ for } Y \in I(D) \tag{2.28}$$

where $c = \text{mid } (Y)$ or also some other point of Y is called the ***meanvalue form function*** (or shorter: ***meanvalue form***) of f, cf. Moore [165, 169]. Frequently, $F'(Y)$ can be computed via natural interval extensions of $f'(\xi)$ (see below) or via an automatic differentiation arithmetic, or via similar techniques that avoid

explicit differentiation, see for example [83]. Because of the meanvalue formula of analysis we have, if $Y \in I(D)$ is given, for $x \in Y$,

$$f(x) = f(c) + (x - c)^T f'(\xi) \in f(c) + (Y - c)^T F'(Y)$$

where ξ is a point on the line segment connecting x and c. Since this formula holds for any $x \in Y$ it is thus obvious that $\Box f(Y) \subseteq T_1(Y)$. Therefore the meanvalue form is an inclusion function for f. Its importance arises from its second order property which is obtained with a low computational effort:

THEOREM 6 *(Krawczyk-Nickel [142]). If F' is Lipschitz then the meanvalue form T_1 satisfies:*

$$w(T_1(Y)) - w(\Box f(Y)) = \mathcal{O}(w(Y)^2).$$

This theorem states that the meanvalue form converges to the range with a second order convergence as the width of the domain interval tends to zero. Later we will simply refer to this as the second order convergence property. This property is extremely useful for subdivision methods when the widths of the patches are reduced. An extensive proof of this theorem can be found in Ratschek-Rokne [212].

Example. Let $f(x) = x - x^2$ be defined on $D = \{x : x \geq 1\} \subseteq R$. ($D$ instead of R is chosen for simplicity in order to avoid different cases.) An inclusion function for $f'(x) = 1 - 2x$ is

$$F'(Y) = 1 - 2Y \quad \text{for } Y \in I(D)$$

(natural interval extension of f') and is Lipschitz. The meanvalue form of f is then

$$T_1(Y) = (c - c^2) + (Y - c)(1 - 2Y) \quad \text{for } Y \in I(D),$$

where $c =$ mid Y. The natural interval extension of $f(x)$ to Y is

$$f(Y) = Y - Y^2 \quad \text{for } Y \in I(D).$$

Finally,

$$\Box f(Y) = [y - y^2,\ x - x^2] \quad \text{for } Y = [x,\ y] \in I(D)$$

since f is monotonically decreasing in D. Let us look at the widths of the inclusion functions:

$$\begin{aligned} w(\Box f(Y)) &= x - x^2 - (y - y^2) = y^2 - x^2 - (y - x) \\ &= w(Y)(y + x - 1). \end{aligned}$$

Using (2.8), (2.13) and the fact that

$$|1 - 2Y| = \max\{|\ 1 - 2x\ |,\ |\ 1 - 2y\ |\} = 2y - 1$$

for $Y = [x, y] \in I(D)$ we get

$$\begin{aligned} w(T_1(Y)) &= w[(Y - c)(1 - 2Y)] \\ &= w(Y - c)|1 - 2Y| \\ &= w(Y - c)(2y - 1) \\ &= w(Y)(2y - 1). \end{aligned}$$

The width of the natural interval extension is

$$\begin{aligned} w(f(Y)) &= w(Y) + w(Y^2) \text{ by } (2.8) \\ &= w(Y) + |Y|^2 - (\delta Y)^2 \text{ by } (2.13) \\ &= (y - x) + (y^2 - x^2) \\ &= w(Y)(y + x + 1). \end{aligned}$$

The width of the range is

$$w(\Box f(Y))) = x - x^2 - (y - y^2) = w(Y)(x + y - 1).$$

Therefore,

$$w(T_1(Y)) - w(\Box f(Y))) = w(Y)(2y - x - y) = w(Y)^2 = \mathcal{O}(w(Y)^2),$$

and

$$w(f(Y)) - w(\Box f(Y))) = 2w(Y) = \mathcal{O}(w(Y)).$$

One recognizes that T_1 is of order 2 and that $f(Y)$ is of order 1.

A short calculation shows that

$$w(T_1(Y)) \le w(f(Y)) \text{ iff } w(Y) \le 2,$$

which means that the meanvalue form is superior for smaller intervals Y. This is consistent with the fact that the meanvalue form is of convergence order 2, but the natural interval extension is only of order 1.

From this example it is clear that it is not always wise to take a meanvalue form - especially for boxes Y with larger width - since its excess-width tends quadratically to ∞ as $w(Y) \to \infty$ whereas the excess-width of the natural interval extension tends only linearly to ∞. This situation is typical for the whole area of inclusion functions such that meanvalue forms as well as Taylor forms, which will be defined below, should only be used if $w(Y) \le 1/(2m)$. This is an average recommendation and results from our own numerical experience.

Remark 1. One obtains, in general, meanvalue forms with smaller widths if *slopes* instead of $F'(Y)$ are used in (2.28). The interested reader is referred to Alefeld-Herzberger [6], Krawczyk [140], Ratschek-Rokne [212].

Remark 2. The quality of the chosen centered form, for instance the meanvalue form, depends on the shape of the function being included such

that for special functions special centered forms are superior, cf. for example Alefeld-Rokne [7], Rokne [233].

Let $f : D \to R, D \subseteq R^m$ be twice differentiable, and let $F'' : I(D) \to I^{m\times m}$ be an inclusion function for the Hessian matrix f''. Then $T_2 : I(D) \to I$ defined by

$$\begin{aligned} T_2(Y) &= f(c) + (Y-c)^T f'(c) \\ &+ \tfrac{1}{2}(Y-c)^T F''(Y)(Y-c) \quad \text{for } Y \in I(D), \end{aligned}$$

where $c = m(Y)$ or any other point in Y, is called a *Taylor form function* (or shorter: *Taylor form*) for f of second order. Because of the Taylor formula of analysis, T_2 is an inclusion function for f. We say that F'' is bounded if a matrix $C \in I^{m\times m}$ exists such that $F''(Y) \subseteq C$ for all $Y \in I(D)$.

THEOREM 7 *(Ratschek-Rokne [212]) If f is twice differentiable, and if f'' has a bounded inclusion function F'' then the Taylor form function, T_2, is of convergence order two.* □

If m is large then it is better to avoid the explicit evaluation of $F''(Y)$ because of the many arithmetic operations one has to perform in order to obtain $T_2(Y)$. For such a recursive computation the automatic differentiation arithmetic is appropriate as well, cf. Rall [201], Griewank-Corliss [83].

Table 2.1 is intended to show how the different approximations to the range behave as the width of the domain changes. We consider the function $f(x) = (x^2 - 2x + 2)e^x$ over intervals $X_r = [1-r, 1+r]$, $r \geq 0$. (It was decided to make the intervals dependent on one parameter such that the results could be displayed in a transparent manner.) We compare the range, $\square f(X_r)$, with the natural interval extension of the function as defined above, $f(X_r) = (X_r^2 - 2X_r + 2)e^{X_r}$, a nested form, $f_n(X_r) = (X_r(X_r - 2) + 2)e^{X_r}$, the meanvalue form, and the Taylor form of second order. Table 2.1 presents the quotient of the width of a form through the width of the range together with a last column indicating the best form of the 4 for that value of r. The results indicate that the overestimation of the range by one of the forms (depending on the range) is surprisingly small. Indeed, Table 1 shows that for any area a reasonable form can be found, and that especially for large intervals the natural interval extension of the original function is an excellent choice.

A large number of other centered forms are possible. We only mention one further possibility for rational functions that does not require the explicit computation of derivatives [233]. We let

$$f(x) = p(x)/q(x) = \sum_{i=0}^{n} a_i x^i \Big/ \sum_{i=0}^{m} b_i x^i \tag{2.29}$$

be a rational function and $k = \max(n, m)$. The polynomials $p(x)$ and $q(x)$ are now developed using Horner's rule at c such that

$$p(x) = p(c) + s(x)(x-c),$$

r	nat/rge	nest/rge	mvf/rge	Tayl/rge	Best
0.0020	5.0001	3.0000	1.0060	1.0015	Taylor
0.0040	5.0000	3.0000	1.0120	1.0030	"
0.0060	4.9999	3.0000	1.0181	1.0045	"
0.0080	4.9998	2.9999	1.0241	1.0060	"
0.0100	4.9997	2.9999	1.0302	1.0076	"
0.0200	4.9989	2.9995	1.0609	1.0152	"
0.0400	4.9957	2.9979	1.1236	1.0310	"
0.0600	4.9904	2.9952	1.1881	1.0473	"
0.0800	4.9830	2.9915	1.2542	1.0641	"
0.1000	4.9736	2.9868	1.3218	1.0816	"
0.2000	4.8973	2.9486	1.6800	1.1792	"
0.4000	5.0096	2.8151	2.4547	1.4356	"
0.6000	5.0506	2.6430	3.2324	1.7839	"
0.8000	4.8896	2.4692	3.9606	2.2215	"
2.0000	3.4634	2.4448	7.3343	6.2222	nested
4.0000	2.4126	2.3537	11.7687	16.4847	natural
6.0000	1.9730	2.2703	15.8920	30.6496	"
8.0000	1.7385	2.2154	19.9385	48.7386	"
10.0000	1.5941	2.1782	23.9604	70.7921	"
15.0000	1.3982	2.1239	33.9823	143.3628	"
20.0000	1.2993	2.0948	43.9900	240.8978	"
25.0000	1.2396	2.0767	53.9936	363.4185	"
30.0000	1.1998	2.0644	63.9956	510.9323	"

Table 2.1: Quotients of widths of inclusions

$$q(x) \quad = \quad q(c) + t(x)(x - c)$$

and where the values $p(c)$, $q(c)$ and the coefficients s_i and t_i of

$$s(x) \;=\; \sum_{i=0}^{n-1} \left(\sum_{j=i+1}^{n} a_j c^{j-i-1} \right) x^i = \sum_{i=0}^{n-1} s_i x^i,$$

$$t(x) \;=\; \sum_{i=0}^{m-1} \left(\sum_{j=i+1}^{m} b_j c^{j-i-1} \right) x^i = \sum_{i=0}^{m-1} t_i x^i$$

are calculated explicitly as part of the Horner process.

If we now write $f(x)$ as

$$f(x) = f(c) + (x - c)\frac{r(x)}{q(x)} \tag{2.30}$$

then $r(x)$ has to obey the relation

$$r(x)(x - c) = p(x) - f(c)q(x).$$

Using the above representation for $p(x)$ and $q(x)$ we get

$$r(x)(x - c) = p(c) + s(x)(x - c) - f(c)(q(c) + t(x)(x - c))$$

and finally

$$r(x) = s(x) - t(x)f(c) = \sum_{i=0}^{k-1} \left(\sum_{j=i+1}^{k} (a_j - f(c)b_j)c^{j-i-1} \right) x^i = \sum_{i=0}^{k-1} r_i x^i$$

where the undefined coefficients a_j or b_j are set to zero. Algorithmically we have $r_i = s_i - t_i f(c)$, $i = 1, \ldots, k - 1$ where s_i and t_i were calculated by the Horner process. The natural interval extension of (2.30) therefore provides an outer estimate of $\Box f(X)$ and it is part of the class of centered forms formed by Moore's definition.

Example. Let $p(x) = 4x + 4x^2 - x^3 - x^5$ and $q(x) = 2 + 2x + 2x^2 + 3x^3$ and let $X = [1, 1 + \epsilon]$. Then

ϵ	inclusion using (2.30)	width	inclusion using (2.29)	width
0.1	[0.58662, 0.67666]	0.09004	[0.47663, 0.80444]	0.32781
0.01	[0.65987, 0.66676]	0.00689	[0.64677, 0.68004]	0.03327
0.001	[0.66599, 0.66667]	0.00068	[0.66467, 0.66800]	0.00333
0.0001	[0.66660, 0.66667]	0.00067	[0.66647, 0.66680]	0.00033

Alander [2] provides further examples of estimating the range of rational functions using the forms given in [210].

2.13 Subdivision for Range Estimation

It was noted in Sec. 2.6 that the main purpose in estimating the range in geometric computations was to test whether a given condition was satisfied or not. For a given function $f(x)$ and an interval X it might be the case that the result is inconclusive, i.e. the test required the determination of $f(x) \neq 0$ for any $x \in X$, but the result was $0 \in F(X)$. It was also noted that one way to improve the possibility of a conclusive result was to subdivide X recursively rejecting subintervals Y for which $0 \notin F(Y)$ until X was exhausted.

The following algorithm provides a skeleton for the more general problem of computing an outer and inner estimate of the range of a function $f(x)$ over an interval X. A second algorithm will be given for a computation of an outer estimate of the zero set of f. Both algorithms only assume the existence of an inclusion function F for f. However, they will only work in a reasonable manner if $w(F(Y)) - w(\Box f(Y)) \to 0$ as $w(Y) \to 0$, cf. [213].

ALGORITHM 3 *(After Moore-Skelboe)*

Step 1. *Set* $Y := X$.

Step 2. *Set* $Z := F(Y)$, *set* $W := Z$, *set* $Q := Z$.

Step 3. *Initialize list* $\mathrm{L} = ((Y, W))$. *Set* $Z = W$.

Step 4. *Choose a coordinate direction* k.

Step 5 *Bisect* Y *normal to the coordinate direction* k *obtaining boxes* V_1, V_2 *such that* $Y = V_1 \cup V_2$.

Step 6. *Calculate* $Z_1 := F(V_1), Z_2 := F(V_2)$.

Step 7. *Remove* (Y, W) *from the list* L.

Step 8. *For* $i = 1, 2$ *if* $Z_i \not\subseteq Q$ *then set* $Q := [\min(\mathrm{ub}Z_i, \mathrm{lb}Q), \max(\mathrm{lb}Z_i, \mathrm{ub}Q)]$. *and enter* (V_i, Z_i) *onto the list.*

Step 9. *Calculate* $Z := \{\bigcup Z_i | (Y_i, Z_i) \in \mathrm{L}\}$.

Step 10. *If termination criteria hold, then go to 13.*

Step 11. *Denote the first pair of the list by* (Y, W).

Step 12. *Go to 4.*

Step 13. *End*

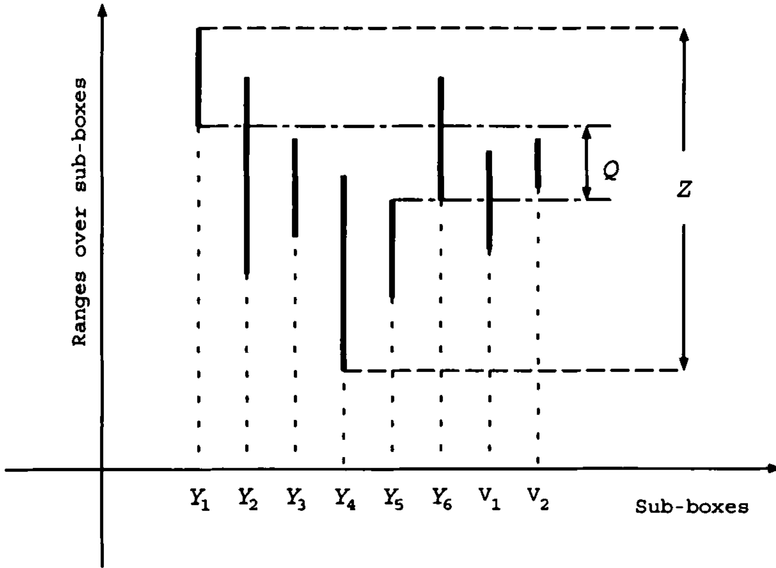


Figure 2.2: Sub-box rejection

The rejection of boxes in Step 8 is a two-sided version of the rejection step in the algorithm in [250].

If the inclusion function F is ***inclusion isotone***, that is, $Y \subseteq Z$ implies $F(Y) \subseteq F(Z)$ for $Y, Z \in I(D)$, then the intersection in step 9 can be dropped. Practically, one will avoid the formation of the union in each iteration, but one will always keep track of the smallest value $\mathrm{lb}Z_i$ and the largest value $\mathrm{ub}Z_j$ during the computation. A schematical description of one step of the algorithm is shown in Figure 2.2. This figure assumes that in Step 4 of the algorithm the list of boxes consisted of the boxes $Y_1, \ldots, Y_6, Y$. The figures then shows the situation after the box Y has been subdivided into boxes V_1, V_2 just prior to Step 8 of the algorithm. The current inclusion for the range over X is Z and Q represents the interior portion of the range which cannot contain the endpoints of the range. In this example it is clear that V_2 will not be added to the list **L** since the endpoints of the range cannot be in V_2.

There are three points deliberately left open in the above algorithm:

(A) The choice of coordinate direction k in Step 5 and the type of subdivision in Step 6.

(B) The recursion termination in Step 10 is not specified.

(C) The order in which the elements are entered onto the list is not specified in Step 8.

These points are now discussed in detail.

(A) A number of strategies for choosing the next coordinate direction are available. They range from simple selection procedures such as a circular choice of coordinate direction (see [169]) to quite expensive computations involving Jacobians for functions having the required smoothness properties (see for example [214]). Each strategy has its own merits depending on the type function computed and the required accuracy. Some of the other strategies are:

1. Bisecting the side of maximum length. This might not be an optimal strategy. Consider eliminating domains of the function

$$f(x) = x_1 + 10^{10}x_2 - 1, x \in [0, 10]^2$$

   using the above algorithm. For any $X = (X_1, X_2)$ it is clear that subdividing X_1 will have little effect on the range of values obtained from the natural interval extension of f whereas recursive bisection of X_2 will eventually result in the elimination of a subbox.

2. Another strategy was given in [214] where the widths sum

$$W^i = w[f(Y^i)] + w[f(Y^{ii})]$$

   is minimized w.r.t. $i = 1, \ldots, n$ where Y^i, Y^{ii} denotes the halves of Y that occur if the i-th edge of Y is bisected.

3. Other bisection recipes are given in Kearfott [125], Csendes [28], Csendes-Ratz [29], [246], Ratschan [205] and Nataraj-Sheela [176].

(B) The algorithm has to have a termination criterion. A number of possibilities for such criteria exist some of which are:

1. The standard criterion will be that the computation shall terminate, if, given some $\epsilon > 0$ as input parameter for the absolute approximation error

$$\begin{aligned} \text{lb}Q - \text{lb}Z &< \epsilon \\ \text{and} & \\ \text{ub}Z - \text{ub}Q &< \epsilon. \end{aligned}$$

   Since $Q \subseteq \Box f(X) \subseteq Z$, the range is included from the inside as well as from the outside with ϵ tolerance.

2. The computations terminate after a fixed number of steps if no success is achieved, that is, if Z does not shrink and Q does not grow.

3. The computations terminate when all boxes Y_i are smaller than a certain tolerance (this strategy is linked to how the elements are entered onto the list in (C)).

4. The computations terminate when the intervals Z_i are all smaller than a given width (also linked to the list ordering in (C)).

(C) The ordering of the boxes on the list is important and depends on the purpose and on the side conditions of the computation. Some of the choices are:

1. A simple choice is to order the boxes by *last in first out*. This results in short lists since the newly generated boxes are worked through immediately. There is no convergence guarantee that Q or Z tend to the range.

2. Another choice is that the widest box is chosen for the bisection. There is no additional ordering effort necessary since the eldest boxes are the ones with largest width. If the inclusion function satisfies $w(F(Y)) \to 0$ as $w(Y) \to 0$, then Q and Z tend to the range.

3. The choice of the next box might be determined by two factors: the width of the box and the extremeness of its range. That is, a widest box Y such that $\text{lb}Y = \text{lb}Z$ or $\text{ub}Y = \text{ub}Z$ of Z in Step 9 is chosen for bisection (see Skelboe [250]).

4. One of the boxes corresponding to the extreme points of Z and Q is chosen. The algorithm circulates among the boxes forming the 4 possible endpoints.

Each of the above choices generate new algorithms with different properties both with respect to expected speed as well as convergence. An initial choice might be (A)(1), (B)(1) and (C)(2), however, this should be modified depending on the function and depending on the use of the result in succeeding computations. If f satisfies certain differentiability conditions then it is possible to enhance the algorithm with other tools. Examples of such tools are the monotonicity property discussed in 2.10 which is expected to be more successful the smaller the box widths are (just consider the trivial one dimensional case of computing the range of $y = x^2$ over $[-1, 1]$ where monotonicity will occur after only one subdivision) and interval Newton iterations which will be discussed in the next chapter.

The second algorithm aims to shrink the domain, X, of f to an outer estimate of the zero set of f in X. (Instead of zero, any other real number can be taken.) This algorithm is slightly different from the previous algorithm in that it aims to reject parts of the domain guaranteed not to contain zero as fast as possible.

This type of algorithm is used if X is rather large or if the zero set is not a discrete set or if it is an involved set such that the application of local methods like Newton's method does not serve any purpose. Typical applications of such algorithms are in contour tracing of implicitly defined surfaces or curves. (This

topic will be treated extensively in Ch. 6.) As an example of this, we consider the problem of embedding the curve defined by $f(x, y) = x^2 + 3xy + y^2 - 10 = 0$ in the plane, in a working area. This working area will be a collection of smaller rectangles which are prepared for a more effective processing of representing or drawing the curve (continuation methods, simplex methods, linearization, ev. Newton's method, etc.) The output of the algorithm is either that $0 \notin \Box f(X)$ or a list of boxes of width smaller than $\epsilon > 0$ that contain the zero set.

ALGORITHM 4 *(Estimating the zero set of F)*

Step 1. *Set $Y := X$.*

Step 2. *Set $Z := F(Y)$, set $W := Z$.*

Step 3. *If $0 \notin F(X)$ then output "zero not contained in $\Box f(X)$" and go to 14.*

Step 4. *Initialize list* $\mathrm{L} := ((Y, W))$.

Step 5. *Choose a coordinate direction k.*

Step 6. *Bisect Y normal to the coordinate direction k obtaining boxes V_1, V_2 such that $Y = V_1 \cup V_2$.*

Step 7. *Calculate $Z_1 := F(V_1), Z_2 := F(V_2)$.*

Step 8. *Remove (Y, W) from the list* L.

Step 9. *For $i = 1, 2$ if $0 \in Z_i$ then enter (V_i, Z_i) onto the list.*

Step 10. *If $\mathrm{L} = \emptyset$ then output "zero not contained in $\Box f(X)$" and go to 14.*

Step 11. *If the widths of all the boxes on the list are smaller than ϵ then go to 14.*

Step 11. *Denote the first pair of the list whose underlying box has width smaller than ϵ by (Y, W).*

Step 13. *Go to 5.*

Step 14. *End*

It is interesting to note that when the computation is executed on a machine and the output is "$0 \notin \Box f(X)$" then it is *guaranteed* that the result is correct (provided the machine interval arithmetic is implemented correctly and provided the program is otherwise correct). This means that any logical decision that depends on the correctness of the above computation is correct and that anomalous results cannot occur.

In this algorithm the details of

(A) the choice of coordinate direction k in Step 5,

(B) the recursion termination in Step 11, and

(C) the order in which the elements are entered onto the list in Step 9,

are again left open. The choices are quite similar to what discussed for the previous algorithm except for the list order. In (C) we therefore recommend that the list be ordered by increasing $\min\{|\mathrm{lb}F(Z_i)|, |\mathrm{ub}F(Z_i)|\}$, expecting the largest success in deleting subinterval when the boundary of the range over a subinterval is closest to zero.

Ab initio one would expect that the ordering of the list is redundant, and that the algorithm would function quite well by simply exhausting parts of the domain in any order, for example, subdivide a given piece until it is rejected and then treat the next piece and so on. This is in fact not the optimal strategy since a termination criterion is present. The simple exhausting strategy would result in a potentially quite unbalanced list of boxes when the algorithm was unsuccessful with large boxes that might have been easily rejected remaining on the list. Further processing would then be less effective.

2.14 Summary

In this chapter we introduced interval arithmetic both theoretically and in a practical implementation. A number of fundamental properties of interval arithmetic were discussed. Finally, we discussed the important concept of inclusion functions in some detail. This discussion is now summarized below.

There are two main kinds of inclusion functions which can easily be constructed:

(1) Natural interval extensions,

(2) Centered forms:

  (a) Meanvalue forms,

  (b) Taylor forms (of second order).

Natural interval extensions may be used in general even if f is not differentiable. Their use is recommended if the domain Y is "larger", that is, if $w(Y) \geq 1/2$ or $w(Y) \geq 1/(2m)$ where m is the number of variables of f.

Meanvalue forms may be used if f is differentiable, if f' has an inclusion function F' which is Lipschitz, and if $w(Y) \leq \frac{1}{2}$.

Meanvalue forms involving generalized gradients may be used if it cannot be decided from the outset whether f is differentiable or only generalized differentiable, and if $w(Y) \leq \frac{1}{2}$. Such an indeterminate situation occurs, for example, if $f(x) = \max(f_1(x), f_2(x))$ with $f_1, f_2 \in C^1$. Then the differentiability properties of f at x cannot be determined before $f_1(x)$ and $f_2(x)$ are evaluated.

This undecided situation does not cause any disturbance when programming the code. One just sets F' as an inclusion function for the generalized gradient. If f is finally (locally) differentiable in Y, then the generalized gradient shrinks to the gradient and F' is an inclusion of the gradient over Y, see Ratschek [211].

Taylor forms may only be used if a direct computation of the meanvalue form is not possible or if the Hessian inclusion $F''(Y)$ is already available and can be incorporated without difficulties. f has to be twice differentiable, f'' must have a bounded inclusion function F'' and $w(Y)$ should not be larger than $1/m$. The main advantage of a Taylor form is that the boundedness is the only side condition and that it is easy to prove in contrast to the Lipschitz condition required for the meanvalue form.

The above inclusion functions might all be used in the subdivision algorithms in Sec. 2.12.

Chapter 3

Interval Newton Methods

3.1 Introduction

Many geometric computations involve non-linear equations. These equations arise most often when defining non-linear surfaces. Intersecting such surfaces in computer aided design (see [78, 108]) or raytracing surfaces in computer graphics (see [266]) are a couple of examples where the manipulation of the surfaces result in the computation of the zeroes of the non-linear functions.

Computing the zeroes of non-linear functions (in one or more variables) is an active area in numerical analysis and several monographs have been written surveying the state of the art (see for example [191] for an excellent survey up to the date of publication).

In the interval setting some effective methods based on a generalization of Newton's method have been developed. These methods, known globally as interval Newton methods, are now discussed here focusing on a particular practical realization.

The interval Newton method was introduced by Moore [165]. It is an excellent method for determining all zeroes of a continuously differentiable function $\phi : X \to R^m$ where $X \in I^m$. The interval Newton method combines global reliability with the excellent local behavior commonly known from non-interval Newton methods. Refinements and further discussions of the method are due to Krawczyk [138], Nickel [181], Hansen[89], Hansen-Sengupta [96], Hansen--Greenberg [94], Alefeld-Herzberger [6], Krawczyk [141], Neumaier [179], Kearfott [127] and many others.

Before the method is specified we have to define what is meant by solving a system of linear interval equations since these occur as an integral part of the interval Newton method. An unfortunate notation is widely used to describe such equations since it uses the notation of interval arithmetic in a doubtful manner. This can lead to misunderstandings. I.e., let $A \in I^{m \times m}$, $B \in I^m$ then

the *solution of the linear interval equation (with respect to x or X)*

$$Ax = B \text{ or } AX = B$$

is not an interval vector X_0 that satisfies the equation, $AX_0 = B$, as one would expect. The solution is defined as the set

$$X = \{x \in R^m : ax = b \text{ for some } a \in A, b \in B\}.$$

The historical reason for this definition, which may appear rather cumbersome, is that A and B are thought of as a matrix resp. a vector which are enhanced by small errors or equipped with data perturbations. Hence any point matrix of A could be the true matrix of the system and any point vector of B could be the true right hand side of the system. An interval solution that keeps the philosophy of always containing the true (but unknown) solution therefore has to include all those combinations that occur in the definition.

Similarly, if $c \in R^m$, then the solution of the linear interval equation

$$A(x - c) = B \text{ or } A(X - c) = B$$

with respect to x or X is defined to be the set

$$X := c + Y := \{c + y : y \in Y\}$$

where Y is the solution of the interval equation $Ay = B$.

For example, the solution of the linear interval equation

$$[1,\ 2]x = [1,\ 2] \tag{3.1}$$

is $X = [1/2,\ 2]$ (which in this case can be found by setting $X = [1,\ 2]/[1,\ 2]$ according to the definition of interval division). If we multiply, for comparison, $[1,\ 2]$ and X we get

$$[1,\ 2]X = [1,\ 2][1/2,\ 2] = [1/2,\ 4]$$

which is an inclusion for the righthand side of (3.1).

In higher dimensions matters only get worse which can easily be seen by considering the system

$$\begin{pmatrix} [2,\ 4] & [-1,\ 1] \\ [-1,\ 1] & [2,\ 4] \end{pmatrix} X = \begin{pmatrix} [-3,\ 3] \\ [0,\ 0] \end{pmatrix}. \tag{3.2}$$

In order to describe this set the tools of interval analysis have to be augmented by the tools and techniques of linear programming.

The solution set of (3.2), obtained in [179], is shown in Fig. 3.1. The interest, from the point of view of interval analysis, is to find the smallest interval vector that contains the solution set. Applying Cramer's rule formally

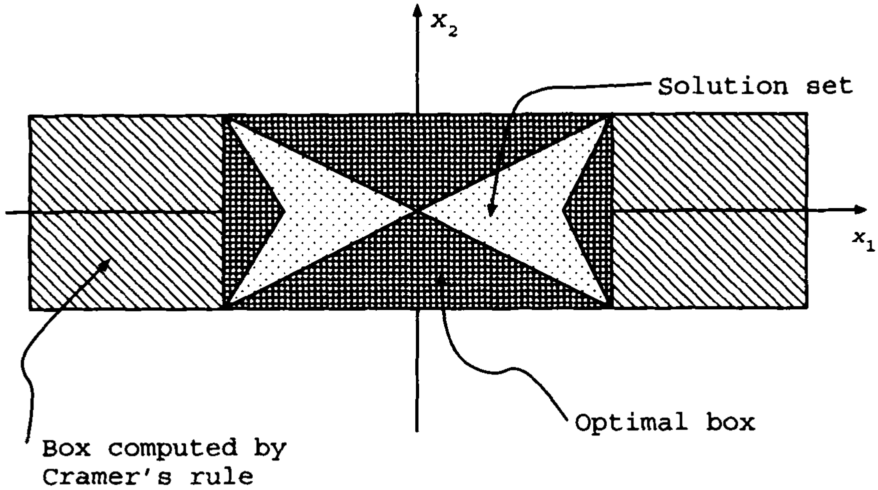


Figure 3.1: Solution set and inclusions

to (3.2), that is, we adapt Cramer's rule for solving systems of linear equations and admit interval entries in the vectors instead of real entries, results in an inclusion $X_C = ([-4,\ 4],[-1,\ 1])^T$. Further, the smallest box including the solution is $X_S = ([-2,\ 2],[-1,\ 1])^T$ (see for example [179] for how to compute X_S for this example). The inclusion X_C computed by Cramer's rule and the smallest box X_S including the solution set are also shown in Fig. 3.1.

We also note that

$$AX_C = ([-17.0,\ 17.0],[-8.0,\ 8.0])^T \text{ and } AX_S = ([-9.0,\ 9.0],[-6.0,\ 6.0])^T.$$

Clearly, $B \subseteq AX_S \subset AX_C$, however, even in the case of AX_S we have $w(AX_S - B) = 12$ which shows that AX_S is quite far from B.

The above examples show that the solution set is not a box in general and that even the smallest box including the solution set can be quite far from the solution set. It is therefore the aim of interval arithmetic solution methods to find a box which contains the solution set and which is as small as possible.

Below we will clarify why the solution set of a system of linear equations need not be an interval vector. Although we can not deal with the whole theory of the computation of solutions to linear interval equations in this monograph, since it would lead us too far from our aims, we can provide a reasonable discussion showing why the solution set can only be expected to be included when computing with interval tools.

The main reason is that simple matrix operations already lead out of the interval domain even if they would be executed optimally as is the case with the interval arithmetic operations. Let us go back to the interval product. Let

$A, B \in I$, then

$$AB = \{ab : a \in A, b \in B\}. \tag{3.3}$$

The right hand side can be interpreted as the range of the function $f(x, y) = xy$ over $A \times B$. Since each variable occurs only once, we can apply Theorem 3 saying that the range is equal to the natural interval extension.

The situation changes if we consider matrix products. Let $A = (A_{ij})$ and $B = (B_{ij})$ be two $m \times m$ matrices with interval entries A_{ij} and B_{ij}. The product was defined in formal analogy to the product of matrices over R, that is

$$AB = (C_{ik})_{i,k=1,\ldots,m}$$

where

$$C_{ik} = \sum_{j=1}^{m} A_{ij} B_{jk}.$$

This definition guarantees that the fundamental principle of interval arithmetic is valid for matrix products.

The definition can, however, no longer be interpreted as as the range of the underlying real matrix products as was the case with the simple product of two intervals, such as

$$\{\tilde{A}\tilde{B} : \tilde{A} \in A, \tilde{B} \in B\}, \tag{3.4}$$

cf. the right hand side of (3.3). The set (3.4) is the range of the function

$$f(x, y) = xy, \quad x, y \in R^{m \times m}$$

for $x \in A$, $y \in B$. Skelboe's principle is no longer applicable, since, if $m \geq 2$, the variables, which are x_{ij}, y_{ij} occur several times if the product were to be written out as a matrix of arithmetic expressions. Simple examples show that it is not possible to avoid it:

In order to make the example as transparent as possible, we chose the product of a real 2×2 matrix and an interval vector,

$$A = \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} [-1, 1] \\ 0 \end{pmatrix}.$$

The regular product definition gives

$$AB = \begin{pmatrix} [-1, 1] \\ [-1, 1] \end{pmatrix}.$$

whereas a product definition via (3.4) gives

$$\left\{ \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} u \\ 0 \end{pmatrix} : u \in [-1, 1] \right\} = \left\{ \begin{pmatrix} u \\ -u \end{pmatrix} : u \in [-1, \;] \right\}$$

which is not an interval vector, but is sufficient to include all possible matrix-vector products for the example such that the fundamental principle of interval arithmetic would be valid for this set.

If one compares the two sets, one recognizes that the second set is just a diagonal of the first set AB which is a square.

Let us finally return to the solution set of a linear system of interval equations. Clearly each interval arithmetic method for solving such a system must be an interval extension of a method for solving non-interval systems of equations (since these are the special case of point matrices for which the interval method must remain valid). This implies that matrix operations are involved which will lead to the overestimates shown in the above example.

Further discussions on interval linear equations are found in Hansen [92, 93], Ning-Kearfott [186], Neumaier [180], Shary [245] and Rohn [231, 230].

3.2 The Interval Newton Method

In the following we develop the interval Newton algorithm for determining the zeros of $\phi : X \to R^m$ in $X \in I^m$. Whereas one step of the non-interval Newton iteration procedure consists of computing a zero of the system of equations linearized about the current iterate to find the new iterate, the interval Newton method computes an including interval system for the linearizations of the equations about a point over the current iteration interval (or an inclusion to these equations) then solves this system in an including manner for the next iterate. This means that the algorithm is not a direct generalization of the scalar Newton algorithm. It is therefore developed from first principles here.

Let $x, y \in X$ and $\phi = (\phi_1, \ldots, \phi_m)^T$ be expanded componentwise by the meanvalue formula at x,

$$\phi(y) = \phi(x) + J(\sigma)(y - x),$$

where

$$J(\sigma) = (\nabla\phi_1(\sigma_1), \ldots, \nabla\phi_m(\sigma_m))^T$$

for a matrix $\sigma = (\sigma_1, \ldots, \sigma_m), \sigma_i \in R^m$ and $\sigma_i \in x \vee y$. We define

$$J(Y), \quad Y \in I(X)$$

a bit outside our usual convention as the natural interval extension of J to Y^m, that is, each σ_i is replaced by $Y, i = 1, \ldots, m$. From the definition of $J(\sigma)$ we obtain

$$J(Y) = J_\phi(Y),$$

such that $J(Y)$ is nothing but a natural interval extension of the Jacobian matrix $J_\phi(x)$ to Y. Note that $J(\sigma)$ is *not* a Jacobian matrix. If $y = \xi$ is any zero of ϕ in X then

$$J(\sigma)(x - \xi) = \phi(x).$$

This equation leads to interval Newton methods, in the the same manner as we get non-interval Newton methods in the non-interval case, cf. the iteration statement given below.

The following prototype is nothing more than a very rough sketch of Newton's method for a first discussion. It will be refined in Sec. 3.3. The norm in Step 2 can be any norm and Φ is an inclusion function for ϕ.

ALGORITHM 5 *(The Interval Newton Algorithm - first sketch)*

Step 1. *Set* $X_0 := X$.

Step 2. *For* $n = 0, 1, 2, \ldots$

(i) *choose* $x_n \in X_n$,

(ii) *determine a superbox* Z_{n+1} *of the solution* Y_{n+1} *of the linear interval equation with respect to* Y

$$J(X_n)(x_n - Y) = \phi(x_n), \tag{3.5}$$

(iii) *set*

$$X_{n+1} := Z_{n+1} \cap X_n, \tag{3.6}$$

(iv) *if* $\|\Phi(X_{n+1})\| < \epsilon$ *(or use any other suitable criterion) go to 3.*

Step 3. *End.*

Interval Newton methods are distinguished by the particular choice of the superbox Z_{n+1}. For example, if Z_{n+1} is the box hull of Y_{n+1}, that is, the smallest box containing Y_{n+1}, then the method is called the interval Newton method (in the proper sense). If Z_{n+1} is obtained by using interval Gauss-Seidel steps combined with preconditioning as will be explained in the next section, then the method is named after Hansen-Sengupta [95]. Krawczyk's [138] method and Hansen-Greenberg's [94] methods are also widely used. Convergence properties exist under certain assumptions. The following general properties are useful for understanding the principle of application of the algorithm, see Moore [165, 167], Alefeld-Herzberger [6], Neumaier [179]:

Basic Properties of the Interval Newton Algorithm

1. If a zero, ξ, of ϕ exists in X then $\xi \in X_n$ for all n. This means that no zero is ever lost! This implies that:

2. If X_n is empty for some n then ϕ has no zeros in X.

3. If Z_{n+1} is obtained by Gauss-Seidel or Gauss elimination suppressing the intersection step (3.6) or its variations (3.10), (3.12) or (3.11) (with or without preconditioning) then

   (i) if $Z_{n+1} \subseteq X_n$ for some n then ϕ has a zero in X,

   (ii) $Z_{n+1} \subseteq \text{int } X_n$ for some n then ϕ has a unique zero in X (where int means topological interior).

4. Under certain conditions one obtains

$$w(X_{n+1}) \leq \alpha(w(X_n))^2$$

   for some constant $\alpha \geq 0$.

Point 3(i) of the basic properties gives rise to develop an

Existence test (first sketch). Execute only one sweep of the interval Newton algorithm, i. e. only the sweep $n = 0$, drop or leave the preconditioning and replace the steps 2(iii) and 2(iv) of the above sketched algorithm by the step

2.(iii') if $Z_1 \subseteq X_0$ then Z_1 contains a zero of ϕ X_0,

Although it is likely that the existence test is valid for almost all reasonable variants of interval Newton methods its validity has been proven for only a few special versions, for example, the Hansen-Sengupta version [96].

Remarks. There are a great variety of possible improvements and refinements to the interval Newton algorithms. We do not incorporate all of the details of these improvements since we want to keep the essentials of the theory transparent. Nevertheless, we mention a few of the possibilities so that the reader can get an idea of what directions they take.

(i) *Use of slopes.* The interval Newton algorithm remains valid if the Jacobian $J(x_n)$ in (3.5) is replaced by the slope matrix or an inclusion of the slope matrix. I.e., the interval in the i-th line and j-th row of such a slope matrix could be an inclusion of the set of slopes

$$\frac{\phi_i(x_j) - \phi_i(x_{nj})}{x_j - x_{nj}}, x_j \in X_{nj},$$

where x_j, x_{nj} and X_{nj} are the j-th coordinates of x, x_n and X_n, respectively.

Since the slope matrix is included in the Jacobian matrix, $J(x_n)$, the convergence of the interval Newton algorithm will be faster if the slope matrix is used. The related theory can be found in Neumaier [179], p. 202.

(ii) *Use of Lipschitz matrices.* If one has to deal with functions ϕ that are not necessarily smooth then it might still be possible to apply the interval Newton algorithms if so-called Lipschitz matrices can be found for ϕ. These Lipschitz matrices then replace replace $J(x_n)$ in (3.5). The reader is referred to Neumaier [179], p. 175. Other methods for non-smooth systems can be found in Kearfott [129].

(iii) *An interval Newton variant of Oliveira* [189]. When creating the Newton iteration step, Oliveira uses a second order Taylor development of ϕ. This results in a variant of (3.5), where $J(x_n)$ is replaced with expressions in $J(x_n)$, which is a point matrix, and interval extensions of the second derivative of ϕ over X_n. Improvements of the Newton algorithm performance could be expected if the dimension of ϕ is not too high and if the box X_n is sufficiently small.

Fomia [55] goes one step further. If a ϕ only consists of polynomials, he uses higher order Taylor expansions of ϕ. This has the effect that the higher derivatives of the polynomial vanish if the degree of the polynomials is lower than the order of the Taylor expansion, and that the non-vanishing polynomial derivatives need to be evaluated on point arguments only.

(iv) *Boundary based interval Newton variants* [249]. This variant does not use the midpoint x_n or any other point x_n of X_n as developing point of the Newton step. Instead facets of X_n are looked for that satisfy $0 \in \phi(F)$. The knowledge of such facets give information on how to chose the formula for the particular Newton step appropriately.

The next section discusses a very effective practical realization of the Newton algorithm which will in the end lead to a list of boxes whose union will contain all the solutions of the equations. Such a list occurs by splitting processes which are part of this realization.

As an example of why splitting may be beneficial consider the equation system (3.2) where it is clear that the list of boxes $([-2,\ -1],[-1,\ 1])^T$, $([-1,\ 1],[-0.5,\ 0.5])^T$, $([1,\ 2],[-1,\ 1])^T$ provides a better inclusion than the optimal single box $([-2,\ 2],[-1,\ 1])^T$.

3.3 The Hansen-Sengupta Version

The Hansen-Sengupta [95] version is a very promising variant where the linear system occurring in the Newton iteration step is solved by

(A) a preconditioning step,

(B) relaxation steps (Gauss-Seidel).

Since we discuss just one iteration of the Hansen-Sengupta variant in the sequel we suppress the indices n when writing down the formulas that occur in the n-th iteration. That is, we write

$$J(X)(x - Y) = \phi(x) \tag{3.7}$$

instead of

$$J(X_n)(x_n - Y) = \phi(x_n)$$

and, accordingly, we search for a superset Z of the solution set of (3.7), where $X, J(X), x$ and $\phi(x)$ are given. The solution set of (3.7) is also denoted by Y. If we want to refer to the former, original box X, we are more likely to avoid misunderstandings if we speak of the initial box X_0, reminding us that the box X_0 was defined as the original box X.

(A) The preconditioning step

It was argued in Hansen-Smith [97] that the system (3.7) was best solved by premultiplying it by an approximate inverse of mid $(J(X))$ where a non-interval floating point arithmetic is satisfactory both for the inversion and the pre-multiplication.

If the approximate inverse is B then we obtain

$$BJ(X)(x - Y) = B\phi(x)$$

or

$$M(x - Y) = b \tag{3.8}$$

where $M = BJ(X)$ and $b = B\phi(x)$. In this manner the system has been modified to a system that is almost diagonally dominant provided the widths of the Jacobian entries are not too large.

Such systems are also amenable to Gauss-Seidel type iterations because of the likely diagonal dominance. This will be discussed below.

It is obvious that the solution set of (3.8) contains the solution set of (3.7) such that no solution is lost in the above transformation. The only thing we have to do now is to solve the linear interval equation (3.8).

In cases where the inverse midpoint preconditioning as described above yields poor results one should try to determine optimum preconditioners by solving the easy linear programming problems given by Kearfott [125].

(B) The relaxation step (Interval Gauss-Seidel step)

We know that all zeros of ϕ lying in X (where this X abbreviates the former X_n), are contained in the solution set of (3.8). The relaxation step tries to shrink X. It can, however, happen that when X is made smaller it is split into two or more disjoint boxes, containing all solutions in question. In order to

avoid an exponential increase of the number of subboxes the further steps are applied to the hull of the subboxes in such cases. A splitting into two subboxes is only done when the current iteration of the corresponding Newton method is finished.

The relaxation procedure for linear interval equations was developed in Hansen-Sengupta [96]. It consists of the application of the well-known non-interval Gauss-Seidel iteration procedure (see for example Conte-de Boor [27]) in an interval context (see also the discussion of related methods in Alefeld--Herzberger [6]). *Single steps (iterations) of this relaxation procedure are here used to solve the preconditioned set of equations* (3.8) and it is expected that the procedure will be efficient since the coefficient matrix M will in most cases contain the identity matrix I due to its construction as $BJ(X)$, although this is not guaranteed since we only required that B should be an approximate inverse of mid $(J(X))$. It should also be noted that the matrix M is kept constant throughout one relaxation step (i.e. $M = BJ(X)$) whereas the vector X is updated.

In one relaxation step the equation $M(x - Y) = b$ is solved for the i-th component Y_i obtaining a superset of Y_i by using the known inclusions X_j for Y_j, $j \neq i$,

$$Z_i = x_i + M_{ii}^{-1}\Big(\sum_{\substack{j=1 \\ j\neq i}}^{m} M_{ij}(x_j - X_j) - b_i\Big) \tag{3.9}$$

where x_j, etc., denotes the j-th component of x, etc. This interval is immediately used to intersect and update the i-th component X_i,

$$X_i := X_i \cap Z_i \tag{3.10}$$

subsuming step 2.(iii) of the interval Newton algorithm.

This calculation is performed for all $i, 1 \leq i \leq m$, first for the indices where $0 \notin M_{ii}$ and then for the remaining indices where $0 \in M_{ii}$. This strategy results from the observation that the updating of (3.10) with components X_i where $0 \notin M_{ii}$ improves (makes smaller) all the components Z_j via formula (3.9). This does however not hold for components X_i with $0 \in M_{ii}$.

If the intersection (3.10) is empty for some i then it follows from the properties cited in the last section that there is no solution in the current box X. This intersection has to be suppressed if the interval Newton iterations are used for the existence verification of solutions.

When the intersection is not empty then the computation continues with the next component and the updated X_i's.

If $0 \in M_{ii}$ and if

$$0 \in \sum_{\substack{j=1 \\ j \neq i}}^{m} M_{ij}(x_j - X_j) - b_i$$

then we set $Z_i = (-\infty, \infty)$. In this case the intersection (3.10) does not result in a narrowing of X_i; hence no useful information is obtained.

If

$$0 \notin \sum_{\substack{j=1 \\ j \neq i}}^{m} M_{ij}(x_j - X_j) - b_i$$

and $0 \in M_{ii}$ then Z_i consists of two non-overlapping semi-infinite intervals separated by an open set (gap) according to the extended interval arithmetic division given in (2.7). The intersection $Z_i \cap X_i$ may now be empty or consist of one or two intervals. In the first two cases the computation proceeds as for the case $0 \notin M_{ii}$.

If the intersection results in two intervals then the box may be split normal to this coordinate direction. It might be impractical when the box is split with respect to several coordinate directions, thus resulting in a proliferation of subboxes as mentioned before. A splitting is therefore only done once during the iteration, i.e. vertical to the direction of the largest gap or two largest gaps at the end of the iteration. In practice one has to keep track of both a gap and the index of the coordinate where it occurs.

The gaps are also not used right away; they are saved until the other techniques have been employed to narrow the current box. Also the gap and its index may be deleted if it is outside the current box.

As a simple numerical example of the Gauss-Seidel process we subject (3.2) to one Gauss-Seidel step starting with the vector $X = ([-4,\ 4], [-2,\ 2])$ and $x = ([0.0], [0.0])$. Since $0 \notin M_{ii}$, $i = 1, 2$ equation (3.9) is applied directly. After the intersection step (3.10) we obtain the new $X_i = ([-2.5,\ 2.5], [-1.25,\ 1.25])$ which is a substantial improvement.

Let X be a box, ϕ, B, and M as defined above. The following iteration aims to shrink X but not to loose any solution of the equation system $M(x - Y) = b$.

ALGORITHM 6 *(One single relaxation step (Interval Gauss-Seidel step))*

Step 1. *For* $i = 1, 2, \ldots, m$

if $0 \notin M_{ii}$ *then*

(a) set

$$Z_i := x_i + M_{ii}^{-1}\Big(\sum_{\substack{j=1\\ j\neq i}}^{m} M_{ij}(x_j - X_j) - b_i\Big)$$

and

$$X_i := X_i \cap Z_i. \tag{3.11}$$

(b) If $X_i = \emptyset$ then terminate and report

- *no solution in X (input box).*

Step 2. *For $i = 1, 2, \ldots, m$*

if $0 \in M_{ii}$ then

(a) set

$$Z_i := x_i + M_{ii}^{-1}\Big(\sum_{\substack{j=1\\ j\neq i}}^{m} M_{ij}(x_j - X_j) - b_i\Big)$$

and

$$X_i := X_i \cap Z_i. \tag{3.12}$$

(b) If $X_i = \emptyset$ then terminate and report

- *no solution in X (input box).*

(c) If X_i has a gap then replace X_i by its hull while keeping track of i and gap.

A further numerical example is given by

$$\begin{pmatrix} [1,\ 2] & [1,\ 2] \\ [1,\ 2] & [-1,\ 2] \end{pmatrix}(x - Y) = \begin{pmatrix} [1.0,\ 1.5] \\ [2.9,\ 3.0] \end{pmatrix}. \tag{3.13}$$

We start the Gauss-Seidel process with the vector $X = ([1,\ 2], [1,\ 2])^T$ and $x = m(X)$. Since $0 \notin M_{11} = [1,\ 2]$ we perform the first step of the procedure and obtain

$$\begin{aligned} Z_1 &= x_1 + (M_{12}(x_2 - X_2) - b_1)/M_{11} \\ &= 1.5 + ([1,\ 2](1.5 - [1,\ 1.5])/[1,\ 2] = [-1,\ 1.5]. \end{aligned}$$

Performing the intersection step we get $X_1 := [1, 1.5]$, a halving of the first coordinate interval. Now, $0 \in M_{22}$ so we compute

$$\begin{aligned} Z_2 &= x_2 + [M_{21}(x_1 - X_1) - b_2)/M_{22} \\ &= 1.5 + ([1,\ 2](1.25 - [1,\ 1.5]) - [2.9,\ 3.0])/[-1,\ 2] =]0.3, 3.9[\end{aligned}$$

where the notation $]a, b[$ indicates that the result is to the left of and including a and to the right of and including b i.e. there is a gap (a, b). The intersection of this result with $X_2 = [1,\ 2]$ now shows that $X_2 := \emptyset$ and hence $X := \emptyset$ which shows that (3.13) has no solutions in X.

If X has been shrunk significantly then it makes sense to repeat the step with the smaller X using the values of M and b from the last iteration (*simplified* relaxation step). This can be repeated several times until the shrinking is no longer significant. (We say that the shrinking is significant if the ratio of new box volume to old box volume is smaller than 0.9. This threshold results from computational experience and should be adjusted upwards with increasing m and increasing expense in recalculating $J(X)$.) After the termination of the relaxation step and the dependent simplified steps the largest one or two gaps are collected and they will be used to split the output box in two or four parts for further processing.

Remark 1. In order to determine the hull of that part of the solution set of the system $M(x - Y) = b$ which lies in X, the complete Gauss-Seidel algorithm (consisting of the Gauss-Seidel steps where after each step the values M, x, and b are calculated anew from the current X) can be applied to X. In order to get convergence to the hull one has to split the current box after each iteration when no shrinking or only insignificant shrinking could be obtained. Then the algorithm has to be applied to each of the boxes separately. Finally, all the solutions to the different branches have to be brought together to make up the whole solution set. The number of boxes can increase quite rapidly as can be seen from Figure 3.1 where sloped lines bounding the solution set have to be covered by axis-parallel rectangles. The number of boxes is therefore proportional to the desired accuracy in the solution. For convergence conditions see Neumaier [179]. The complete algorithm is not described here in further detail since we only use the algorithm as part of the interval Newton process to be described below.

Remark 2. Instead of a relaxation iteration Gauss elimination can be used. This is nothing more than the well-known Gaussian elimination performed in an interval setting. Gaussian elimination is not as robust as the Gauss-Seidel steps. It is, however, more effective under certain conditions (for instance, if the Jacobian or the preconditioned Jacobian matrix is diagonally dominant, see Neumaier [179]). Practical experiences show that it is best to combine Gauss-Seidel steps with Gaussian elimination, cf. Hansen [92], Neumaier [179], Ratschek-Rokne [213].

Remark 3. Further means to improve the efficiency of the relaxation steps is not to take $x =$ mid (X), but points with function values near to zero as developing point. For this one applies a non-interval Newton or quasi-Newton method to mid (X) and stops at a point $x \in X$ with small norm $\|\phi(x)\|$ or at the boundary of X if the trajectory followed by the non-interval Newton

iterations leave X. Details may be found in Hansen [91], Ratschek-Rokne [213].

We now describe the interval Newton algorithm in more detail.

Inclusion functions $J(Y)$ and $\Phi(Y)$ for $J_\phi(x)$ and $\phi(x)$ are needed as well as an $\epsilon > 0$ or $\tilde{\epsilon} > 0$ for a terminating criterion.

ALGORITHM 7 *(Interval Newton Algorithm after Hansen-Sengupta)*

Step 1. *Let X be given.*

Step 2. *Initialize list* $\mathrm{L} = (X)$.

Step 3. *Calculate $\Phi(X)$. If $0 \notin \Phi(X)$ then go to 15.*

Step 4. *Calculate $J(X)$.*

Step 5. *Calculate B, an approximate inverse of* mid $(J(X))$.

Step 6. *Set x equal to the midpoint of X, that is $x =$* mid (X).

Step 7. *Calculate $M := BJ(X)$, and $b := B\phi(x)$.*

Step 8. *Apply one Gauss-Seidel step to $M(x - Y) = b$ to obtain Y. Set $X := Y$. (Keep track of gaps and their indices.) If some $X_i = \emptyset$ then go to 15.*

Step 9. *If X improved significantly in Step 8 set $x :=$ mid (X) and perform the simplified relaxation procedure. If $X = \emptyset$ then go to 15.*

Step 10. *If X improved significantly in Step 9 set $x :=$ mid (X) and return to 9.*

Step 11. *If X improved significantly in Step 8 and if $\|\Phi(X)\| \geq \epsilon$ go to 3.*

Step 12. *If the gaps together with their coordinate directions were saved in Step 8 then*

- *update the gaps (they could increase or even vanish by the continued shrinking process(3.10)). Use the coordinate direction with the largest gap for splitting X using the gap obtaining boxes V_1 and V_2. This gap is no longer part of V_1 or V_2. (The remaining gaps are still included.) Go to 14.*

Step 13. *If X did not improve significantly in Step 8 then*

- *choose a coordinate direction v parallel to which $Y_1 \times \ldots \times Y_m$ has an edge of maximum length, i.e. $v \in \{i : w(Y) = w(Y_i)\}$. Bisect Y vertical to direction v getting boxes V_1, V_2 such that $Y = V_1 \cup V_2$.*

Step 14. *Enter V_1 and V_2 onto the list.*

Step 15. *Remove X from the list. If the list is empty, terminate and report*

- *no solutions.*

Step 16. *Apply termination criteria. For example: (i) If* $\| \Phi(X) \| < \epsilon$ *for* all *boxes X of the list then terminate. Or: (ii) If* $w(X) < \tilde{\epsilon}$ *for* all *boxes X of the list then terminate since a continuation seems to be unsuccessful. Etc.*

Step 17. *Set X to be the box of the list with the largest width.*

Step 18. *Return to 3.*

3.4 The Existence Test

After the termination of the computation of Alg. 7, for example by criterion (i) of Step 16 the list L contains only boxes whose points have absolute function values smaller than ϵ. Further, all the zeros of $\phi(x)$ that lie in the starting box lie also in the final boxes since no zero is ever lost.

If one wants more specific information, one might apply the *existence test* (cf. Sec. 3.2) to the boxes Y of the final list. That is, apply *one single interval Gauss-Seidel step* to Y (with or without executing any preconditioning beforehand) but without executing the intersection operations (3.10), (3.12) and (3.11). (When dropping the preconditioning set $M := J(X)$, $b := \phi(x)$ in Step 7 of Alg. 7.) After its termination check for the inclusions $Z_i \subseteq Y_i$ for $i = 1, ..., m$. If they all are valid, Y contains a zero of ϕ.

Boxes, that fail the existence test, can still contain a zero, but a decision was not possible during the computation, that is, w. r. to the parameters and the accuracy.

What is the reason that one makes a rough separation between the existence test from the interval Newton method? The reason is that there are many older and newer versions of interval Newton methods. They have two aims: The first aim is to eliminate parts of the domain of a function that have no zeros, and the second aim is to prove that a zero does exist in a certain domain. In the first case the method is mainly used in computations where the shrinking of the domain is most important. This is especially important when the domain is globally processed where the shrinking of the domain implies that the removed parts need not be processed. This makes the computation faster and cheaper.

A typical example for this kind of procedure is the Hansen-Sengupta algorithm which was primarily designed for solving global optimization problems, cf. Hansen-Sengupta [96]. Their algorithm was applied to the derivative of the objective function with the intention that the parts removed during the shrinking process could not contain a local extremum (except on the boundary of X) and hence, cannot contain a global extremum. To apply the existence

test in such cases would be a wasting of time since it only would confirm zeros of the derivative which would be reasonable when one would search for local extrema, but does not help in most cases when one searches for a global extremum. This is also the reason that only a few sweeps of the algorithm were recommended to shrink the domain in order not to concentrate the remaining subboxes around local minimizers which are finally not needed. On the other hand if the local extrema which are not global are already eliminated by other means, for example, Moore's [165] famous *midpoint test*, it makes again sense to apply the existence test since it can confirm the global extremum since it also a local one.

To know about the existence of zeros is important especially for geometric computations: Whether two explicitly defined curves, say $y = f(x)$ and $y = g(x)$, intersect is equivalent to the question whether the function $y = g(x) - f(x)$ has zeros. The question has to have a definite answer which is embodied in the meaning of the existence tests. As we will see, it is first important to shrink the domain with the interval Newton method and then to confirm the existence of the zeros in small subareas.

One also can ask for the *uniqueness* of zeros which is an important issue in many mathematical and computational disciplines. We drop this interesting field since is not so relevant for this monograph. For good surveys of existence and uniqueness tests cf. Neumaier [179] and Kearfott [127].

Thus the idea of the *existence test* is first to apply the interval Newton algorithm with an appropriately chosen $\epsilon > 0$ (or $\tilde{\epsilon} > 0$ or both). After its termination by Step 16 of Alg. 7 a list of subboxes is rendered with widths that are sufficiently small (or smaller than $\tilde{\epsilon}$). These boxes are characterized by the fact that the computation up to the termination was not able to prove that these subboxes do not contain any zeros. This means that these boxes contain either points which are almost zeros or points which are, in fact zeros. Hence the existence test is an excellent tool for figuring out boxes which contain a zero.

The chances of the test to indicate a zero increase the smaller the parameter ϵ or $\tilde{\epsilon}$ is and hence the smaller the widths of the final boxes are. Theoretically one could already apply the test to the whole domain without any shrinking by the interval Newton method, but then a positive answer will hardly be obtained. The reason is that the smaller the box width is the smaller is the excess-width and the better is the approximation of the range of the box by the inclusion function and thus the better the numerical results are.

The following variant of the Hansen-Sengupta algorithm combines the interval Newton method, which provides an adequate shrinking of the domain, with the existence test for verifying that there are zeros in the domain as it is described in the listing of the *basic properties* in Sec. 3.2. In addition to the objective function $\phi(x)$ and its domain X, inclusion functions $J(Y)$ and $\Phi(Y)$ for $J_\phi(x)$ and $\phi(x)$ are needed as well as an $\epsilon > 0$ as upper bound for the

absolute function values in the boxes considered in the terminating criterion. If ϕ is very flat then one also should consider the termination via $\tilde{\epsilon}$ additionally or alternatively. As a first guess for $\tilde{\epsilon}$ we suggest numbers from $w(X)/100$ for smoother functions up to $w(X)/1000$ for functions with more variations. It might be necessary to adapt these default values to the actual conditions. We also suggest to drop the preconditioning in the existence test since the dimensions for the purposes of computational geometry are at most 3 so that the chances of the test to render a positive result will not be improved too much by the preconditioning.

ALGORITHM 8 *(Interval Newton Algorithm with Existence Test)*

Perform the interval Newton method, Alg. 7. When the computation is terminated because

(i) the processing list L *is empty then STOP (there is no zero of* $\phi(x)$ *in* X*),*

(ii) all boxes of the (nonempty) list L *are of width smaller than* ϵ*, then apply one sweep of Alg. 8 but without preconditioning and without executing the intersection operations such as (3.10), (3.12), and (3.11) for these boxes. If*

$$Z \subseteq Y$$

for some of these boxes Y *where* Z *is the box resulted from the application of the above-mentioned sweep to* Y *then STOP (*Z *contains a zero and hence* Y *and* X *do).*

(iii) If $Z \subseteq Y$ *holds for no box* Y *of the list* L*, then STOP (no decision results from this computation whether a zero exists in* X *or not).*

Chapter 4

The Exact Sign of Sum Algorithm (ESSA)

4.1 Introduction

In the introduction to the book we noted that many geometric algorithms are dependent on the sign of a finite sum. Examples of such algorithms are left-turn test, orientation questions, Boolean algorithms (point in circle vs. not in circle), etc. Implementing such algorithms in fixed length floating point arithmetic can lead to inaccurate or wrong geometric configurations due to falsification of the computation by rounding errors. Interval analysis techniques can filter some of the computations and provide guaranteed results in certain cases, however, some cases are left that have to be dealt with using exact techniques.

For such cases ESSA [221] was developed. It is an algorithms which determines the sign of a sum of real quantities in a guaranteed manner.

The algorithm is especially designed for computations involving geometry where rounding error free algorithms are particularly desirable due to the strong influence of rounding errors on logical decisions as mentioned earlier.

In order to meet the condition of being rounding-error-free, the algorithm is so constructed that it processes data that is already in a binary form. Conversion errors are therefore avoided and hence, as noted in 4.3, only machine numbers are considered and the algorithm will determine the sign exactly, that is the result is guaranteed to be correct.

An extensive literature exists on the computation of the sum of a set of floating-point numbers and in some cases on the relationship of this computation to the stability of numerical and geometric computations [45, 104, 200]. Most of this literature does not mention the restricted problem of the determination of the sign of such a sum.

A short abridge of the *contents* of this chapter is as follows. In the next section, we consider some of the literature of geometric computations that show the need of our algorithm concept. In Sec. 4.3, the algorithm is established. The most important properties of the algorithm are described in Sec. 4.4. Numerical results are shown in Sec. 4.5 where the algorithm was tested with both well- and ill-conditioned sums with 10000 randomly generated summands. In Sec. 4.6 we have a close look at a few of the direct applications of the algorithm. Finally, in Sec. 4.7, a program for the algorithm written in C++ code is included.

4.2 The Need for Exact Geometric Computations

In this section we reflect further on the relationship between exact computations and computations implemented on a computer, especially for the case of geometric applications, on the still missing balance and distinction between these two kinds of computations, and on the need for exact execution of some primitives occurring in geometric computations.

Simple computational primitives form the basis for many geometric algorithms. As an example the *Graham scan algorithm* [198] for computing the convex hull of a finite point-set in the plane relies on the well-known *left-turn test* which is a primitive that decides whether a query point lies to the left, right or on a directed line defined by two other different points. The three-valued result of this primitive can be transformed into the result of the computation of the sign of a 3×3 determinant, a seemingly simple computational task. When the formula for the determinant is implemented in finite precision arithmetic, however, roundoff errors may falsify the result so that primarily, a wrong sign of the determinant, and secondarily, a perturbed or a non-convex hull is computed. If this hull is used in further computations then configurational anomalies can occur. A rich variety of solutions have been proposed for the algorithmic part of the convex hull determination ranging from Graham scan (mentioned above) to gift wrapping and quickhull [198, 190]. Algorithms based on these solutions would all return convex hulls (i.e. a set of points forming the vertices of a convex polygon and the adjacency information for the vertices) if they operated over the *field of real numbers*. This is not possible in general since real number implementations of algorithms would require infinite representations of data, intermediate results and outputs. Realistic implementations must deal with finite representations such as the representations available on a computer. The problems that arise from implementing the algorithms on a computer using fixed-length floating point computations were, however, mostly only mentioned and then ignored. In trying to overcome such obstacles Forrest [56] suggested more generally *for all kinds of geometric computations*

> One of the steps in developing a geometric computing environment is to identify geometric primitives and then to implement these correctly. If primitives are indeed primitive, they ought to be simple and hence the identification of special cases and their correct treatment should be possible. Given correctly implemented primitives, there is a reasonable prospect of building an environment in which such primitives can be utilized in a proper manner to build more reliable geometric systems.

A number of authors have also attempted to provide correctly implemented geometric computations. One approach is the *epsilon geometry* discussed in the thesis by Salesin [237]. He provides a general model for imprecise geometric computations which include fixed and floating point arithmetic as special cases. Algorithms based on this model take numerical data as input and provide combinatorial data as output that provide an exact solution to a perturbed version of the input. The decisions based on the computation of geometric primitives are derived with *interval tools* such that the decisions are either *true*, *false* or *uncertain* and it is guaranteed that all undecidable decisions are of the latter type. The uncertain result of a computation is undesirable and part of the research in [237] is to construct procedures such that the range of inputs resulting in the uncertain case is limited. Another tool used to implement primitives in this model is *backward error analysis*.

Another approach is discussed in [123] where the primitive *sign of determinant* is discussed. A variety of other geometric primitives can be transformed to this primitive. *Interval arithmetic* is employed as an *interval filter* as far as possible to get a decision for the sign of the determinant. When this fails they resort to variable precision arithmetic. A determinant used as a primitive in the construction of a Delaunay triangulation illustrates the method.

A numerically stable convex hull algorithm for a simple polygon as proposed in [115] is based on *backward error analysis*. Here the primitive is *slope of line* and the decision as to whether a sample point is a point of the convex hull or not is made by comparison of slopes. In [116] the authors provide an algorithm that when implemented in fixed length floating point arithmetic constructs a *truly convex hull*, that is, the algorithm returns a point set that are the vertices of a convex polygon even in the presence of roundoff error, where the underlying concept is again *backward error analysis*.

Sugihara [259] assumes that the precision of the input data is a given number of bits, typically less than the mantissa length of a floating point number in the computational device used. He then shows that degeneracy (vertex degree > 3, i.e. more than 3 edges meeting at a vertex) can be avoided in a 2D Voronoi diagram computation implemented in quadruple precision if vertex perturbation is allowed. He also shows that quadruple precision does not lead to numerical error in the decisions derived from the numerical computations for a 2D Voronoi diagram.

Arbitrary precision arithmetic has been implemented by, for example, [103]. It is used by [70] to compute exact results for planar maps. It is easily shown, however, that the required precision increases rapidly and that the computations become very expensive. In order to reduce the cost of the computations [159] advocates *double-precision* arithmetic. The algorithm in [159] calculates an arrangement of lines that is topologically correct except that neighboring vertices might be rounded to the same point.

Many of the inconsistencies encountered in the implementation of geometric algorithms can be traced back to *degeneracies* or special cases. Examples of such degeneracies are three lines meeting at a point, one point lying on a line defined by two other points, parallel lines or four points lying on a circle. Edelsbrunner and Mücke [41] and Yap [274] therefore *perturb the data* slightly so that the degeneracies are avoided. This simplifies the logical structure and the design of algorithms considerably since it removes the degeneracies that are difficult to manage and which often lead to uncertain decisions. The disadvantage with this approach is that the degeneracies might be an integral part of the geometric model. The perturbations might therefore change essential features of the model, such as its topology, in an unwanted manner. An example of such a situation is easily found in solid modelling where many objects have faces defined by more than three points, yet faces defined by more than three points constitute degenerate cases.

Let us return to the computation of the convex hull of a finite set of points in the plane, mentioned earlier. This computation provides possibly the best example done up to the present for research into a numerical method for a particular geometric problem in the presence of numerical errors.

Although some authors realized that numerical error influenced the result of a convex hull computation, it was Li and Milenkovic [147] who made a first attempt to avoid the dependency on those errors. They designed an $O(n \log_2 n)$ convex hull algorithm which computed an *ϵ-strongly convex $o(\epsilon)$-hull*. This means that an approximate hull was constructed so that no point of the set was more than $o(\epsilon)$ away from the required hull and that every vertex could be moved by a distance ϵ without violating the convexity property. Certainly, the hull property for the original set was lost. The algorithm uses the notions of *spines*, *vertebrae* and *extenders* which are quite sophisticated means developed to exclude instability and wrong evaluation of the geometric situation caused by the rounding errors which arise from the left-turn test. The algorithm is an *ϵ-geometry* type algorithm.

In [59] a similar result is presented using the so-called *ϵ-arithmetic*. This arithmetic is based on elementary axioms for floating-point arithmetic. Although the algorithms are numerically stable they also only compute an approximate hull.

In this context we again mention the construction of a truly convex hull, that is based on *slope comparisons* and *backward error analysis*, cf. [116].

In [122] the approach is again different. The authors use a blend of algorithms, some based on *interval analysis*, to compute different parts of the convex hull, and they switch to exact computations with *variable precision arithmetic* when interval arithmetic fails. The result is an *exact hull*, that is, it is truly convex in the above sense, and it is exact, that is, it is the hull of the original and not the perturbed set. The computational cost is, however, quite high.

Finally we would like to mention the idea that in order to determine the sign of the value of an expression such as a determinant it might be possible to determine the value of only a part expression. This idea seems to have originated in the paper [123] where it is stated that:

> In particular, many such primitive tests, including orientation of $d+1$ points in E^d and point-hyperplane classification, can be formulated as the sign of the determinant of a matrix. Since computing values of determinants is very expensive in arbitrary- precisions arithmetic, it is natural to ask whether is is possible to compute the sign *without computing the value.*

Our algorithms strictly confirms and emphasizes this fine idea.

4.3 The Algorithm

In order to be precise we give the algorithm to be described a name and we call it *ESSA* which is an abbreviation for the ***E****xact* ***S****ign of* ***S****um* ***A****lgorithm*.

We now present the details of ESSA which is of extremely simple logical and algorithmical structure. ESSA determines the *sign* of the sum of a finite set of binary floating point numbers of a fixed mantissa length, t. Although ESSA could be made valid for any finite sum we design ESSA for sums with a number of summands not exceeding 2^{t-1}, which is large enough for almost all applications. If we were to drop this restriction then ESSA and its discussion would loose their transparency and become too sophisticated. Certainly, variable precision arithmetic could also be used to compute the exact sign of a sum, however, the complexity of variable precision arithmetic increases much faster then the complexity of ESSA described here.

At this point we note that the algorithm resembles algorithms for the computation of the value of a sum based on the *distillation principle* (see for example [200]). The difference is the requirement that the sign is computed with no error and that the algorithm terminates without having to consider all terms in most of the cases.

Even though no mantissa manipulation is required for writing the code, it is necessary to introduce the kind of representation we are working with. We assume that the machine numbers are binary floating point normalized numbers having single precision, that is, they have some fixed mantissa length

$t > 0$. Hence, besides 0, $a \neq 0$ is a *machine number* in our context iff a is of the form

$$a = 0.\alpha_1 \dots \alpha_t \times 2^E \tag{4.1}$$

where $\alpha_1 = 1$ (normalization), $\alpha_2, \dots, \alpha_t \in \{0, 1\}$, $\alpha_1 \dots \alpha_t$ being the *mantissa* of a and E the *exponent part* (shortly, *exponent*) of a to the base 2. (There is no need for a binary representation of E in this paper. We also do not consider overflow, underflow or any restrictions of the size of E, which is the user's responsibility.) Depending on (4.1), a has a value

$$a = \sum_{i=1}^{t} \alpha_i 2^{-i} \times 2^E = \sum_{i=1}^{t} \alpha_i 2^{E-i}.$$

One should be cautious if a is already a power of 2, for example, $a = 2^8$. Here, the exponent of a is not 8, but 9 due to the representation (4.1).

Preprocessing. Let a collection of $l \geq 0$ machine numbers, i.e. normalized binary floating point numbers, $s_i \neq 0, i = 1, \dots, l$, be given as summands. Note that zero as summand is excluded.

As already mentioned we pose the side condition

$$l \leq 2^{t-1}, \tag{4.2}$$

which can be omitted if necessary. In that case ESSA will be less transparent and its computational cost will increase, see Remark 5 in Sec. 4.4.

We sort the summands into positive and negative summands. Also, we work with the absolute values of the negative summands instead of their proper values. Both of the classes of summands that we work with are ordered by "$\geq$", that is, the larger-than-or-equal relation.

Input for ESSA:

1. The ordered list of positive summands, $a_1 \geq a_2 \geq \dots \geq a_m$ (> 0), where $m \geq 0$
2. the ordered list of the absolute values of the negative summands, $b_1 \geq b_2 \geq \dots \geq b_n$ (> 0), where $n \geq 0$.

Hence, the sum, the sign of which we need, is

$$S = \sum_{i=1}^{l} s_i = \sum_{i=1}^{m} a_i - \sum_{i=1}^{n} b_i. \tag{4.3}$$

Clearly, $l = m + n$. By E_i we denote the exponential part of a_i, by F_j the one of b_j ($i = 1, \dots, m$; $j = 1, \dots, n$).

ALGORITHM 9 *(Exact sign of sum algorithm* **ESSA**.*)*

Step 1. (Termination Criteria)

(i) *If* $m = n = 0$ *then* $S = 0$. **Stop**.
(ii) *If* $m > n = 0$ *then* $S > 0$. **Stop**.
(iii) *If* $n > m = 0$ *then* $S < 0$. **Stop**.
(iv) *If* $a_1 \geq n2^{F_1}$ *then* $S > 0$. **Stop**.
(v) *If* $b_1 \geq m2^{E_1}$ *then* $S < 0$. **Stop**.

Step 2. (Auxiliary variables).
Set $a_1^{'} = a_1^{''} = b_1^{'} = b_1^{''} := 0$.

Step 3. (Comparison and processing of the leading summands of the lists.)

(i) *Case* $E_1 = F_1$:
If $a_1 \geq b_1$ *then set* $a_1^{'} := a_1 - b_1$
else set $b_1^{'} := b_1 - a_1$.

(ii) *Case* $E_1 > F_1$:
Set $u := 2^{F_1 - 1}$ *if* $b_1 = 2^{F_1 - 1}$ *otherwise set* $u := 2^{F_1}$.
Set $a_1^{'} := a_1 - u$, $a_1^{''} := u - b_1$.

(iii) *Case* $F_1 > E_1$:
Set $v := 2^{E_1 - 1}$ *if* $a_1 = 2^{E_1 - 1}$ *otherwise set* $v := 2^{E_1}$.
Set $b_1^{'} := b_1 - v$, $b_1^{''} := v - a_1$.

Step 4. (Rearrangement of the lists while keeping S constant).

(i) *Discard* a_1 *and* b_1 *from lists.*

(ii) *Enter those of the values* $a_1^{'}, a_1^{''}$ *and* $b_1^{'}, b_1^{''}$ *that are not zero to the* a_i *- list resp.* b_i *- list such that the lists remain sorted.*

(iii) *Rename lists (as well as list lengths) as*
$a_1 \geq a_2 \geq \ldots \geq a_m (> 0)$, $b_1 \geq b_2 \geq \ldots \geq b_n (> 0)$.

(iv) *Goto Step 1.*

Dissection of the steps

Step 1. The termination criteria are obvious. Parts (iv) and (v) express just a dominance of the positive or negative summands of S, for instance, case (iv),

$$S = \sum_{i=1}^{m} a_i - \sum_{i=1}^{n} b_i \geq a_1 - nb_1 \geq n(2^{F_1} - b_1) > 0,$$

since F_1 is the exponent of b_1.

Here, $m > 0$ and $n > 0$ were already used; otherwise the computation would already have been stopped by (i), (ii) or (iii).

Note that the products $n2^{F_1}$ and $m2^{E_1}$ that occur in the computation are of type integer and thus exactly executable.

The conditions in (iv), (v) are not only termination criteria, but are responsible for the effectiveness of ESSA. Together with the assumption $l \leq 2^{t-1}$, they cause

$$|E_1 - F_1| \leq t - 1 \tag{4.4}$$

for the current values of a_1 and b_1, cf. the Lemma. This means that the mantissas of a_1 and b_1 overlap, when the larger of the two numbers is represented normalized, and the other is potentially represented unnormalized, so that $\max(E_1, F_1)$ is its exponent part.

Step 3. In case (i), it is obvious that the partial sum $a_1 - b_1$ resp. $b_1 - a_1$ can be executed exactly. Note that the exponent of $a_1 - b_1$ is less than $E_1 = F_1$.

Case (ii). Generally, $a_1 - b_1$ cannot be computed exactly. However, $a_1 - b_1$ can be approximated by a_1' with a remainder a_1'' because of the equation

$$a_1 - b_1 = a_1 - u + u - b_1 = a_1' + a_1''. \tag{4.5}$$

Both values, a_1' as well as a_1'', can be computed exactly because of (4.4).

For example, if $b_1 \neq 2^{F_1 - 1}$ then $u = 2^{F_1}$. Due to (4.4), u can be represented as

$$u = 2^{F_1} = 0.\delta_1 \ldots \delta_t \times 2^{E_1}$$

where

$$\begin{aligned} \delta_i &= 1 \text{ if } E_1 - F_1 = i \\ &= 0 \text{ otherwise.} \end{aligned}$$

Hence, subtracting u from $a_1 = 0.1\alpha_2 \ldots \alpha_t \times 2^{E_1}$ can be done exactly in our setting.

The second difference, $a_1'' = u - b_1$, is best demonstrated by writing it out in the following manner,

$$\begin{aligned} u &= 1.00\ldots0 \times 2^{F_1} \\ -b_1 &= -0.1\beta_2 \ldots \beta_t \times 2^{F_1} \end{aligned}$$

gives

$$a_1'' = 0.0\beta_2' \ldots \beta_t' \times 2^{F_1} = 0.\beta_2' \ldots \beta_t' \times 2^{F_1 - 1}.$$

Here, β_i' denotes the dual complement of β_i.

Note that

$$a_1' \leq a_1 - b_1 < a_1$$

and that the exponent of a_1'' is less than F_1.

Case (iii) is analogous to Case (ii).

Step 4. This step replaces the leading summands of the lists, a_1 and b_1, by

$$\begin{array}{ll} |a_1 - b_1| & \text{in case (i),} \\ a_1', a_1'' & \text{in case (ii),} \\ b_1', b_1'' & \text{in case (iii).} \end{array}$$

The new elements are put on the lists in the proper order.

In case (i), one of the lists is shortened by 1 element. In case (ii), the a_i-list gets longer by 1, the b_i-list shorter by 1. This part of the step aims to shrink the b_i-list as much as possible in order to extend the dominance of the a_i-list (i.e., $E_1 > F_1$) and to confirm an expected positivity of S. (The results of numerical experiments with random numbers coincide with this hypothesis, cf. Sec. 4.5, Table 1.) The sum S remains constant because of

$$a_1 - b_1 = a_1' + a_1''.$$

Case (iii) is analogous to Case (ii).

4.4 Properties of ESSA

In this section, we collect the basic properties of ESSA that enable us to work with it. Some of the properties required are given as remarks since their proofs are quite simple.

Since formula (4.4) is essential for ESSA to work it will be proven. The assumptions are that $a_1 \leq n2^{F_1}$, $b_1 \leq m2^{E_1}$, $m \neq 0, n \neq 0$ (otherwise the computation had already been terminated by Step 1).

LEMMA 1 *If $l \leq 2^{t-1}$ then $|E_1 - F_1| \leq t - 1$ for the current values in Step 3 of ESSA.*

Proof by contradiction. Without restricting the generality, we assume

$$a_1 > b_1$$

which implies $E_1 \geq F_1$. In order to get a contradiction, we further assume

$$E_1 - F_1 \geq t. \tag{4.6}$$

We want to show that $n2^{F_1} < a_1$ which gives a contradiction.

Let

$$a_1 = 0.1\alpha_2 \cdots \alpha_t \times 2^{E_1}.$$

Then

$$2^{E_1 - 1} \leq a_1.$$

Since $l = m + n$ and $m \neq 0$, we have $n < l$. Hence, by (4.6) it follows that

$$n2^{F_1} < l2^{F_1} \leq 2^{t-1}2^{F_1} \leq 2^{E_1-1} \leq a_1,$$

which provides the intended contradiction. □

THEOREM 8 *If the input data of ESSA are machine numbers, the computation is exact and delivers the sign of the sum $S = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j$.*

Proof. The computation is exact since no rounding errors arise during the execution of the arithmetic operations in Step 3 as already explained in the last section. Further, the iterations change the arrangement of the two lists, but the value of the sum (as difference $\Sigma a_i - \Sigma b_j$ with the current values a_i, b_j) is kept constant, cf. the dissection of Step 3 in Sec. 4.3. The computation terminates if at least one of the lists is empty or one of the two lists dominates the other, cf. the dissection of Step 1 in Sec. 4.3. That ESSA does terminate will be made clear in Rem. 1. □

Remarks

1. *ESSA terminates* after a finite number of iterations. Clear:

   (i) In each iteration, the leading element of one of the two lists is replaced by a number with strictly smaller exponent or by 0, the leading element of the other list is replaced by a strictly smaller element or by zero.

   (ii) If $E_{\min} = \min(E_m, F_n)$ is the smallest occurring exponent in the originally given sequences, that is, in the iteration 0 at the input level, then no non-zero element can arise during the whole computation with an exponent smaller than $E_{\min} - t + 1$. The reason for this is that, if two numbers with the same exponent are added or subtracted, the exponents never decrease if the results are represented unnormalized. Hence, if the final results are then normalized, the result is 0 or a number whose exponent is not less than $E_{\min} - t + 1$.

   (iii) Since one of the two leading exponents is reduced by 1 in each iteration, cf. (i), and since no exponent can ever be less than $E_{\min} - t + 1$, cf. (ii), ESSA must terminate.

2. *Upper bound for the number of iterations.* The maximum number of iterations does not exceed $l^2 t$. This number results from the number of exponent reductions by 1 of all the summands (Step 3, (ii) or (iii)) until they reach the lowest possible exponent, $E_{\min} - t + 1$, such that finally a number of Step 3 (i) operations will be executed, which is to be added. This is a worst case arrangement.

   In order to prove this assertion one has to go into the details:

   A. First we design a worst case model. For this purpose we arrange the set $\{a_i : i = 1, \ldots, m\} \cup \{b_i : i = 1, \ldots, n\}$ as a sequence $(c_\nu)_{\nu=1}^{l}$ ordered by the $\leq$-relation. Let G_ν be the exponent of c_ν. Then we assume

$$G_1 - G_2 \leq t - 1, \tag{4.7}$$

$$G_{\nu-1} - G_\nu \leq 2t - 2 \text{ for } \nu = 3, \ldots, l. \tag{4.8}$$

These two assumptions make the discussion of the worst case simpler and they do not restrict the worst case: If (4.7) does not hold then one has $G_1 - G_2 > t$ and further, $|E_1 - F_1| > t$. This implies from Lemma 1 that one of the termination conditions hold such that no worst case situation is present.

If (4.8) does not hold then we have

$$G_{\mu-1} - G_\mu > 2t - 2 \tag{4.9}$$

for some $\mu \geq 3$. Correspondingly, we subdivide the sequence $(c_\nu)_{\nu=1}^{l}$ into two subsequences, a *left* subsequence, i.e. $(c_\nu)_{\nu=1}^{\mu-1}$ and a *right* subsequence, i.e. $(c_\nu)_{\nu=\mu}^{l}$. Since ESSA works from left to right, there will be some stage of the computation where the right subsequence is not yet involved, but where one of the following cases occurs:

a) the left subsequence consists only of numbers from the a_i-list or only of numbers from the b_i-list,

b) the left subsequence consists only of two numbers and $c_1 = c_2 = a_1 = b_1$ holds for them such that the next execution of Step 3 will be done by case (i) resulting in values $a_1' = b_1' = 0$. Thus, the left subsequence is completely dissolved.

During the processing of the left subsequence until the stage a) or b) is reached, the gap shown by (4.9) can become smaller, but is at least of length t, that is, we have at each stage of this processing

$$G_{\text{left}} - G_{\text{right}} \geq t \tag{4.10}$$

for each c_{left} of the left subsequence and each c_{right} of the right subsequence. This is understood if the reasoning of part (ii) of Remark 1 is applied to $c_{\mu-1}$ (instead of $E_{\min}$). If now the processing discussed terminates with case a), then the assertion of Lemma 1 is not satisfied for the current values of a_1 and b_1, since one of them would belong to the left subsequence and the other to the right subsequence such that $|E_1 - F_1| \geq t$ is implied by (4.10). Therefore one of the termination conditions of Step 1 are satisfied as was the case in the discussion of (4.7) and this causes the computation to stop. Hence, such a model cannot be a worst case model. If now the above-mentioned processing terminates with stage b), where the left subsequence has zero as a partial sum such that this subsequence vanishes then the next iterate will start to work with the right subsequence. Note that in this case the size of the gap $G_{\mu-1} - G_\mu$ (with current values μ, $G_{\mu-1}$ and G_μ) plays no role at all. Hence the number of iterations remain unchanged if we assume that (4.8) holds as well. This means that we are justified in stating that the restrictions (4.7), (4.8) are consistent with he worst case scenario for ESSA.

B. Secondly, using this form of the sequence with properties (4.7), (4.8) it is easy to count the number of iterations, which are needed in order to move all the summands to the right, until their exponent is $E_{\min -t+1}$. The number of iterations is the larger the more the connected sequence is spread out, that is,

$$\begin{aligned} G_1 - G_2 &= t-1, \\ G_{\nu-1} - G_\nu &= 2t-2 \text{ for } \nu = 3,\ldots,l. \end{aligned}$$

Hence, the last number in the sequence, c_l, having $E_{\min}$ as exponent, needs at most

$$0t + 1(t-1) \text{ iterations}$$

in order to reach exponent $E_{\min -t+1}$, the one before, c_{l-1}, at most

$$1t + 2(t-1) \text{ iterations,}$$

in order to reach exponent $E_{\min -t+1}$, the one before, c_{l-2}, at most

$$2t + 3(t-1) \text{ iterations,}$$

etc. Finally, c_2 needs at most

$$(l-2)t + (l-1)(t-1) \text{ iterations}$$

and c_1 at most

$$(l-2)t + l(t-1) \text{ iterations}$$

to reach exponent $E_{\min -t+1}$.

We now have a collection of at most l processed summands with exponent $E_{\min -t+1}$. Since they arose from the initial sequence where the last occurring exponent was $E_{\min}$, they all have the form

$$\pm 0.10\ldots0 \times 2^{E_{\min -t+1}}.$$

Now only Step 3 (i) operations are applicable where each addition results in zero until $m = 0$ or $n = 0$ ensues. This needs at most $l/2$ iterations. Hence the overall sum of iterations in this worst case model is

$$\begin{aligned} & t[1+2+\cdots+(l-2)] + (t-1)[1+2+\cdots+(l-1)] \\ & +t(l-2) + (t-1)(l-1) + l/2 \\ & = l^2(t-1/2) - t \\ & \leq l^2 t. \end{aligned}$$

□

For a complexity analysis one has additionally to count the initial sorting with $O(l \log_2 l)$ arithmetic operations, further the number of operations at each iteration, which are two, and finally the updating of the lists at each iteration needing $O(\log_2 l)$ comparisons. Hence, if one had not to care about the initial restriction $l \leq 2^{t-1}$, one would relate ESSA to the $O(l^2 \log_2 l)$ complexity class.

3. The average numerical costs and speed of ESSA is, as is also the case with comparable methods, much better than the arrangement of the worst case which probably never will occur in reasonable practical applications. Looking at the numerical results in Sec. 4.5 one could conjecture that the number ought to be γl rather than tl^2 iterations. In fact, in the case of the purely randomly generated sums in Table 1 one can observe a value $\gamma \leq 0.15$.

4. *Alternatives to ESSA*: ESSA is already surprisingly simple. It can even be made still simpler, if in Step 3 (ii) always

$$u := 2^{F_1}$$

and in step 3 (iii) always

$$v := 2^{E_1}$$

is set. Practically, there is almost no difference to the course of the original ESSA, but the variant shows a few ambushes towards complexity considerations like those in Remark 2.

A second alternative is to set

$$a_1' := a_1 - 2^{F_1-1},\ b_1' := b_1 - 2^{F_1-1}$$

in Step 3 (ii) and

$$b_1' := b_1 - 2^{E_1-1},\ a_1' := a_1 - 2^{E_1-1}$$

in Step 3 (iii). This variant has the feature, that the lengths of the a_i-list and b_i-list remain in general constant in the execution of Step 3 (ii), (iii), together with Step 4. In contrast to this, the original ESSA shows the feature that if one list is already more or less dominant then this dominance will be strengthened and the other list shrunk in order to force an early termination and decision.

A further possibility is offered if the user is not adverse to splitting the mantissa in his code. The splitting has to be done so that a split part of a_1 is subtracted from b_1, or conversely, resulting in only one subtraction (instead of two in each execution of Step 3, case (ii) or (iii)). The analysis of the convergence of this modification might, however, be difficult.

5. *Dropping the restriction (4.2)*. This restriction was needed for the proof of formula (4.4), cf. the Lemma, which again was the premise that Step 3 could be performed exactly. If, however, l tends to ∞ (which is more of mathematical interest than of practical use), then (4.4) is no longer valid. Hence, one has to create (4.4) artificially, for example by creating slack summands that do not change the overall sum, S. For example, if a_1 and $b_1 < a_1$ do not satisfy $E_1 - F_1 \leq t-1$ then $\tilde{a} = \tilde{b} = 2^{E_1-t+1}$ are

appropriate slack summands where $\tilde{a}$ is put into the a_i-list, and $\tilde{b}$ is put as leading element into the b_i-list, and formula (4.4) is now valid for a_1 and $\tilde{b}$ as a leading couple.

6. The preprocessing as well as the updating of the lists depends on an ordering by the $\geq$-relation. One could also design ESSA so that it uses the $>$-relation. Then the proper steps of ESSA would be much more effective, but the preprocessing of the list would cause trouble, at least in the sense that the theory would be more difficult. The reason is that, if some part of the list is already ordered and if two elements are identical, then they have to be added or subtracted first so that the order requirement can be satisfied. This may again result in a list with two new identical elements so that the list has to be preprocessed again. It is not possible to establish a limit on this recursive behavior without further investigation.

7. The numerical results obtained in the experiments with randomly generated summands described in Table 1 of the next section show that on the average only about 5% of the summands are processed before a decision is made. This suggests that it might be beneficial to design another algorithm that simply processes summands "on the fly." That is, the largest positive summand and the negative summand with the largest modulus, a_1 and b_1 are selected, processed, and put back on the two unsorted arrays and so on, recursively. Alternatively, the largest (with respect to the modulus) 10% of the positive and negative summands are picked out and processed, putting items back in order only if they are larger than the smallest sorted summands. Since both suggestions would imply a bubble-sort in the worst case, the worst-case complexity would be $0(n^2)$; however, the average case complexity might be lower. The above-mentioned percentages, i. e. 5% and 10% depend on the sample material and can vary considerably. These ideas are being investigated.

4.5 Numerical Results

We tested ESSA with several series of randomly generated sums. Here we pick out two short series of representative examples. For both series we used the heap sort algorithm (cf., for example, [199]) for the ordering of the list of positive and negative summands in the preprocessing phase and also in an abbreviated form for the updating of the lists in Step 4 of ESSA. We also executed the test series with the bubble sort method (cf., for example, [199]), however, we rejected the results since the average execution time needed was about 100 times larger than the execution times using only heap sort. (This reflects the $O(n \log_2 n)$ to $O(n^2)$ order relationship between heap sort and bubble sort). The examples were calculated on a SUN Sparc 20 workstation with the C code given in Sec. 4.7. The length of the mantissa was $t = 24$.

For the test series, we chose sets of 10000 summands each that were machine numbers in the range $\pm[2^{-127}, 2^{127}]$. This resulted in well-conditioned sums, and we show our samples from this series of tests in Table 1. In order to create a series of ill-conditioned samples we randomly generated a sequence $(c_k)_{k=1}^{1111}$ from the above range. Then, for each c_k, we defined 9 summands, one was $8c_k$ and the remaining 8 were each $-c_k$. This resulted in 9999 summands. As a last summand we added 2^{-60} or -2^{-60}. Hence the value of the exact sum of the 10000 summands is $\pm 2^{-60}$. In Table 2 the results are shown for the ill-conditioned case in the same format as for the first series of results.

Each line in the tables describes the statistics of the application of ESSA to a sequence of summands.

m	=	number of positive summands,
n	=	number of negative summands,
Step 3		consists of 3 columns each counting the number of executions of the three cases in Step 3 of ESSA during the computation,
it	=	number of iterations,
m_t	=	number of remaining positive summands at termination,
n_t	=	number of remaining negative summands at termination,
time	=	computation time in microseconds (includes the time for the preprocessing ordering as well as the computation of S' as described below),
sgn	=	exact sign of the sum,
S'	=	result of performing the summation in double precision arithmetic starting with the largest and ending with the smallest summand.

4.6 Merging with Interval Methods, Applications

As mentioned in Sec. 4.2, there are several important applications of ESSA to typical processes in geometric computations. We discuss a few of them in more detail, and we also provide some suggestions as to when it is reasonable to combine ESSA with interval methods.

Interval methods render an inclusion of a value instead of the required value itself if implemented correctly. This means that, in the case of the sign of a sum S, an inclusion

$$S^I \text{ with } S \in S^I$$

where S^I is an interval is computed. Then S^I is interpreted as follows:

m	n	Step 3 (i)	(ii)	(iii)	it	m_t	n_t	$time$	sgn	S'
5028	4972	78	377	0	456	5331	4520	13	1	2.629359e+32
5015	4985	114	352	0	467	5260	4520	12	1	1.215179e+32
4984	5016	36	370	0	407	5318	4610	12	1	9.825325e+32
5004	4996	107	375	0	483	5283	4516	12	1	1.105210e+32
5013	4987	87	0	368	456	4558	5280	12	-1	-1.912094e+32
5052	4948	351	15	403	770	4337	4994	14	-1	-2.400341e+29
4987	5013	30	0	385	416	4572	5368	13	-1	-7.295567e+32
5034	4966	199	380	1	581	5220	4392	13	1	9.335404e+30

Table 1 Well-conditioned sum.

m	n	Step 3 (i)	(ii)	(iii)	it	m_t	n_t	$time$	sgn	S'
4983	5017	5053	1129	973	7156	951	329	33	1	1.152922e+18
4885	5115	5074	1524	1086	7685	1069	439	34	1	1.614090e+19
4976	5024	5056	1154	761	6972	1045	436	31	1	-4.611686e+18
4927	5073	5037	1381	810	7229	1105	327	32	1	-1.268214e+19
4989	5011	4944	877	1289	7111	378	1091	31	-1	-2.536427e+19
4863	5137	5071	919	1015	7006	324	1028	32	-1	-1.152922e+18
5073	4927	5159	819	1273	7252	338	996	32	-1	1.498798e+19
5073	4927	4986	831	1189	7007	440	1056	32	-1	1.383506e+19

Table 2 Ill-conditioned sum.

If	$S^I > 0$	then $S > 0$,
if	$S^I < 0$	then $S < 0$,
if	$S^I = [0, 0]$	then $S = 0$,
if	$0 \in S^I$,	no decision can been derived from this computation so far.

Since interval methods control all kinds of numerical errors, the first three cases are guaranteed. In order to get a decision in the fourth case one has to refine or even to change the method that has been applied so far.

Next we want to pursue the question, *when it is worthwhile to use ESSA* and to *combine it with interval methods*:

If one has to determine the sign of a sum, principally 3 *types* of approaches are available:

a) usual summation, various kinds of improvements (multiple precision, restricted fixed point accumulation) ,

b) interval arithmetic, methods with guaranteed error bounds, machine-exact addition,

c) ESSA.

Let us consider these three types when they are applied to a *well-posed* problem:

Type a) will always work, but one has never a guarantee that the outcome is correct.

Type b) will also work well, and one has the guarantee of a correct outcome.

Type c) works exactly, and because of the well-posing of the problem, one of the two lists will clearly be dominant such that a termination criterion will apply soon.

Le us turn to an *ill-posed* problem:

Type a) is no longer reliable.

Type b) will deliver an answer whether the result is reliable or not, but the probability of unreliable answer will increase with the degree of ill-posedness.

Type c) will still deliver the exact result, even though the decision for the sign might require a number on iterations.

Summing up: ESSA is the best choice, as long as one really needs the exact result.

Nevertheless, there are some situations where a *combined use of interval analysis with ESSA* is reasonable:

This is frequently the case if an arithmetic expression must first be prepared or replaced by one which is mathematically equivalent in order that ESSA can

be applied. Let us consider as a simple example the left-turn test, cf. Appl. 1. The computational part of this test consists in determining the sign of the determinant

$$D = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix}.$$

Nobody with computational experience would multiply through the determinant obtaining the expression

$$x_1y_2 + y_1x_3 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3 \tag{4.11}$$

which results in 6 products and 5 additions. ESSA requires the expression to be a sum of elements, however, and the only way of achieving this is to use (4.11). Procedures of type a) or b) would probably use an expression like

$$x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)$$

requiring 3 products and 5 additions and being more stable than the former expression. Another expansion is

$$(x_1 - x_3)(y_1 - y_3) - (x_2 - x_3)(y_2 - y_3) \tag{4.12}$$

which requires only 2 products and 5 additions. However, ESSA is not able to evaluate the latter two expressions because it can not handle the differences $x_1 - x_3$, $x_2 - x_3$, $y_2 - y_3$, $y_1 - y_3$, $y_1 - y_2$ so that exactness is maintained. The differences in the computational performance of these simple expressions are certainly not significant.

The difference will increase considerably when we pass to other applications, where determinants of higher dimension are involved or where the items of the determinants are again arithmetic expressions (cf. Appl. 2, 3, 4) such that the determinant has to be multiplied through completely in order to generate an expression that ESSA can handle.

A similar situation arises if multiple precision is necessary in order to get the sign guaranteed and one does not want the programming hassle with the multiple precision but splits it into a sum of lower precision numbers. Then one also has to multiply through the expression until it is represented as a sum of the lower precision numbers. For details see Appl. 2.

These examples show that it is not always wise to use ESSA for each occurring sign determination. There are situations where we recommend merging ESSA with machine interval arithmetic as follows:

1. Determine the required sign with a stable *low complexity expression* using *single precision machine interval arithmetic.*

   This will, in general, be sufficient for a decisive sign determination. In the rare remaining cases, there is no way out and one has

2. to go through the higher complexity calculation using ESSA.

Let us consider now a few typical applications of ESSA.

Application 1 (*Left-turn test*)

Let p_1, p_2, p_3 be 3 points in the plane so that $p_i = (x_i, y_i)$, $i = 1,2,3$. If $p_1\bar{p}_2$ is the line through p_1 and p_2, then p_3 is to the left, on or to the right of $p_1\bar{p}_2$ (looking from p_1 to p_2) iff

$$D = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} \tag{4.13}$$

is positive, zero or negative. Expanding the determinant we get a sum

$$D = x_1 y_2 + x_2 y_3 + x_3 y_1 - y_1 x_2 - y_2 x_3 - y_3 x_1. \tag{4.14}$$

If $p_i = (x_i, y_i)$, $i = 1,2,3$ is represented in single precision arithmetic, then the products appearing in (4.14) can be computed exactly using double precision arithmetic. Thus, one has two ways to proceed.

1. Apply a double precision version of ESSA to the sum (4.14). (If one only wants to use single precision then one can do this by doubling the number of summands.) One always gets the right sign.

2. Apply first a single precision machine interval arithmetic version to the expression (4.12) or an equivalent expression. If no guaranteed sign of D is delivered, execute the calculation as in 1. with ESSA.

The left-turn test is one of the most important applications of ESSA. It is used for example, in the *Graham scan algorithm* [198] for determining the convex hull of a plane point set, cf. Sec. 4.2. The complexity order of Graham's algorithm is $O(n \log_2 n)$ if n is the number of points and exact arithmetic underlies the computation. If the point set is read into the computer, an initial input data error called *conversion error* is *unavoidable*. But no further errors need to be accepted; one just has to replace each point of the input data set by the smallest machine representable set that contains that point. Clearly, the set is stored by means of its corners, and can be a rectangle, a straight line segment, or the point itself. Now we determine the convex hull of this enlarged (exactly represented) point set with the left-turn test. If ESSA is used for the left-turn test, we get the exact convex hull of the enlarged point set which is the smallest machine representable convex set that contains the convex set that would be generated by the original input data set.

The importance of the construction of the convex hull for a point set is best demonstrated by various replacement procedures that have now been developed to overcome the lack of a reasonable exact left-turn test. For example, Knuth

[137] defined axioms that relate points in the Euclidean plane to each other via the sign of the determinant (4.13). Based on these axioms he defines a convex hull and error-free algorithm for computing convex hulls provided the sign of the determinant can be computed exactly. The determinant (4.13) is therefore expanded as in (4.12). If the input data is rounded to a fixed point range of b bits then it is shown that $2b+1$ bit arithmetic suffices for exact computation of the determinant.

The order of the algorithm remains $O(n \log_2 n)$ since the computational cost of the interval or exact computation is constant. The above determinant also indicates the orientation of the three points. If $D < 0$ then $p_1p_2p_3$ form a counterclockwise cycle and if $D > 0$, $p_1p_2p_3$ form a clockwise cycle with $D = 0$ indicating collinearity.

Application 2 (*Coplanarity test in 3D*).

In three dimensions it is well known that a point p_4 is on a plane defined by three non-collinear points p_1, p_2, p_3 where $p_i = (x_i, y_i, z_i), i = 1, \ldots, 4$, iff

$$D = \begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} = 0.$$

Expanding the determinant we obtain a sum of products of the form $\mp x_i y_j z_k$. If these products are computed in triple precision, then ESSA can be used to obtain exact results for whether the point is on the plane or not. I.e., if s is the mantissa length in use, for example, single precision mantissa, the mantissa length used in ESSA has to be $t = 3s$ in order that exact result can be expected. If one does not want to write codes in triple precision numbers then one can write these as a sum of two double precision numbers (or even as a sum of three single precision numbers) by just splitting the longer numbers and hence one does not work beyond the basic tools of C or C++.

In the triple precision case, $t = 3s$, ESSA has to deal with 24 summands, in the double precision case, $t = 2s$, with 48 summands, and in the single precision case, $t = s$, with 72 summands.

It is highly recommended when executing the coplanarity test, to first compute D or the sign of D in single precision interval arithmetic. ESSA should only be used if the result is not decisive.

With this approach the test can be employed as a primitive in the gift wrapping method [198] thus becoming a rounding error free algorithm for the convex hull of exactly representable numbers in 3D. Similarly, as in Appl. 1, if the originally given point set does not only consist of machine numbers, the gift wrapping method renders the smallest convex hull with machine representable vertices that encloses the convex hull of the original data.

The gift wrapping method with this primitive retains the $O(n \log_2 n)$ worst case complexity since the computational costs of the interval or exact compu-

tations are constant since they are independent of n, the number of points in the given set.

Application 3 (*Order of 3 lines in plane*)
Let three lines in the plane be defined by

$$a_i x + b_i y + c_i = 0, \ i = 1, 2, 3.$$

The determinant

$$D = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

determines the following relationships between the lines:

1. if $D < 0$ then the lines are oriented clockwise,

2. if $D = 0$ then the three lines either intersect in a point or at least two of the lines are parallel,

3. if $D > 0$ then the lines are oriented counterclockwise.

   (Three lines l_1, l_2, l_3 in the plane are said to be *oriented clockwise* (*counterclockwise*) if

   (i) l_1 and l_2 intersect in a point p_3,
   l_1 and l_3 intersect in a point p_2,
   l_2 and l_3 intersect in a point p_1,

   (ii) p_1, p_2, p_3 are unequal,

   (iii) running through the triangle with vertices p_1, p_2, p_3 in this order is done clockwise (counterclockwise)).

The complete procedure for computing the sign of D is analogous to the procedure in the previous application. The procedure can be seen to form the core primitive of algorithms to compute line arrangements in the plane [190]. Again, ESSA combined with interval arithmetic can be applied successfully to receive a guaranteed result, provided the input data are machine numbers.

Application 4 (*In-circle-test*)
Let $p_i = (x_i, y_i), i = 1, 2, \ldots 4$ be four points in the plane and assume p_1, p_2, p_3 (not collinear) define a circle C. Then the relationship of p_4 to C is determined by the sign of the determinant

$$D = \begin{vmatrix} x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \\ x_4 & y_4 & x_4^2 + y_4^2 & 1 \end{vmatrix}.$$

Assume that p_1, p_2, p_3 in this order lie clockwise on the circle. (This can be checked with the left-turn test, cf. Appl. 1.) Then

if	$D > 0$	then	p_4 is inside C,
if	$D = 0$	then	p_4 is on C,
and if	$D < 0$	then	p_4 is outside C.

When the determinant is multiplied through then products of the form

$$x_i y_j (x_k^2 + y_k^2) = x_i y_j x_k^2 + x_i y_j y_k^2$$

result. Each product of the form $x_i y_j x_k^2$ would require quadruple precision for getting exact results with ESSA if the points p_i are of single precision. However, those expressions could also be accomplished by 4 double precision quantities without too much mantissa manipulations, hence remaining thus in a comfortable environment of the language C. This is done as follows: We start with x_i, y_j, x_k in single precision, compute the products $x_i y_j$ and x_k^2 in double precision, but split each of them immediately in the sum of two single precision numbers,

$$x_i y_j = (x_i y_j)_L + (x_i y_j)_R, \; x_k^2 = (x_k^2)_L + (x_k^2)_R.$$

Finally, we execute the four products

$$(x_i y_j)_\nu (x_k^2)_\mu, \; \nu, \mu = L, R$$

in double precision. Their exact sum is just $x_i y_j x_k^2$. Hence the determinant is the sum of 192 double precision quantities, and the computation of the sign of D can be done exactly by ESSA.

In [137] the determinant is computed using as sophisticated analysis of expansion in minors obtaining an exact result assuming that the input data was rounded into a certain fixed point range. The same problem is considered in [13], where the exact sign of a 2×2 determinant is computed.

The in-circle test is accepted as a primitive for the so-called incremental method for the construction of a Voronoi diagram [123]. In [259] numerical error is reduced by computing the primitive to quadruple precision.

As in the former applications, we recommend the combination of single precision machine interval arithmetic with ESSA in order to get exact results. This makes, however, sense only then if the input parameters p_i are already machine numbers, which is, for example, the case if the p_i's stem from other calculations. This also holds for Appl. 3, but not for Appl. 1 and 2, where the exact sign of the determinants at least guarantee the convexity of the hull computed from the smallest possible boxes including the input parameters after their conversion to machine numbers.

4.7 ESSA and Preprocessing Implementation in C

The following C code due to G. Mackenbrock implements Alg. 9. The sort algorithm is from [199]. It can be replaced by any other suitable sort algorithm.

```
*************************************************************
/*-----------------------------------------------------------
Function :sgnsum
Description: sgnsum calculates the exact sign of
             the sum of the S[i].
Input: S..  an array of summands of type float;
       nS.. length of S.
Output: the sign of the sum.
Local variables: n.. length of the list b;
                 m.. length of the list ;
                 E.. exponent of a[1];
                 F.. exponent of b[1];
                 sg.. -1 -> sum negative
                       0 -> sum 0
                      +1 -> sum positive
the remaining variables are auxiliary variables.
-----------------------------------------------------------*/
int sgnsum(float *S, int l)
{
 int n,m,E,F,i,sg;
 float *a,*b,as,ass,bs,bss,uu,u,v;

/*-----------------------------------------------------------
Initialization of the lists a and b.
-----------------------------------------------------------*/
 if ( (a=(float*)calloc(l+3,sizeof(float)))==NULL )
 {
  printf("No mem. Program terminated.\n");
  exit(1);
 }
 if ( (b=(float*)calloc(l+3,sizeof(float)))==NULL )
 {
  printf("No mem. Program terminated.\n");
  exit(1);
 }
```

```
/*-------------------------------------------------------------
Splitting of S into positive and negative summands.
-------------------------------------------------------------*/
 n=m=0;
 for (i=1;i<=l;i++)
  if (S[i]>0) a[++m]=S[i];
  else if (S[i]<0) b[++n]=fabs(S[i]);

/*-------------------------------------------------------------
Sorting of the lists a and b in descending order.
-------------------------------------------------------------*/
 if ( m>1 && sg==-2 ) sort(m,a);
 if ( n>1 && sg==-2 ) sort(n,b);

/*-------------------------------------------------------------
Main loop (the proper algorithm ESSA).
-------------------------------------------------------------*/
LoopStart:

/*-------------------------------------------------------------
 Step 1: (Termination Criteria)
-------------------------------------------------------------*/
  if ( n==0 && m==0 ) { sg=0; goto LoopEnd; }
  if ( n==0 ) { sg=1; goto LoopEnd; }
  if ( m==0 ) { sg=-1; goto LoopEnd; }
  frexp(b[1],&F);
  if ( n==0 || a[1]>=ldexp(n,F) && m>0 ) { sg=1; goto LoopEnd;}
  frexp(a[1],&E);
  if ( m==0 || b[1]>=ldexp(m,E) && n>0 ) { sg=-1; goto LoopEnd;}

/*-------------------------------------------------------------
 Step 2: (Auxiliary variables)
-------------------------------------------------------------*/
  as=ass=bs=bss=0;

/*-------------------------------------------------------------
 Step 3: (Comparision and processing of the leading
         summands of the lists)
-------------------------------------------------------------*/

/*-------------------------------------------------------------
 E contains the exponent of a[1] and F the exponent of b[1] in
 base 2.
-------------------------------------------------------------*/
```

```
/*-------------------------------------------------------
 Step 3, case (i):
--------------------------------------------------------*/
  if (E==F)
  {
   if (a[1]>=b[1]) as=a[1]-b[1];
   else bs= b[1]-a[1];
  }

/*--------------------------------------------------------
 Step 3, case (ii):
--------------------------------------------------------*/
  else if (E>F)
  {
   uu= ldexp(1,F-1);
   if (b[1]==uu) u=uu;
   else u=uu*2;
   as=a[1]-u;
   ass=u-b[1];
  }

/*-------------------------------------------------------
 Step 3, case (iii):
--------------------------------------------------------*/
  else if (F>E)
  {
   uu=ldexp(1,E-1);
   if (a[1]==uu) v=uu;
   else v=uu*2;
   bs= b[1]-v;
   bss= v-a[1];
  }

/*-------------------------------------------------------
 Step 4: (Rearrangement of the lists, keeping S constant.)
--------------------------------------------------------*/

  if (as==0 && ass==0)
  {
   a[1]=a[m];
   m--;
   BuildHeapFromTop(m,a);
  }
```

```
  if (as==0 && ass!=0)
  {
   a[1]=ass;
   BuildHeapFromTop(m,a);
  }
  if (as!=0 && ass==0)
  {
   a[1]=as;
   BuildHeapFromTop(m,a);
  }
  if (as!=0 && ass!=0)
  {
   a[1]=as;
   BuildHeapFromTop(m,a);
   a[++m]=ass;
   BuildHeapFromBelow(m,a);
  }

  if (bs==0 && bss==0)
  {
   b[1]=b[n];
   n--;
   BuildHeapFromTop(n,b);
  }
  if (bs==0 && bss!=0)
  {
   b[1]=bss;
   BuildHeapFromTop(n,b);
  }
  if (bs!=0 && bss==0)
  {
   b[1]=bs;
   BuildHeapFromTop(n,b);
  }
  if (bs!=0 && bss!=0)
  {
   b[1]=bs;
   BuildHeapFromTop(n,b);
   b[++n]=bss;
   BuildHeapFromBelow(n,b);
  }
 goto LoopStart;
LoopEnd:
```

```
 free(b);
 free(a);

 return sg;
}
/*------------------------------------------------------------
Function sort:
Description: Heapsort from ''Numerical Recipes in C'' by Press,
   Flannery, Teukolsky and Vetterling, Cambridge University
   Press, page 247.
Input: ra.. floating point numbers;
       n.. length of ra.
Output: ra sorted in descending order.
------------------------------------------------------------*/
void sort(int n, float *ra)
{
 int l,j,ir,i;
 float rra;

 l=(n >> 1)+1;
 ir=n;
 for(;;)
 {
  if (l>1)
   rra=ra[--l];
  else
  {
   rra=ra[ir];
   ra[ir]=ra[1];
   if (--ir==1)
   {
    ra[1]=rra;
    return;
   }
  } /* (l<=1) */
  i=l;
  j=l<<1;
  while (j<=ir)
  {
   if ( j<ir && ra[j]>ra[j+1] ) ++j;
   if ( rra>ra[j] )
   {
    ra[i]=ra[j];
    j+=(i=j);
```

```
   }
   else
    j=ir+1;
  }
  ra[i]=rra;
 } /* for(;;) */
}
/*------------------------------------------------------------
Function BuildHeapFromTop:
Description: The heap property for ra is reestablished under
   the assumption that the property is only violated at the
   root (a[1]) of the heap.
Input: ra,n.
Output: ra
-------------------------------------------------------------*/
void BuildHeapFromTop(int n, float *ra)
{
 int i=1,m;
 float top=ra[1];

 while (2*i<=n)
 {
  m= 2*i;
  if ( ra[m]<ra[m+1] ) if (m<n) m++;
  if (top<ra[m]) { ra[i]=ra[m]; i=m; }
  else break;
 }
 ra[i]=top;
}
/*------------------------------------------------------------
Function BuildHeapFromBelow:
Description: The heap property for ra is reestablished under
   the assumption that the property is only violated at the
   place n in the heap.
Input: ra,n.
Output: ra.
-------------------------------------------------------------*/

void BuildHeapFromBelow(int n, float *ra)
{
 int i=n,m;
 float last=ra[n];

 while (i/2>0)
```

```
 {
  m= i/2;
  if ( ra[m]<last ) { ra[i]=ra[m]; i=m;}
  else break;
 }
 ra[i]= last;
}
*****************************************************
```

Chapter 5

Intersection Tests

In this chapter a class of intersection problems are considered where the use of interval arithmetic can facilitate the geometrical and algorithmic understanding and hence the logical design and computational effort required. This is, surprisingly, not only the case, if one of the objects can be represented by intervals directly, but also if certain parts of the algorithm need an exact determination of the range of a function and if this can be obtained partially or completely with interval tools.

5.1 Introduction

An important class of primitives for geometric computations are intersection operations. Many geometric computations simulate real world actions in two and three dimensions and it is clear that, for example, in solid modelling two objects in three dimensions cannot occupy the same space at the same time. Determining whether this is the case or not involve, in part, intersection tests.

This is certainly completely trivial for most people. If the two objects, however, are not directly visible, for instance, if the objects are presented only by their data, such as their vertices (tetrahedron, cube, etc.), by midpoint and radius (ball), or other parameters, an immediate answer to whether the objects do intersect or not, might not be available. This means that the relationship between the two objects has to be determined. If their relationship is such that they do occupy the same space at the same time, it follows that the simulation cannot describe a real world configuration. A robotics manipulator cannot lift an object from one table and place it on another table if there are objects in the path of the movement. In computer graphics one object may obscure another object seen from some viewpoint. In order to confirm that the obscured object need not be displayed one must have that the projection of the first object is included in the second object and that the second object is farther away from

the viewpoint than the first object. As an industrial example the plates to be welded into the hull of a ship can only be cut from pieces of sheet steel if they do not intersect. In geographical information systems, which are becoming increasingly important, roads can be drawn to be constructed at less cost if they do not intersect expensive objects such as buildings. Planar integrated circuits contain hundreds of thousands of elements. These elements must be shown to not mutually intersect and their data paths must be routed in such a manner that no intersections occur.

The problems encountered in devising procedures for intersection computations are hinted at in [255]:

> Singular configurations are frequently ignored in these treatments although they must be addressed in practical applications. (In a singular configuration, two solids intersect in such a way that small perturbations in location change the topology of the boundary of their intersection).

There are essentially two types of intersection algorithms:

1. The first type of intersection algorithm processes two objects, and returns Boolean answers for intersection and no intersection. An extension of this is the information that one object is included in the other object.
2. The second type of intersection algorithm returns the actual intersection, i.e. a point, a plane figure or a solid.

Both types of intersection algorithms have been studied extensively in a variety of areas of geometric computations. In solid modelling Zeid [280] states on page 360:

> In various geometric problems involving solid models, we are often faced with the following question: given a particular solid, which point, line segment of a portion of another solid intersects with such a solid. These are all geometric intersection problems.

He then proceeds to discuss both the general intersection problem and specific cases of intersection problems. Specifically he considers ray/box intersection on p. 563 and mentions surface/surface intersections of quadric solids on p. 404. The book includes a total of 15 references to various intersection problems showing the importance of the intersection algorithms for solid modelling. In Preparata-Shamos [198] we find the geometric intersection problem studied from a point of view of computational geometry. In this context the intersection algorithms are studied with respect to order i.e. the algorithms are studied with respect to computational cost as the number of points in a polygon or the number of objects etc. increase. Intersection algorithms are also of great importance in computer graphics, in particular in the hidden line and hidden

surface problem where only visible lines and surfaces need be considered. It is furthermore of interest for ray tracing where intersections of rays with surfaces have to be computed. Particular techniques have been developed in this area, such as z-buffering see for example Foley et al. [54].

There is also grass-roots interest in the intersection problems as exemplified by the following posting in a newsgroup on the Internet:

> I am looking for good algorithms to perform the following (boolean) tests:
>
> 1. (convex) polygon/circle intersection,
> 2. box/cone intersection.
>
> Can anybody offer any advice as to where to find these routines (references to papers particularly preferred) - no joy in the graphics gems[1].

Another question to a newsgroup (March 9, 1999) was:

> I need to determine the existence of an intersection between a cube and a sphere. I have found a solution that solves the problem, but I would like to apply the most efficient proposal. Which is the most efficient algorithm that solves this problem?

The objects that are treated in the geometric computations may often be thought of as sets in two or three dimensions. If these sets have some regularity properties, then it is sometimes possible to treat them within the field of interval analysis. A box with sides parallel to the coordinate axes in three dimensions can be represented by a three-dimensional interval vector. Such boxes often arise in geometric computations from bounding objects by faces perpendicular to the coordinate axes [10]. Similarly, a rectangle in two dimensions with sides parallel to the coordinate axes can be represented by a two-dimensional interval vector. This often leads to the natural use of interval analysis for intersection problems, in particular, for the first type of intersection problems mentioned above.

In this chapter we therefore introduce elementary interval techniques to intersection computations.

The advantage of interval analysis implementations of some of these computations are:

1. They tend to produce simple, clear algorithms that are relatively easy to implement provided a robust implementation of interval arithmetic is available.

[1]The writer of the posting wanted to express that he found no suitable algorithms in Graphics Gems, a series of 5 volumes concerned with graphics algorithms (see for example [101]).

2. They provide guaranteed results for a number of intersection computations.

In some cases these algorithms are more computationally expensive than the algorithms that can be developed using a more detailed case-by-case analysis. We also present the more efficient algorithm for those problems in spite of the fact that the easy overview is lost. This is analogous to the simple definition of interval multiplication given in Eq. (2.4) as compared to the more complex definition of interval multiplication given for example in [169].

As an introduction to this kind of algorithm we discuss intersection tests for axis-parallel rectangles.

$$\left.\begin{array}{l} A = ([\mathrm{lb}(A_x), \mathrm{ub}(A_x)], [\mathrm{lb}(A_y), \mathrm{ub}(A_y)]) \\ B = ([\mathrm{lb}(B_x), \mathrm{ub}(B_x)], [\mathrm{lb}(B_y), \mathrm{ub}(B_y)]) \end{array}\right\} \in I^2.$$

Then A, B represent two axis-parallel rectangles in the plane. Now compute

$$C = A - B \tag{5.1}$$

in interval arithmetic. Then we have the following results:

1. If $0 \in C$ then the rectangles A and B intersect.

2. If $0 \notin C$ then the rectangles A and B are disjoint.

Note that $0 \in C$ means that zero is a member of each interval coordinate and that $0 \notin C$ means that zero is not a member of at least one coordinate interval. The reason that the test is valid is that zero can only be in C if there is a point $x \in A \cap B$, i.e. A and B intersect. If there is no such point then zero is not in the intersection and A and B are disjoint.

As a simple example consider $A = ([2,4],[1,2])$ and $B = ([1,3],[0,2])$ as shown in Figure 5.1. The interval subtraction results in $C = ([-1,3],[-1,2])$. Since zero is a member of each interval coordinate the rectangles intersect.

Clearly the above results 1. and 2. could also have been achieved via comparisons and tests of the boundaries of the rectangles. This latter method is called the *direct method* as opposed to the *interval method*. A comparison gives for exact arithmetics:

Direct method	Interval method
if $\max(\mathrm{lb}(A_x), \mathrm{lb}(B_x)) \leq \min(\mathrm{ub}(A_x), \mathrm{ub}(B_x))$ and if $\max(\mathrm{lb}(A_y), \mathrm{lb}(B_y)) \leq \min(\mathrm{ub}(A_y), \mathrm{ub}(B_y))$ then intersection otherwise no intersection	if $0 \in A - B$ then intersection otherwise no intersection

If the two methods are implemented on a computer, the direct method is error free as long as the vertices are machine numbers. If the input vertices

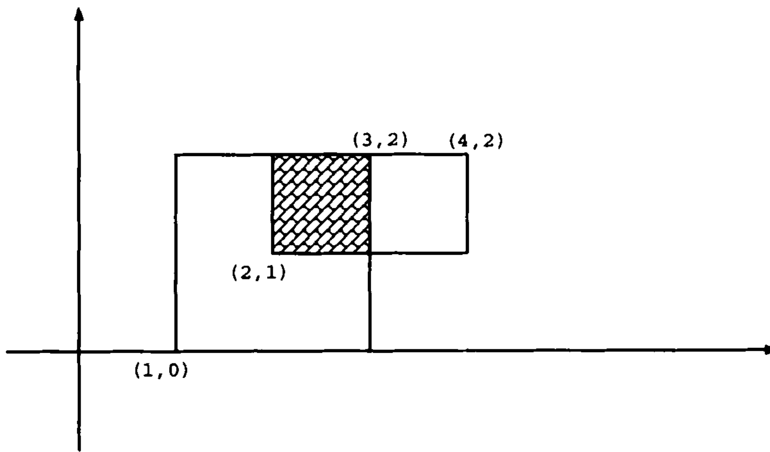


Figure 5.1: Intersection of two axis-parallel rectangles

are not machine numbers there is a small possibility of getting a wrong result because of conversion errors.

When interval arithmetic is implemented on a computer as machine interval arithmetic with the common outward rounding and a result

$$C_{out} = A - B$$

is obtained, the conclusion

$$0 \notin C_{out} \Rightarrow A, B \text{ disjoint} \tag{5.2}$$

will always be correct, even if the vertices are not machine numbers. It is, however, possible that A and B are disjoint but that the computation shows the result

$$0 \in C_{out}$$

because of the outward rounding. Such a constellation happens infrequently and can only occur if the vertices of the rectangles are machine numbers and if the distance of the two rectangles has the smallest possible positive value. (This value depends on the related components and the meaning is that if the rectangle vertices are perturbed by an arbitrarily small amount towards each other so that the vertices remain machine representable, the rectangles will no longer be disjoint. This situation could be discovered easily if desired.)

If such an extreme constellation is not given, the conclusion

$$0 \in C_{out} \Rightarrow A \text{ and } B \text{ intersect}$$

is valid.

Alternatively, the rectangle difference can additionally be executed using inward rounding (available in almost all interval packages) such that a rectangle

$$C_{in} = A - B$$

is obtained. This allows the conclusion

$$0 \in C_{in} \Rightarrow A \text{ and } B \text{ intersect.} \qquad (5.3)$$

The advantage of the combined application of inward and outward rounding is that the simple logical structure of the intersection test is kept. The arithmetical result of the difference is then an almost identical nested pair of rectangles which are appropriate for the test phases (5.2) and (5.3) and which always render a correct result.

ESSA is an appropriate tool for those constellations which neither satisfy (5.2) nor (5.3) since the exact test condition $0 \in C$ or $0 \notin C$ is equivalent to determining the signs of the differences of the components of the related vertices. All these 3 cases together give a complete and consistent test provided the vertices of the rectangles are machine numbers.

If the appropriate endpoints of the edges are subtracted using outward rounding instead of computing the inward rounding in the computation of the difference of the rectangles then pairs of intervals result which is equivalent to inward rounding of the intervals.

More about inward rounding can be found in Sec. 5.3.

In the remainder of this chapter we discuss further, more sophisticated intersection problems.

5.2 Line Segment Intersections

We consider: *how to computationally test reliably if two line segments* (shortened in the sequel as segments) *intersect in the plane.* The test should also distinguish between intersections where an endpoint of one segment lies on the second segment and where the intersection point is between the endpoints of the segments for both segments. This test is frequently a part of algorithms in geometric modelling, computer graphics, GIS and computational geometry, to name a few areas.

The test is often implemented on a computer using single or double precision floating point arithmetic. One implementation is to compute the intersection point of the lines which the segments belong to then checking if the intersection point lies on both segments. All of the implementations using fixed precision floating point arithmetic may fail due to the numerical errors causing multiple, missed or displaced intersections.

The paper by Douglas [35] discussed an early implementation of a line intersection testing routine. His description is typical of what happens when

trying to implement even simple routines reliably on a computer. The problem was optimistically stated as:

> Simple in concept, but tricky. I want a general purpose subroutine in FORTRAN which will tell if two line segments in the plane cross each other,...

He then discovered that when it came to dealing with parallel segments, close to parallel segments, segments that overlap, multiple segments meeting at a point etc. inconsistencies could occur and he realized that

> All of these inconsistencies eventually drag the programmer down from his high level math (i.e. algebra), through computer language (i.e. FORTRAN), into the realm of the machine methods actually used to perform arithmetic operations and their restrictions.

He eventually had a routine were the original problem had been split into 36 cases. The program worked most of the time.

A general method for avoiding the numerical errors and degeneracies caused by the rounding errors in finite precision floating point arithmetic is to use exact computations. When the intersection is determined exactly the above problems cannot occur [62, 63, 80, 123]. The approach works under the reasonable assumption that the data items under consideration are originally represented as fixed-precision floating-point numbers. Then, in each step of the algorithm, the exact values of all the components are calculated, which leads to the correct result [62, 63].

Most of the approaches that deal with the problem of testing for segment intersection exactly, compute the point of the segment intersection in simulated real arithmetic [35, 63, 106, 119, 159]. Such a simulation can for example be done in rational arithmetic and up to a point, in multiple precision arithmetic. Since these arithmetics are computationally expensive a two step approach is often employed where interval arithmetic is first applied as a filter (see Sec. 1.8) followed by the more expensive simulated real arithmetic when interval arithmetic fails to give a guaranteed result [119, 134].

In the remainder of this section we follow the approach of [67] where the algorithm that tests for the intersection of two line segments exactly, uses fixed precision floating point arithmetic but without actually computing the intersection point. The algorithm considers the point of intersection between two segments as the solution of a system of linear equations. First, an interval filter is applied to determine whether the segments intersect. When an inconclusive answer is obtained, ESSA is applied in together with the equations for the intersection point to determine whether the segments intersect without computing the intersection point. In comparison with the routine described in [35] the programmer is elevated from the low level of machine methods to at least the level of C code, and number of cases required in the approach are reduced

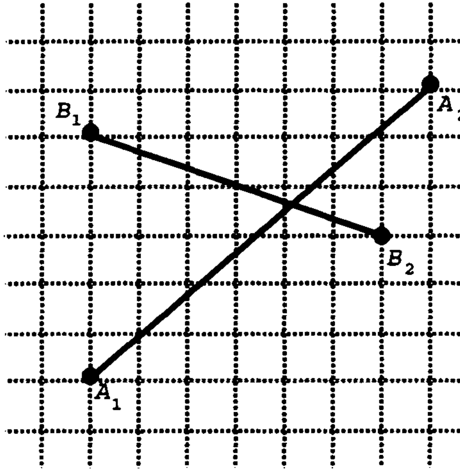


Figure 5.2: Intersecting segments

to the 10 more manageable cases given by the algorithm in the sequel which when implemented as a computer program results in a program that always provides the correct answer.

We now give a more precise definition of the line segment intersection test. Given are two line segments $\overline{A_1A_2}$ and $\overline{B_1B_2}$ in R^2 with endpoints defined by vectors $A_1 = (a_{1x}, a_{1y})$, $A_2 = (a_{2x}, a_{2y})$ and $B_1 = (b_{1x}, b_{1y})$, $B_2 = (b_{2x}, b_{2y})$. We want to determine reliably whether the two line segments intersect or not.

The coordinates of endpoints of the segments along the x and y axes are assumed to be represented as machine numbers with fixed precision, i.e. the endpoints of the segments are aligned to the grid defined by the representable floating point numbers (see Figure 5.2).

Let the equation of the line segment between A_1 and A_2 be $X = t_1A_1 + (1 - t_1)A_2$, $t_1 \in [0,1]$. Analogously for the segment $\overline{B_1B_2}$. Then the two segments intersect if and only if the equation

$$t_1A_1 + (1 - t_1)A_2 = t_2B_1 + (1 - t_2)B_2 \tag{5.4}$$

is solvable for $t_1, t_2 \in [0,1]$.

This can be written as the following system in the unknowns t_1, t_2:

$$\begin{cases} a_{1x}t_1 + a_{2x}(1 - t_1) = b_{1x}t_2 + b_{2x}(1 - t_2), \\ a_{1y}t_1 + a_{2y}(1 - t_1) = b_{1y}t_2 + b_{2y}(1 - t_2), \\ 0 \le t_1 \le 1, \\ 0 \le t_2 \le 1. \end{cases}$$

Let U be the determinant of the coefficient matrix of the 2 equations which can be obtained after a simple rearrangement of the equations above. Then

$$U = a_{1x}b_{2y} - a_{1x}b_{1y} - a_{2x}b_{2y} + a_{2x}b_{1y} - b_{2x}a_{1y} + b_{2x}a_{2y} + b_{1x}a_{1y} - b_{1x}a_{2y}.$$

If $U \neq 0$, the two equations are solvable, and we get by the well-known Cramer's rule that

$$t_1 = \frac{D_1}{U} = \frac{-b_{2x}b_{1y} - a_{2x}b_{2y} + a_{2x}b_{1y} + b_{2x}a_{2y} + b_{1x}b_{2y} - b_{1x}a_{2y}}{U}, \quad (5.5)$$

$$t_2 = \frac{D_2}{U} = \frac{a_{1x}b_{2y} - a_{1x}a_{2y} - a_{2x}b_{2y} - b_{2x}a_{1y} + b_{2x}a_{2y} + a_{2x}a_{1y}}{U}. \quad (5.6)$$

Furthermore $U = 0$ if and only if the two equations of the system are linearly dependent, that is, the two segments $\overline{A_1A_2}$ and $\overline{B_1B_2}$ are parallel.

The original intersection problem is now transformed to the *validated* investigation whether solutions t_1 and t_2 of the two equations exist at all and whether both t_1 and t_2 belong to the interval $[0,1]$. This will be the background of the intersection test and the following algorithm.

The algorithm begins with the determination of the sign of the determinant U, which can be done exactly if ESSA is used. First the case is considered when the determinant U is not equal to 0. Then it is determined whether t_1 is larger than 0. To do this it suffices to compute *the sign* of t_1 exactly. This can also be done by ESSA. To compare t_1 against 1 one must compute the sign of the difference $(D_1 - U)$ exactly. This can be done with ESSA too. Then the analogous action is applied to t_2.

The various constellations which can occur are grouped in Fig. 5.3. Case A shows the two cases of no intersection, i.e. where the generated lines intersect outside the segments (Case A, left figure) and where they intersect inside at least one segment (Case A, right figure). In Case B the situations where an endpoint of one segment is on the other segment and where two segment endpoints coincide are included. In Case C the two lines intersect with the intersection point being strictly between the endpoints of both line segments.

These cases will be referred to in the algorithm as well. The parameter *Flag* in the algorithm is incorporated in order to demonstrate the connections to the cases in Fig. 5.3.

Consider now the degenerate case when the two segments are parallel (i.e. $U = 0$). To determine whether the segments intersect we check whether one endpoint, B_1 of one segment $\overline{B_1B_2}$ belongs to the straight line passing through the other segment $\overline{A_1A_2}$. This can be done by applying the *CCW* (Counter Clock Wise) orientation test and computing the sign of the underlying determinant exactly by using ESSA as shown in Sec. 4.6. Let

$$CCW(A_1, A_2, B_1) = \begin{vmatrix} 1 & a_{1x} & a_{1y} \\ 1 & a_{2x} & a_{2y} \\ 1 & b_{1x} & b_{1y} \end{vmatrix}. \quad (5.7)$$

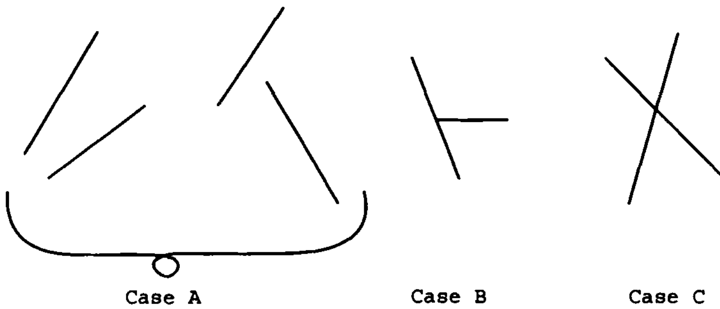


Figure 5.3: The three cases of linesegment intersection

Then this test says if $CCW(A_1, A_2, B_1) = 0$ then all three endpoints (and hence all four) lie on the same line. In this case the coordinates of the endpoints are checked to make a final conclusion about whether the two segments intersect or not.

The algorithm proceeds in the following manner:

ALGORITHM 10 *(Line segment intersection test)*

Step 1. *Set Flag* $= 0$.

Step 2. *Compute the sign of the denominator U using ESSA.*

Step 3. *If $U \neq 0$ then*

1. *For $i = 1, 2$*
   (a) *Compute the sign of the numerator D_i using ESSA.*
   (b) *If $sgn(D_i) = 0$ then $D_i/U = 0$.*
   (c) *If $sgn(D_i) = -sgn(U)$ then $D_i/U < 0$. STOP (No intersection, Case A.)*
   (d) *Otherwise, $D_i/U > 0$. Compare D_i/U to 1:*
      i. *Compute the sign of the expression $(D_i - U)$ using ESSA.*
      ii. *If $sign(D_i - U) < 0$ then $D_i/U < 1$. $Flag := Flag + 5$.*
      iii. *If $sign(D_i - U) = 0$ then $D_i/U = 1$. $Flag := Flag + 3$.*
      iv. *If $sign(D_i - U) > 0$ then $D_i/U > 1$. STOP (No intersection, Case A.)*
2. *If $Flag = 10$ then STOP (Intersection, Case C) otherwise STOP (Intersection, Case B.)*

Step 4. *If $U = 0$ then*

1. *If $CCW(A_1, A_2, B_1) \neq 0$ then STOP (No intersection) .*

2. *Otherwise,* $CCW(A_1, A_2, B_1) = 0$
   (a) *If* A_1*,* A_2*,* B_1 *and* B_2 *lie on the same vertical line, i.e.* $a_{1x} = a_{2x} = b_{1x}$ *then compare the y coordinates of the points:*

      If $a_{1y} \leq b_{1y} \leq a_{2y}$ *or* $a_{1y} \leq b_{2y} \leq a_{2y}$ *then*
      STOP (Intersection).
      Otherwise STOP (No intersection)

   (b) *Otherwise,* A_1*,* A_2*,* B_1 *and* B_2 *do not lie on the same vertical line. Compare their x coordinates:*

      If $a_{1x} \leq b_{1x} \leq a_{2x}$ *or* $a_{1x} \leq b_{2x} \leq a_{2x}$ *then*
      STOP (Intersection).
      Otherwise STOP (No intersection)

We note that each of the sign comparisons can be performed exactly in a fixed precision floating point machine arithmetic if the components of the segment endpoints are machine numbers. Nevertheless, it is recommended to apply an interval filter before the comparisons to reduce the average computational cost. The values of D_1, D_2 and U should be evaluated in nested form in case of the interval arithmetic computation.

In order to demonstrate the features of the algorithm and the importance of validated tests, we show the results of numerical experiments.

Two close to degenerate case configurations of the segments were selected for the tests. In the first configuration the segments were almost *parallel* to each other (Figure 5.4(a)), and in the second they were almost *perpendicular* (see Figure 5.4(b)).

The coordinates of the endpoints were generated by a procedure that randomly selected them from the small areas that are shaded in the figure where the size of the shaded areas were selected according to a *perturbation* parameter, an integer power of 2 in the range 0 to 26, where 2^0 results in the maximum perturbation - the segments are practically random and where 2^{-26} implies practically no perturbation, i.e. the coordinates of the segment endpoints differ only in the least significant bit of the mantissa representation.

Three different algorithms were implemented and run for each of the above configurations. The number of test runs for each algorithm and each perturbation was 5000. First, the ESSA-based method was applied to compute the intersection test exactly. We do not show the results because they all were, in fact, exact. Then the direct computation algorithm was implemented for computing the intersection directly from the system given where only floating point arithmetic was used.

The number of incorrectly reported intersections was recorded for varied perturbation values for the direct method. Similarly, the direct method was executed on an interval platform (interval method). In this case the number of

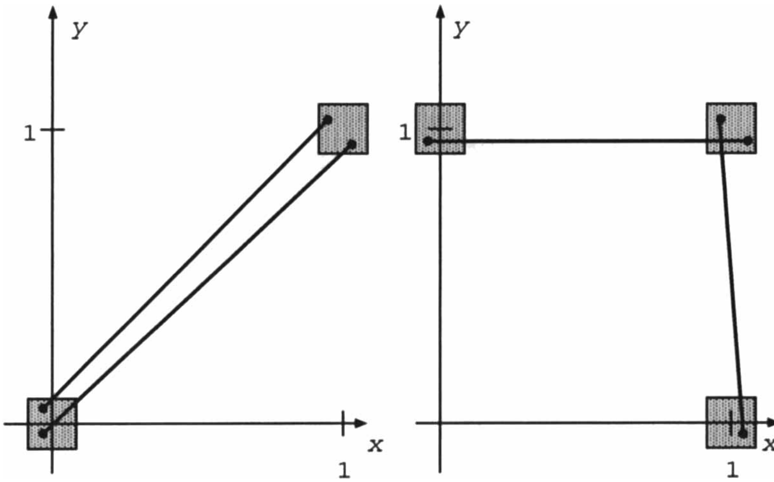


Figure 5.4: Close to degenerate configuration of segments

inconclusive results were recorded. In these cases ESSA was called in order to finally obtain a correct result.

Both the number of incorrect results and inconclusive results recorded for 5000 test runs for each perturbation value of the parallel segment configuration are presented in Table 5.1 starting with the perturbation 2^{-11} since the perturbations 2^0 to 2^{-11} resulted in no incorrect or inconclusive results.

The test results show that when the perturbation parameter is small the number of incorrect and inconclusive results is close to 1%. However, when the perturbation parameter increases above 18 the number of inconclusive and incorrect results grows exponentially, reaching the 100% mark for the perturbation parameter 24. This can be explained by the fact that with high perturbation parameters the segments are not parallel in almost 100% of the cases, but the direct algorithm reports that they are.

The second series of experiments were conducted for the perpendicular segment configuration. The test results are presented in Table 5.2. The percentage of results that are wrong for each of the two configurations are shown in Figs. 5.4 and 5.5 labeled "Direct Wrong". On the same figures the number of inconclusive interval evaluations are shown as "Interval Inconclusive".

The exponential grows in the number of inconclusive and incorrect results are noticeable for the lowest values of the perturbation parameter. The main difference in the results is that the number of incorrectly reported segment intersections reaches 50% for perturbation 2^{-26}. This is justified by the fact that the program that initially generated the perpendicular segments generated intersecting segments in 50% of the cases, but the direct intersection algorithm reports that segments intersect in all cases, thus it is wrong in a half of the

Power of 2	Perturbation	Number of incorrect results (direct method)	Number of inconclusive results (interval method)
-11	0.000488281	0	0
-12	0.000244141	1	0
-13	0.000122070	2	0
-14	6.10352E-05	1	0
-15	3.05176E-05	2	0
-16	1.52588E-05	5	0
-17	7.62939E-06	17	0
-18	3.81470E-06	32	1
-19	1.90735E-06	57	2
-20	9.53674E-07	130	12
-21	4.76837E-07	271	48
-22	2.38419E-07	499	133
-23	1.19209E-07	916	416
-24	5.96046E-08	1547	2011
-25	2.98023E-08	4349	5000
-26	1.49012E-08	5000	5000

Table 5.1: Test results for parallel segment configuration

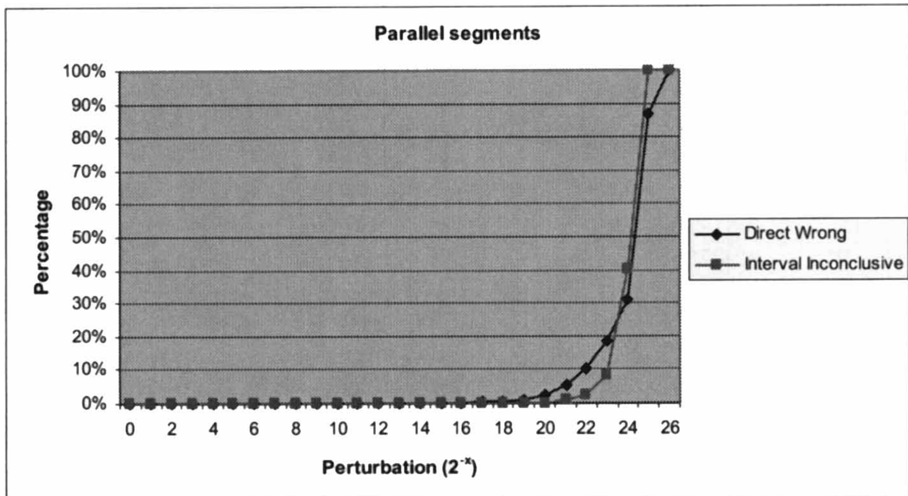


Figure 5.5: Dependence of wrong and inconclusive results on the perturbation

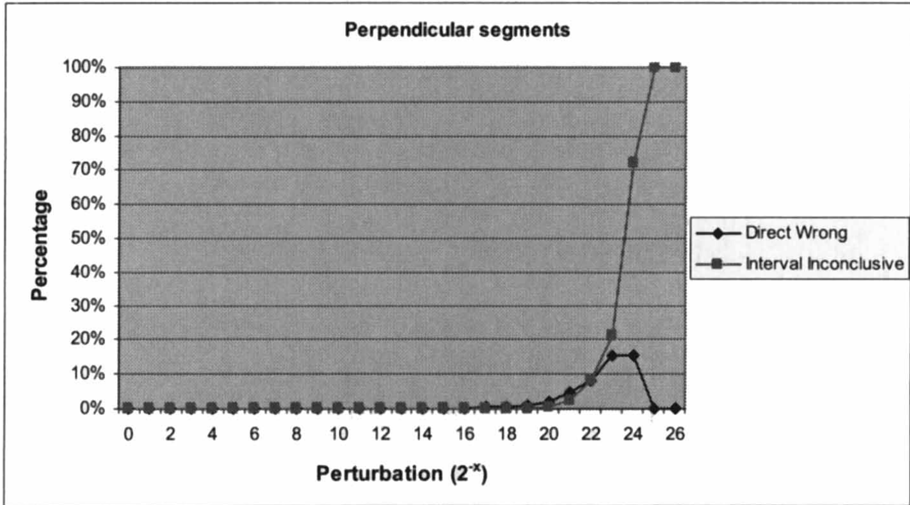


Figure 5.6: Dependence of wrong and inconclusive results on the perturbation

Power of 2	Perturbation	Number of incorrect results (direct method)	Number of inconclusive results (interval method)
-11	0.000488281	0	0
-12	0.000244141	1	0
-13	0.000122070	1	0
-14	6.10352E-05	2	0
-15	3.05176E-05	2	0
-16	1.52588E-05	8	0
-17	7.62939E-06	13	0
-18	3.81470E-06	26	3
-19	1.90735E-06	56	8
-20	9.53674E-07	107	35
-21	4.76837E-07	241	126
-22	2.38419E-07	400	402
-23	1.19209E-07	764	1065
-24	5.96046E-08	1202	3596
-25	2.98023E-08	1698	5000
-26	1.49012E-08	2600	5000

Table 5.2: Test results for perpendicular segment configuration

reported answers.

5.3 Box-Plane Intersection Testing

Considering the interactions and correlations between logic, accuracy, robustness, reliability and costs we discuss a very simple problem, in order to give an example for the specific class of computational geometry problems where selfvalidation is required and for the specific way of reasoning used to arrive at reliable solutions. That problem is to provide a reliable and reasonable Boolean-valued (and not triple-valued!) algorithm to decide whether an axes-parallel box and a plane intersect or not. We show that it is in fact possible *to avoid each kind of uncertainty and error* provided the box corners and the coefficients of the plane equation are already machine numbers[2]. Then the answer will be given for any geometrical constellation and will be completely reliable and never uncertain.

We provide three algorithms. Two of them are fairly obvious whereas the third one is not. We show that each of them has its advantages and each of them its disadvantages with respect to numerical costs, mathematical transparency and overhead. It is, in fact, surprising to see that the easiest way of implementing the test which everybody would likely use as a first trial algorithm is far from being the best among the three. Indeed, as one can see from our discussion below, one cannot say that any one of the three algorithms is the absolute best one.

We want to test whether an axes-parallel 3D box represented by $X = (X_1, X_2, X_3)$, where X_i, $i = 1, 2, 3$ are intervals, intersects a plane defined by the equation

$$f(x) = a_1x_1 + a_2x_2 + a_3x_3 + a_4 = 0$$

where the parameters, a_i, $i = 1, 2, 3, 4$ are given. It is assumed that the input data, that is, the coefficients of the plane equation and the vertices of the box, is already machine representable and that the test guarantees to provide the *correct* result for all possible configurations. The test is therefore only executed with the usual proviso that any actual real life numbers will first have been converted to machine numbers. If the input to the data for the test is, for example, the result of previous computations then the representation is already exact.

The arithmetic and logic that stand behind such a test are extremely simple, i. e.

plane and box X intersect iff there exist $x, y \in X$ such that $f(x) \leq 0 \leq f(y)$ or, equivalently,

---

[2]A note on box and plane intersections where the plane is defined by various means is given in [215].

plane and box X do not intersect, iff either $f(x) < 0$ for any $x \in X$ or $f(x) > 0$ for any $x \in X$ holds.

Various numerical implementations of this definition are possible: One could use all the points x of X in the second form of the definition (as shown in approach A below), or any special representatives of the box such as vertices or diagonals (as shown in approaches B and C, below). Independent of the choice of implementation one has to fight with and control the rounding errors. We select three representative implementations for demonstration and discussion. We make use of very few basic principle of interval arithmetic from Ch. 2.

A. Direct Interval Computation with Outward and Inward Rounding

As before the range of f over X is denoted by $f(X)$. Furthermore let

$$F(X) = a_1X_1 + a_2X_2 + a_3X_3 + a_4$$

be the natural interval extension of $f(x)$ to X. Generally, $F(X)$ cannot be represented on the computer since there are only finitely many machine numbers available. In order not to loose the logical connection between $F(X)$ and the plane equation, $F(X)$ is evaluated twice, once with outward rounding, as is commonly the case in interval computations, and once with inward rounding, which is also possible in almost all interval software packages (see also [132]). Thus we obtain two approximations of $F(X)$, that is, $F_{out}(X)$ and $F_{inw}(X)$, respectively. Note that $F_{inw}(X)$ can be the empty set. In any case,

$$F_{inw}(X) \subseteq F(X) \subseteq F_{out}(X).$$

Theorem 3 says that the natural interval extension of a multivariate function gives the range if each of the variables occurs at most once and of power one. One recognizes that this theorem is applicable to f, and we get

$$f(X) = F(X).$$

Hence, we have machine representable outer and inner approximations of the range, $f(X)$. That is,

$$F_{inw}(X) \subseteq f(X) \subseteq F_{out}(X).$$

This inclusion chain renders the following validated result:

(i) If $0 \in F_{inw}(X)$, the box and the plane intersect,

(ii) if $0 \notin F_{out}(X)$, box and plane do not intersect.

(iii) In the remaining cases, which can be summarized as

$$0 \in F_{out}(X) \setminus F_{inw}(X)$$

the computation up to this stage was not effective or precise enough to allow a decision.

Generally, the two intervals $F_{out}(X)$ and $F_{inw}(X)$ will have almost the same size so that case (iii) will only occur very infrequently. Nevertheless, one also has to provide a way to get a reliable test result in this case. We will return to this issue later in point D.

B. Direct Interval Computation of the Vertices of the Box

The numerical procedure will in this case follow the fact that box and plane do not intersect iff the eight vertices of the box lie in only one of the two open half spaces defined by the plane. Let

$$v_1, ..., v_8$$

be the eight vertices of the box X. Then box and plane intersect iff two vertices v_i and v_j exist that satisfy

$$f(v_i) \leq 0 \leq f(v_j).$$

Let $F(v_i)$ be the machine interval arithmetic computation of $f(v_i)$ with regular outward rounding for $i = 1, ..., 8$. Then one gets the following result:

(i) If

$$F(v_i) \leq 0 \leq F(v_j)$$

is satisfied for some indices $i, j \in \{1, ..., 8\}$ then the box and the plane intersect.

(ii) If $F(v_i) < 0$ for any $i = 1, ..., 8$ or
if $F(v_i) > 0$ for any $i = 1, ..., 8$
then the box and the plane do not intersect.

(iii) In the remaining cases, the computation up to this stage was not effective enough to allow a decision.

As in implementation A, one needs a way to cover the constellations related to this case (iii). This will be settled in point D.

C. Testing Only a Representative Main Diagonal

The numerical procedure in this implementation is based on the very surprising and almost unknown fact that *the box intersects the plane iff the plane intersects that main diagonal of the box that has the smallest unoriented angle to the normal of the plane.*

In order to use this fact, the box vertices are labeled as in Fig. 5.7. Let then the main diagonals $D_1, ..., D_4$ of the box be defined in such a manner that

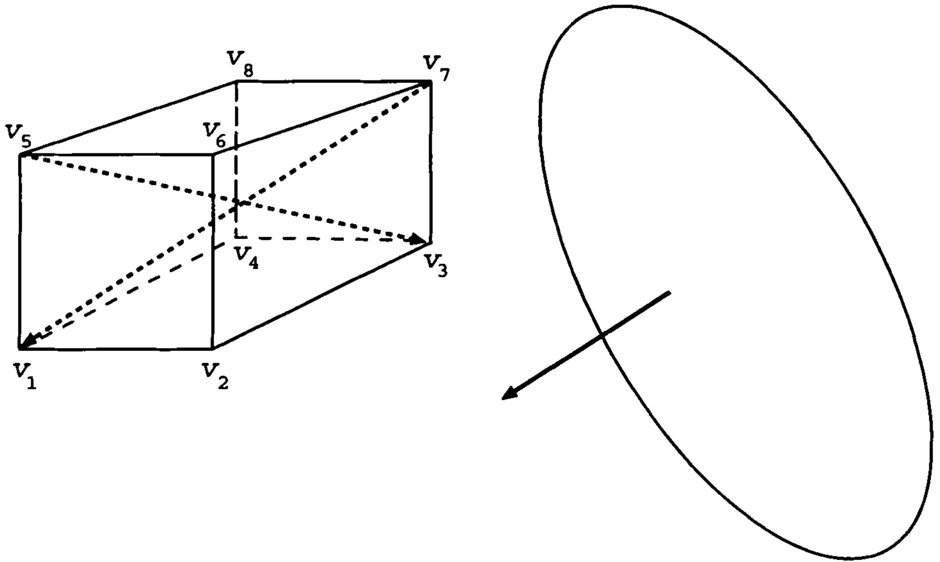


Figure 5.7: Box-plane intersection

D_1 connects v_1 with v_7,
D_2 connects v_2 with v_8,
D_3 connects v_3 with v_5,
D_4 connects v_4 with v_6.

Since we have to deal with orientation, these diagonals are represented as vectors which are also denoted by $D_1, ..., D_4$:

$D_1 = v_7 - v_1$,
$D_2 = v_8 - v_2$,
$D_3 = v_5 - v_3$,
$D_4 = v_6 - v_4$.

As one can see, the numerical values of the coordinates of the vectors are not essential for the characterization of the diagonals, but only the sign of the coordinates, cf. Fig. 5.7. Thus the diagonals are uniquely defined by the sign distribution of their coordinates,

$D_1 \sim (+\ +\ +)$,
$D_2 \sim (-\ +\ +)$,
$D_3 \sim (-\ -\ +)$,
$D_4 \sim (+\ -\ +)$.

If degeneracies are admitted, that is, the box is no longer solid, the characterization is still valid, if $+$ stands for a nonnegative real. Then some diagonals will coincide. They are still uniquely defined by the signs, but be aware, their numbering is not.

Let now $n = (n_1\ n_2\ n_3)$ be a vector and $n_3 \geq 0$. Let further D_0 be a main diagonal which has the sign distribution $(sgn(n_1)\ sgn(n_2)\ +)$ where $sgn(0)$ is assigned $+$ as in the degenerate case. Then D_0 is that main diagonal which has the smallest unoriented angle among $D_1, ..., D_4$ to n.

Crucial for our test is the following

Proposition. *The box diagonal with the smallest unoriented angle to the plane normal intersects the plane iff the box intersects the plane.*

Proof. (i) It is obvious that box and plane intersect if one of the diagonals intersect the plane since the diagonals are part of the box.

(ii) Now assume that the box and plane intersect and further assume that no box edge is parallel to the plane. Then there exist vertices v' and v'' so that v' is farthest from the plane on one side and v'' is farthest from the plane on the other side.

Next we prove that v' and v'' are the endpoints of a main diagonal:

Suppose v' and v'' are on the same edge. Then there exist at least one v''' on an edge joining v' (or on an edge joining v'') that is farther away from the plane than v' (or v''). Hence v' and v'' cannot be on the same edge.

Similarly, they cannot be on the same face. Hence v' and v'' must form the endpoints of a box diagonal.

It is now easy to see that the diagonal formed by v' and v'' has the smallest unoriented angle with the plane normal:

The projection of this diagonal onto the plane normal must be the largest projection of all the diagonals since the endpoints of the diagonal are furthest away from the plane on either side. Since all of the diagonals have the same length the diagonal with the largest projection must form the smallest angle by the definition of the cosine of the angle. Since v' and v'' are on different sides of the plane (degeneracies included) the connecting diagonal intersects the box.

If one box edge is parallel to the plane, cf. the technical assumption at the begin of (ii), one proceeds analogously to construct an appropriate main diagonal, but it is no longer uniquely defined. □

We return to our concrete geometric situation. The plane was given by the equation

$$f(x) = a_1x_1 + a_2x_2 + a_3x_3 + a_4 = 0.$$

Let for a moment $a_3 \geq 0$. Then, by the normal form of the plane, the normal vector of the plane is, up to the length, defined by

$$(a_1 \ a_2 \ a_3).$$

Hence the main diagonal which is required for the test is determined as follows:

Let the coefficients of the plane equation, $a_1, ..., a_4$, be given (again without any restrictions). Then we obtain the following procedure:

1. Determine a main diagonal D_0 which has the sign distribution

$$(sgn(a_1) \ sgn(a_2) \ +) \quad \text{if } sgn(a_3) \geq 0,$$

or the sign distribution

$$(-sgn(a_1) \ - sgn(a_2) \ +) \quad \text{if } sgn(a_3) < 0.$$

2. Let v_i and v_j be the endpoints of D_0. Evaluate $F(v_i)$ and $F(v_j)$ as in B. Then one gets the following result:

(i) If

$$F(v_i) \leq 0 \leq F(v_j)$$

or

$$F(v_j) \leq 0 \leq F(v_i)$$

is satisfied, the plane and D_0, hence the plane and the box intersect.

(ii) If $F(v_i), F(v_j) > 0$ or if $F(v_i), F(v_j) < 0$

the plane and D_0, hence the plane and the box do not intersect.

(iii) In the remaining cases, the computation up to this stage was not effective or precise enough to enforce a decision. This will be settled in point D.

D. Procedure for the Remaining Undecidable Cases

In each of the three implementations there remained a percentage of constellations which could not be decided. In order to complete the tests one has to apply an *exactly* working algorithm for comparisons of sums, or which is the same, for determining the sign of a sum since all the undetermined cases can be brought to this form. An algorithm which meets these requirements is ESSA. Therefore, each of the implementations A, B, or C together with ESSA is able to give a *complete and reliable* answer to the question whether the box and the plane intersect.

5.3.1 Which of the 3 Approaches is the Best?

After having described three implementation samples, *one question is still open, that is the question which of them is best and can be recommended.* This is not easy to say and depends mainly on the expectations of the user.

Implementation A is the simplest one and it is very easy to handle. It could be a favorite implementation on the average since, statistically, it will have the most favorable computation time. However, if the undetermined case (iii) in A is addressed, i. e.

$$0 \notin F_{inw}(X), \; 0 \in F_{out}(X)$$

then the former computational information is completely worthless and cannot be used for a supplementary final correct computation. The reason is that if (iii) occurs, "most" of the corners are on the one side of the plane and the remaining "few" are so near the plane that the application of A cannot figure out definitively on which side of the plane they are. So the corners have to be checked again with ESSA. The spectrum ranges from 1 till all 8 corners, so that worst case analysts will never use implementation A.

The code for implementation B is as simple as the code for A. The average costs, however, are higher than at A, since one has between 2 and 8 corner evaluations already in the definitive case without ESSA. But if the undetermined case (iii) in B happens, one already knows the critical corners and one only has to process these with ESSA. This shows that the worst case analysis is already better than the one for A.

The code for the implementation C is already more difficult than the previous code, and the mathematical background is charming but not too easy to understand. The average computational costs, however, are lower than in case B and comparable with the costs in case A. The worst case analysis for C is better than for case A and case B.

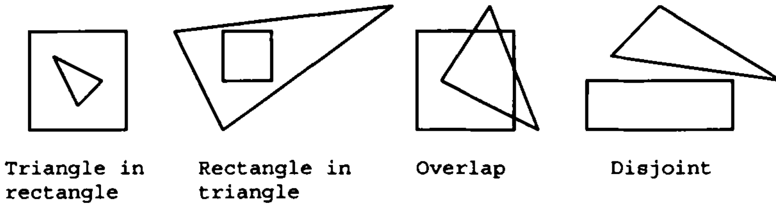


Figure 5.8: The four possible rectangle-triangle configurations

5.4 Rectangle-Triangle Intersection Testing

One of the primitive operations that occur in many geometric computations is to establish the relationship between a rectangle which can be assumed to be axis-parallel and an arbitrary triangle both lying in the same plane. Typically this operation might occur as part of a subdivision process or as part of the interrogation of a map.

In order to discuss the possible cases we first define the term *overlap* for two sets, or geometric objects A and B:

$$A \text{ overlaps } B \text{ iff } A \cap B \neq \emptyset \text{ and neither } A \subseteq B \text{ nor } B \subseteq A.$$

That is, A is not contained in B, B is not contained in A but A and B intersect.

The aim of this section, based on [219], is to develop two procedures for recognizing the possible rectangle-triangle relationships:

1. containment of the triangle in the rectangle,
2. containment of the rectangle in the triangle,
3. the triangle and the rectangle overlap,
4. the triangle and the rectangle do not intersect.

These four cases are illustrated in Figure 5.8.

In the first test we apply Skelboe's principle to determine the set of barycentric coordinates of all points of the rectangle in concise form. Hence we get something like ***interval barycentric coordinates*** being defined in Subsec. 5.4.1. These were already used in [215] to develop a plane-box intersection test.

Hence the main feature of the first test is

> the use of interval barycentric coordinates which are the collection of the barycentric coordinates of all points in the rectangle. This computation is almost as simple as the computation of barycentric coordinates of points.

The second test is a non-interval test. It is based on a direct checking of the relationships between the edges of the rectangle and the triangle. This might appear to be fairly complicated, however, it turns out to be reasonably simple: By introducing the idea of rectangle edges that are "visible' from a triangle vertex and then using the slopes of the triangle edges it is only necessary to test the relationship of one triangle vertex with one or two of the visible rectangle edges when the triangle and the rectangle are tested for intersection. The numerical cost is a bit higher if they are checked for overlapping.

All together, both versions of this approach need only a few arithmetic and logical operations. The drawback is that this (second) test is logically rather involved so that a computer implementation has to incorporate many branches that need to be distinguished.

5.4.1 Interval Barycentric Coordinates

The non-degenerate triangle we want to relate with the rectangle is defined by three vertices $r, s, t \in R^2$ and it is denoted by T in the sequel.

It is well known that the barycentric coordinates of a point $q \in R^2$ w.r.t. T can be computed as

$$\gamma_1(q) = \frac{\text{area}(q,s,t)}{\text{area}(r,s,t)}, \quad \gamma_2(q) = \frac{\text{area}(r,q,t)}{\text{area}(r,s,t)}, \quad \gamma_3(q) = \frac{\text{area}(r,s,q)}{\text{area}(r,s,t)}$$

where

$$\text{area}(r,s,t) = \frac{1}{2}\begin{vmatrix} r_1 & s_1 & t_1 \\ r_2 & s_2 & t_2 \\ 1 & 1 & 1 \end{vmatrix}$$

(see [252, 46]). Although this definition is complete, we note that area(q, s, t) is the area of the oriented triangle with vertices q, s, t. Orientation means that the boundary curve of this triangle passes q, s, t in this order.

Among the various properties of barycentric coordinates we mainly need the following two:

$$\gamma_1(q) + \gamma_2(q) + \gamma_3(q) = 1 \text{ for } q \in R^2, \tag{5.8}$$

$$q \text{ is a point of the triangle iff } 0 \le \gamma_i(q) \le 1 \text{ for } i = 1,2,3. \tag{5.9}$$

We now note that for each i the barycentric coordinates for points q with respect to T partitions the plane into three regions by two parallel lines, one passing through a side of T and the other passing through the opposite vertex. This partitioning results in:

1. a region where q is restricted by $\gamma_i(q) < 0$,
2. a region where q is restricted by $\gamma_i(q) \in [0, 1]$,

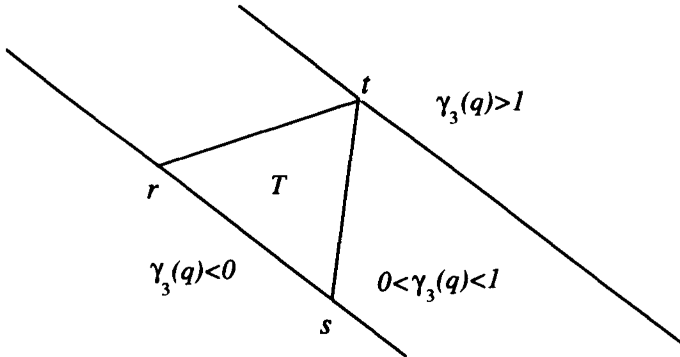


Figure 5.9: Partitioning by one of the barycentric coordinates

3. and a region where q is restricted by $\gamma_i(q) > 1$.

An example of this partition with $i = 3$ is shown in Figure 5.9. The intersection of the three regions where q is restricted by $\gamma_i(q) \in [0,1]$, $i = 1,2,3$ defines the triangle, i.e. $\{q : \gamma_i(q) \in [0,1], i = 1,2,3\} = T$.

Consider now the axis-parallel rectangle $D = D_1 \times D_2$. Using the notation for ranges, $\square\gamma_i(D) = \{\gamma_i(q) : q \in D\}$, $i = 1,2,3$ it follows that

$$D \subseteq T \text{ iff } 0 \leq \square\gamma_i(D) \leq 1, \quad i = 1,2,3. \tag{5.10}$$

It turns out that by the principle of Skelboe the natural interval extension $\gamma_i(D)$, $i = 1,2,3$ compute the range provided the determinants involved are expanded so that the interval variables only occur once and to the first power. This means that

$$\gamma_1(D) = \frac{\text{area}(D,s,t)}{\text{area}(r,s,t)}, \quad \gamma_2(D) = \frac{\text{area}(r,D,t)}{\text{area}(r,s,t)}, \quad \gamma_3(D) = \frac{\text{area}(r,s,D)}{\text{area}(r,s,t)}$$

computes the exact range provided the determinants are expanded in the following manner ($i = 1$ is chosen as an example):

$$\begin{aligned} \gamma_1(D) &= \frac{\text{area}(D,s,t)}{\text{area}(r,s,t)} = \frac{1}{2\text{area}(r,s,t)} \begin{vmatrix} D_1 & s_1 & t_1 \\ D_2 & s_2 & t_2 \\ 1 & 1 & 1 \end{vmatrix} \\ &= \frac{(D_1 - t_1)(s_2 - t_2) - (D_2 - t_2)(s_1 - t_1)}{2\text{area}(r,s,t)}. \end{aligned} \tag{5.11}$$

We note that each interval variable only occurs once in this expansion which means that Skelboe's principle holds. The interval evaluation (5.11) therefore computes the range and we can replace (5.10) by

$$D \subseteq T \text{ iff } 0 \leq \gamma_i(D) \leq 1, \quad i = 1,2,3. \tag{5.12}$$

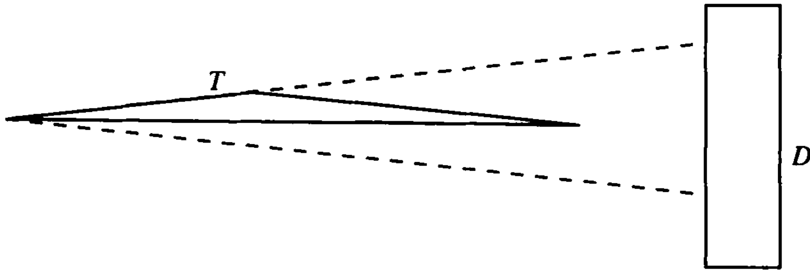


Figure 5.10: A counterexample to intuition

provided all the $\gamma_i(D)$, $i = 1, 2, 3$ are evaluated in the same manner as (5.11).

Because of (5.8), criterion (5.12) is equivalent to

$$D \subseteq T \text{ iff } 0 \leq \gamma_i(D), \quad i = 1, 2, 3. \tag{5.13}$$

One notes that relationship (5.12) is already appropriate as a criterion for the case $T \subseteq D$ and it will thus be incorporated in the complete test in the next subsection. Another criterion which is immediately obvious due to the definition of interval barycentric coordinates, is based on the partition of the plane into regions, cf. Fig. 5.9 and it says that the rectangle and the triangle do not intersect, if the rectangle is contained in one of the halfplanes defined by

$$\gamma_i(q) < 0 \text{ or } \gamma_i(q) > 1 \text{ for some } i = 1, 2, 3.$$

That is,

$$\text{if } \gamma_i(D) < 0 \text{ or } \gamma_i(D) > 1 \text{ for some } i = 1, 2, 3 \text{ then } D \cap T = \emptyset. \tag{5.14}$$

The converse of this conclusion is, however, not true. Hence (5.14) will be completed in the next section when more geometric insight is obtained.

5.4.2 The Geometry of Test 1

In the previous subsection we noted that the interval evaluation of the barycentric coordinates enabled us to make a definite decision for intersection or disjointedness for some of the triangle-rectangle configurations. The remaining cases which have not been covered by the simple constellations (5.12) and (5.14) need some further contemplation and the incorporation of geometric aspects. One even has to be careful not to succumb to any "obvious" geometric insights. For example, one could easily be attempted to conclude that D and T intersect if $\gamma_i(D) \supseteq [0, 1]$ for $i = 1, 2, 3$. However, there are examples that show that this conclusion is wrong, cf. Fig. 5.10. The rectangle in the figure only has to touch the two dotted lines to satisfy those assumptions.

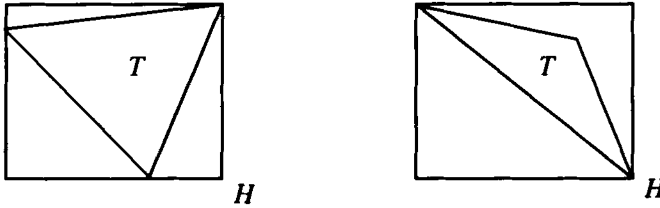


Figure 5.11: The bounding rectangle possibilities

In order to proceed we introduce the axis-parallel bounding rectangle $H = H(T)$ of T, that is, the isothetic rectangle hull of T (being the smallest axis-parallel rectangle that contains T) as shown in Fig. 5.11.

In the first part of the figure the bounding rectangle is generated by the three vertices of T and in the second part only by two (i.e. one of the vertices can be moved a bit in a certain direction without changing H).

From the definition of H another partial test follows immediately:

$$\text{If } D \cap H = \emptyset \text{ then } D \text{ and } T \text{ are disjoint.} \tag{5.15}$$

This means that any rectangle D lying outside H forms an empty intersection with the triangle T. We only have to concern ourselves with the rectangles D that intersect H.

For this reason we introduce the intersection

$$D_H = D \cap H$$

and we use it for the further development of the complete test. One notes that the triangle T has almost the same relationship to D_H as to D. That is, D intersects [is contained in, contains, is disjoint with] T if and only if D_H intersects [is contained in, contains, is disjoint with] T. One only has to be cautious with overlapping which can change to a degenerate situation when D_H is just a straight line or a point.

Hence, (5.12), (5.14) and (5.15) remain valid if in the right-hand side of (5.12) as well as in the assumptions of (5.14) and (5.15), D is replaced with D_H.

The advantage of dealing with D_H instead of D is twofold:

(i) One can execute the investigation of the various cases within H which reduces the number of cases

(ii) one can get an earlier decision during the algorithmic execution of the test.

As an example for (ii) assume that $\gamma_1(D) < 0$ does not hold when executing (5.14). In spite of this $\gamma_1(D_H) < 0$ may hold so that we have the immediate

result that $T \cap D_H = \emptyset$ which means that $T \cap D = \emptyset$ thus terminating the query.

Test 1 *for the relationship between the axis-parallel rectangle D and the nondegenerate triangle T.*

Let H be the axis-parallel rectangle hull of T, let $D_H = D \cap H$ and $\gamma_i(D_H)$ be the interval barycentric coordinates of D_H w.r.t. T. Then

A1. If $D_H = \emptyset$ then D and T are disjoint.

A2. If $H = D_H$ then T is contained in D.

A3. If $\gamma_i(D_H) < 0$ or $\gamma_i(D_H) > 1$ for some $i = 1, 2, 3$ then D and T are disjoint.

A4. If $0 \leq \gamma_i(D_H) \leq 1$ for $i = 1, 2, 3$ then

(i) D is contained in T, if $D = D_H$

(ii) D and T overlap if $D \neq D_H$.

A5. In the remaining cases, D and T overlap.

A test is called complete if it considers all constellations which can occur as subject of the test. A test is called correct of consistent if no wrong result is rendered. One notes that *Test 1 is complete* (due to the passage "In the remaining cases..." in step A5). It remains to prove that the test is *correct*, which will be done in the remainder of this section. Since the correctness of A1 to A4 is obvious and connected with (5.12), (5.14), (5.15) and the transitions from H to D_H, only the proof of A5 is required.

A few of the conclusions of the test are obviously reversible:

A2'. If T is contained in D then $H = D_H$.

A4'. If $0 \leq \gamma_i(D_H) \leq 1$ for $i = 1, 2, 3$ then

(i) if D is contained in T then $D = D_H$

(ii) if D and T overlap then $D \neq D_H$.

It suffices to prove A5 in a weaker version, namely

A5'. In the remaining cases, D and T intersect.

The reason is that for the remaining cases no inclusion relation can occur: $T \subseteq D$ would imply $H = D_H$ by A2' which is already subsumed by the test step A2, and $D \subseteq T$ would imply $0 \leq \gamma_i(D) \leq 1$ and $0 \leq \gamma_i(D_H) \leq 1$ for $i = 1, 2, 3$ then we would get $D = D_H$ by A4' which is already subsumed by test step A4(i).

We first cite two lemmas which certainly could be proven by drawing sample figures. Our preference is, however, for analytic proofs.

LEMMA 2 *If $\gamma_i(D_H) > 0$, $\gamma_j(D_H) > 0$ and $0 \in \gamma_k(D_H)$ where $\{i,j,k\} = \{1,2,3\}$ then T and D_H intersect.*

Proof. Since $0 \in \gamma_k(D_H)$ there exist an $x \in D_H$ with $\gamma_k(x) \geq 0$. The remaining assumptions imply $\gamma_i(x) > 0$ and $\gamma_j(x) > 0$. Since $\sum_{\nu=1}^{3} \gamma_\nu(x) = 1$ holds after (5.8), we get

$$0 \leq \gamma_\nu(x) \leq 1 \text{ for } \nu = 1,2,3.$$

This means that $x \in T$ by (5.9). Hence $x \in D_H \cap T$ and $D_H \cap T \neq \emptyset$. □

LEMMA 3 *If, for $\nu = 1,2,3$,*

$$0 \in \gamma_\nu(D_H) \text{ or } 0 < \gamma_\nu(D_H)$$

and if D_H and T do not intersect, there exist at least two different indices i, j with

$$0 \in \gamma_i(D_H) \text{ and } 0 \in \gamma_j(D_H).$$

Proof. In order to get a contradiction we assume that at most one of the lines $\gamma_\nu(x) = 0$ meets D_H.

If exactly one of the lines meets D_H, Lemma 2 says that D_H and T intersect. This is a contradiction.

If none of the lines meets D_H then, by assumption, $0 < \gamma_\nu(D_H)$ holds for side condition (5.8) of the barycentric coordinates, $\sum_{\nu=1}^{3} \gamma_\nu(x) = 1$ for any $x \in R^2$, and the fact that $\gamma_\nu(D_H) = \square\gamma_\nu(D_H)$ for $\nu = 1,2,3$ we get $\gamma_\nu < 1$ for $\nu = 1,2,3$. This means that $\gamma_\nu(D_H) \subseteq [0,1]$ for $\nu = 1,2,3$ and it follows that D_H and T do not intersect. □

THEOREM 9 *Test step A5 is correct.*

Proof. We mentioned already that it is sufficient to prove the correctness of A5' instead of A5.

We assume that D and T obey "a remaining case" as expressed with A5', which means that D and T do not meet the assumptions of A1 to A4. Focusing on A3 and A4 and using (5.13) says that there does not exist an index $i = 1,2,3$ with

$$\gamma_i(D_H) < 0 \text{ or } \gamma_i(D_H) > 1, \tag{5.16}$$

but that there exist an index $k = 1,2,3$ such that

$$0 \leq \gamma_k(D_H)$$

does not hold. These two conditions together imply that

$$0 \in \gamma_k(D_H) \text{ for some } k = 1, 2, 3. \tag{5.17}$$

We now have to distinguish two cases.

1. H is generated by all three vertices of T (lefthand side of Fig. 5.11). In this case, the vertices of T lie on the boundary of H, and those parts of the lines $\gamma_i(x) = 0$ that lie in H are parts of the edges of T, i.e.

$$\text{if } x \in H \text{ and } \gamma_i(x) = 0 \text{ then } x \in T \text{ for } i = 1, 2, 3. \tag{5.18}$$

By (5.17), there exists $x \in D_H \subseteq H$ with $\gamma_k(x) = 0$. By (5.18), $x \in T$ is implied, and D_H and T intersect, accordingly D and T intersect, which was to be proven.

2. H is generated by two of the vertices of T (cf. right side of Fig. 5.11). That means the third vertex can be dropped or slightly moved in a certain direction without changing H. In order to get a contradiction, it is assumed that D_H and T (and accordingly D and T) do not intersect. This assumption together with (5.16) and (5.17) implies that the assumptions of Lemma 3 hold. Therefore, an index $l \neq k$ with $0 \in \gamma_l(D_H)$ exists. Without restricting the generality, we set $k = 1$, $l = 2$ such that

$$\begin{array}{lcl} 0 \in \gamma_1(D_H), & & 0 \in \gamma_2(D_H), \\ 0 \in \gamma_3(D_H), & \text{or} & 0 < \gamma_3(D_H) \end{array}$$

holds. There exist points $p, q \in D_H$ with $\gamma_1(p) = \gamma_2(q) = 0$. By assumption, $p, q \notin T$. Hence $p \in D_H \subseteq H$ lies on the line $\gamma_1(x) = 0$ and $q \in D_H \subseteq H$ on the line $\gamma_2(x) = 0$. This is only then possible if the vertex t of T being the cutting point of the two lines $\gamma_1(x) = 0$, $\gamma_2(x) = 0$ is an interior point of H,

Therefore, r and s are the vertices of T that generate H, they connect opposite corners of H, and the line segment from r to s is a diagonal of H. Without restricting the generality we assume that r is the left lower corner of H, that s is the right upper corner of H, and that t lies above the related diagonal, as shown in Fig. 5.12. It follows from this configuration that the lines $\gamma_1(x) = 0$ and $\gamma_2(x) = 0$ increase monotonically, that is, p is left of and below t, and t is left of and below q. This gives, expressed analytically,

$$p_1 \leq t_1 \leq q_1 \text{ and } p_2 \leq t_2 \leq q_2.$$

One notes that the axis-parallel rectangle hull of p and q, denoted by $H(p,q) = \{x \in R^2 : p_i \leq x_i \leq q_i, i = 1, 2\}$ and that each axis-parallel rectangle B contains $H(p,q)$ where $p, q \in B$. Hence, $t \in H(p,q)$. Since $p, q \in D_H$ where D_H is an axis-parallel rectangle, it follows that $H(p,q) \subseteq D_H$. This yields $t \in D_H \cap T$ obtaining a contradiction by $D_H \cap T \neq 0$. □

Figure 5.12: The geometry of the proof

5.4.3 An Algorithm for Test 1

The collection of steps A1 - A5 is certainly correct and forms a complete test and can immediately be chosen as the set of steps for an algorithm and for a numerical execution.

It is, however, reasonable to add a few simple comparisons and to change the order of the steps slightly in order to set the cheaper computational steps firstly and to compute the (more expensive) interval barycentric coordinates secondly. We accept hereby a small percentage of overlapping of conclusions hoping that the early steps will lead to a decision so that the algorithm can be terminated.

Let again $r, s, t \in R^2$ be the affinely independent vertices of the triangle T with coordinate representation $r = (r_1, r_2)$, etc. Let $D = D_1 \times D_2$ be the rectangle with $D_1 = [c_1, d_1]$ and $D_2 = [c_2, d_2]$. The four corners of the rectangle are (c_1, c_2), (d_1, c_2), (d_1, d_2) and (c_1, d_2) in counterclockwise order.

Further let

$$\begin{aligned} a_i &= \min\{r_i, s_i, t_i\}, i = 1, 2, \\ b_i &= \max\{r_i, s_i, t_i\}, i = 1, 2. \end{aligned}$$

The rectangle hull $H = H(T)$ is then given by $H = [a_1, b_1] \times [a_2, b_2]$, and the four corners of H are (a_1, a_2), (b_1, a_2), (b_1, b_2), and (a_1, b_2) in counterclockwise

order. Hence H can be seen as generated by $a = (a_1, a_2)$ and $b = (b_1, b_2)$ as well, $H = H(a, b)$.

With these preparations the algorithmic version of the test is established.

ALGORITHM 11 *(For testing the relationship between the axis-parallel rectangle D and the nondegenerate triangle T.)*

Step 1. *If $b_1 < c_1$ or $b_2 < c_2$ or $d_1 < a_1$ or $d_2 < a_2$ (that is, D and H are disjoint) then D and T are disjoint.*

Step 2. *If $c_i \leq a_i$ and $b_i \leq d_i$ for $i = 1, 2$ (that is, $H \subseteq D$) then T is contained in D.*

Step 3. *Compute $D_H = D \cap H$.*
(If $k_i = \max\{a_i, c_i\}$, $l_i = \min\{b_i, d_i\}$ for $= 1, 2$ then $D_H = [k_1, l_1] \times [k_2, l_2]$.)

Step 4. *If $r \in D$, $s \notin D$ or $t \notin D$ or if $s \in D$, $t \notin D$ or $r \notin D$ or if $t \in D$, $r \notin D$ or $s \notin D$ (that is, one vertex of T lies in D and one outside D) then*

(i) D is contained in T if $D = D_H$,

(ii) D and T overlap if $D \neq D_H$.

Step 5. *Compute the interval barycentric coordinates $\gamma_i(D_H) = \square\gamma_i(D_H)$, $i = 1, 2, 3$ as explained in Subsec. 5.4.1.*

Step 6. *If $\gamma_i(D_H) < 0$ or $\gamma_i(D_H) > 1$ for some $i = 1, 2, 3$ then D and T are disjoint.*

Step 7. *If $0 \leq \gamma_i(D_H) \leq 1$ for $i = 1, 2, 3$ then*

(i) D is contained in T if $D = D_H$,

(ii) D and T overlap if $D \neq D_H$.

Step 8. *In all remaining cases, D and T overlap.*

The advantage of this algorithm is that it is transparent and the disadvantage is that one can develop algorithms requiring fewer operations. In a worst case analysis we counted 13 subtractions, 7 of them with intervals, 6 products with intervals, 3 divisions with intervals, and 36 real comparisons. Equality checking was not counted. The average number of operations will always be lower, however.

The experienced programmer will certainly not program Steps 1 to 3 one after the other as shown above. One can reduce the computation time if these

steps are combined and reformulated by a tree structure where the branching depends on the relations of the assumptions of the Steps 1-3.

If the computations for the algorithm are executed on a computer, there is a very small percentage that the results will be falsified by rounding errors. A good means for controlling and overcoming such problems is machine interval arithmetic, cf. Sec. 2.4. This does not add substantially to the cost of the algorithm since interval calculations already form the basis for the algorithm.

One could imagine that it is a simple matter to lift the considerations of this section by dimension 1 in order to obtain a test or an algorithm for the relationship of a box and a tetrahedron. This is, however, not the case for topological reason: The difference area $H\backslash T$ consists in general, of two or three connected components. In dimension 3 the difference area $\tilde{H}\backslash\tilde{T}$ (where $\tilde{T}$ is a tetrahedron and $\tilde{H} = \tilde{H}(\tilde{T})$ the axis-parallel hull of $\tilde{T}$) will only consist of one component which involves at least one interior hollow leading to a number of different cases caused by a variety of sites which cannot occur in two dimensions. Hence, a test for the relationship between a box and a tetrahedron will appear in a separate section.

5.4.4 Numerical Examples Using Test 1

As an example we consider the situation in Figure 5.12 where a triangle T defined by the vertices $r = (0,1)$, $s = (4,1)$ and $t = (6,6)$ and three rectangles D, D' and D'' are shown.

1. Clearly the vertices of T are not in D. The bounding rectangle $H = [0,6] \times [1,6]$ is therefore computed. Since $D_H = H \cap D \neq \emptyset$ we therefore compute $\gamma_1(D_H) = [0.5, 1.15]$, $\gamma_2(D_H) = [-0.95, 0.5]$ and $\gamma_3(D_H) = [0.2, 0.8]$. Since Steps 6 and 7 do not hold we have that the rectangle D overlaps the triangle T (found in Step 8).

2. The vertices of T are not in D'. The bounding rectangle $H = [0,6] \times [1,6]$ is therefore computed. Since $D'_H = D' \cap H \neq \emptyset$ we therefore compute $\gamma_1(D'_H) = [0.1, 0.45]$, $\gamma_2(D'_H) = [0.15, 0.7]$ and $\gamma_3(D'_H) = [0.2, 0.4]$. Since $\gamma_i(D'_H) \subseteq [0,1]$, $i = 1,2,3$ holds and $D' = H'_H$ it implies that the box D' is completely contained in the triangle (found in Step 7).

3. For the rectangle $D'' = [5,6] \times [1,2]$ we have that the vertices of T are not in D''. We again compute the bounding rectangle $H = [0,6] \times [1,6]$. Since $D''_H = D'' \cap H \neq \emptyset$ we compute $\gamma_1(D''_H) = [-0.5, -0.15]$, $\gamma_2(D''_H) = [0.95, 1.5]$ and $\gamma_3(D''_H) = [0.0, 0.2]$. Since $\gamma_1(D''_H) < 0$ it follows that rectangle D'' is disjoint from T (found in Step 6).

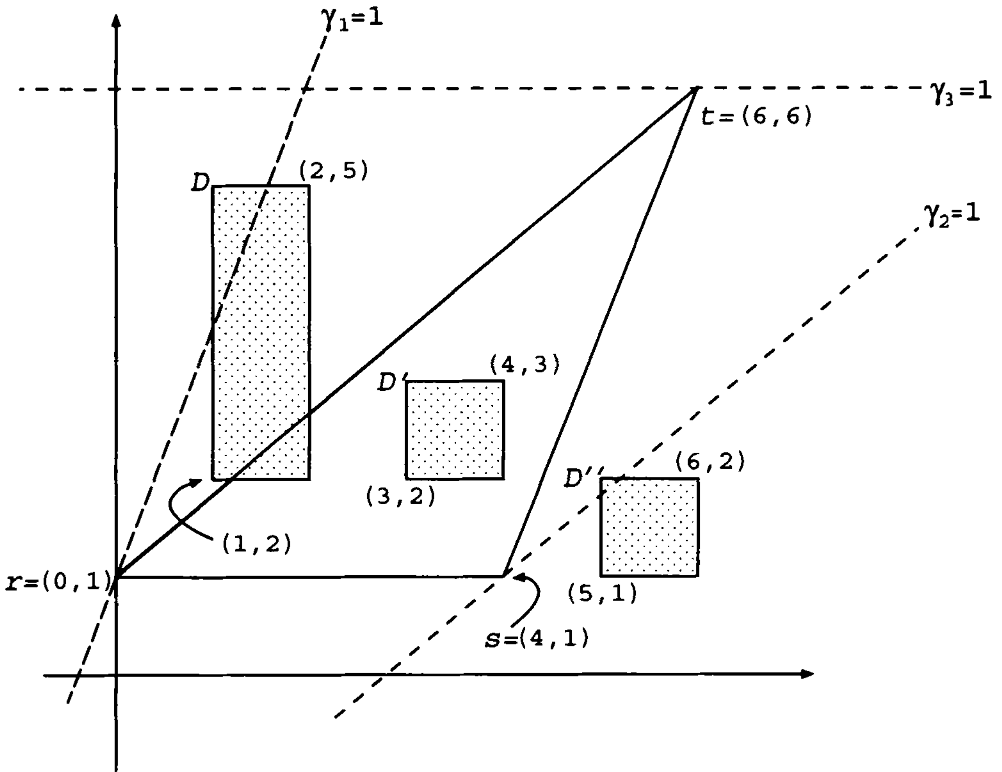


Figure 5.13: Rectangle triangle intersection examples

5.4.5 An Alternative Test

Test 1 has a very simple logical and geometric structure. Even so, the correctness of the test is not at all evident. Another feature of the test is that almost no logical or computational effort is invested in order to distinguish between the cases of overlapping and intersection of T and D. (Hence no distinction was made.) The price for this convenience is paid in numerical effort. We therefore provide an alternative test to Test 1 which has diametrically opposed properties: The logical structure is much more complex, but the correctness of the test is evident and does not require a proof. The number of arithmetic and logical operations is also lower in the worst case analysis. Some logical and computational differences also arise if overlapping and intersection is distinguished.

We use the notation of the previous sections, i.e., we use the rectangle hull $H = H(T)$ of the triangle T and the intersection $D_H = D \cap H$ between the rectangle D and the hull H.

Then the second algorithm for testing the relationship between the axis-parallel rectangle D and the nondegenerate triangle T is as follows (where the steps of the algorithm and the accompanying explanations are merged in order to improve the understanding of the algorithm):

ALGORITHM 12 *(Second rectangle- triangle test)*

First **Steps 1** *to* **4** *of Alg. 11 are executed. If they have not led to a decision then the remaining situation is that*

α) *D and H are not disjoint ($D_H \neq 0$),*

β) *T is not contained in D,*

γ) *all vertices of T lie outside D and thus outside D_H.*

In order to reduce the number of cases as far as possible the following Steps 5 and 6 are used to normalize the geometric constellation.

Step 5. *Find a vertex of T which is also a corner of H. (There exist at least one such vertex.) Let this vertex be the left lower corner of H. (This can be achieved by mirroring the objects in one or both of the coordinate axes, i.e. $x_1 \mapsto -x_1$ or $x_2 \mapsto -x_2$ or both.) Denote this vertex by r, cf. Fig. 5.14.*

Step 6. *Denote the remaining vertices of T by s and t so that (cf. Fig 5.15)*

(i) s lies on the right edge of H and

(ii) t lies above the line segment from r to s.

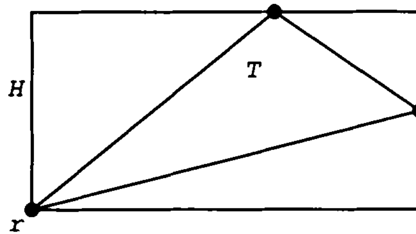


Figure 5.14: The requirement of Step 5

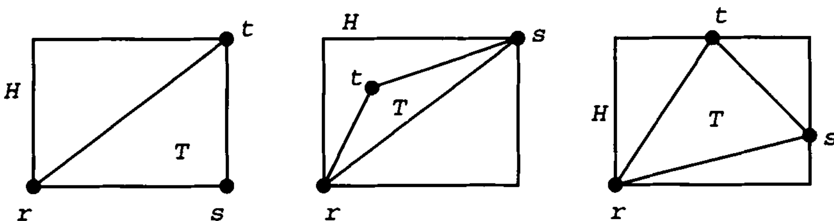


Figure 5.15: The requirement of Step 6

If conditions (i) and (ii) cannot be achieved, mirror the objects of the test on the main diagonal of the plane (that is, swap coordinates x_1 and x_2).

In order to test T and D_H for ***intersection*** it is only necessary to investigate the two edges of D_H which are "visible" from the vertex r. Only the endpoints need to be checked. If, however, one wants to know whether D is contained in T or not, then one has to consider all the edges, respectively all the corners.

We denote the corners of D_H by

$$k = (k_1, k_2), \quad k' = (l_1, k_2), \quad k'' = (k_1, l_2), \quad k''' = (k_2, l_2),$$

cf. Step 3 of Alg. 11. The corners k, k', k'' and the edges from k to k' and from k to k'' are interpreted as ***visible from*** r if the relationships between r and the corners are as shown in Fig. 5.16.

The intersection test is now based on observing only whether the visible corners lie on one side or on different sides of the triangle edges. This is best done by utilizing slopes. Let $x, y \in R^2$ then the (directed) ***slope of the line segment from*** x ***to*** y is denoted by

$$sl(x, y) = (y_2 - x_2)/(y_1 - x_1).$$

We admit values $\pm\infty$ by setting

$$sl(x, y) = (sgn(y_2 - x_2)) = \infty \quad \text{if } x_1 = y_1, \quad x_2 \neq y_2.$$

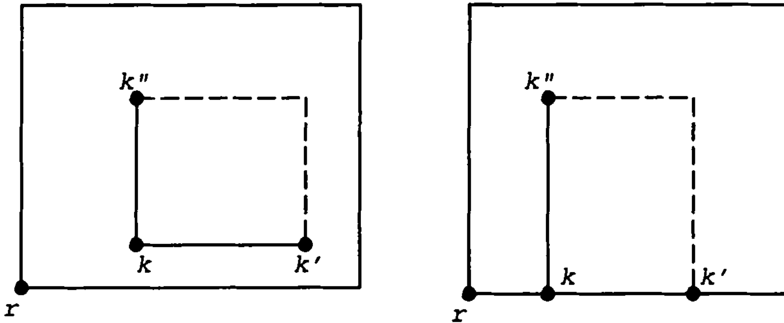


Figure 5.16: k, k' and k'' are visible from r

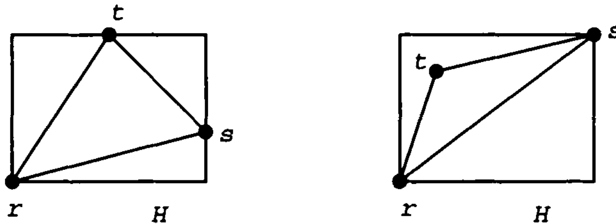


Figure 5.17: The two cases for location of t in Step 7

If $x = y$ then we introduce a special convention. In this case $sl(x, y)$ is assigned that slope value with which $sl(x, y)$ is compared. For example, if $x = y$ and

$$sl(x, y) > sl(r, s)$$

is to be evaluated $sl(x, y)$ is set to $sl(r, s)$. (In such situations $r = s$ will not occur.) The background for this convention is that it works well and that it is helpful in avoiding cases that arise from degenerate situations (i.e. the cases where D_H shrinks to a line segment or to a point).

We distinguish two cases, i.e. t lies on the upper edge of H or it does not, cf. Fig. 5.17. The latter case implies that s is the right upper corner of H.

The reason for taking this difference into consideration is that it allows us to develop steps that are taylored to the geometry. Furthermore, it allows us to commence the branching of the algorithm in such a manner that the computational costs are kept low.

Step 7. *If $t_2 = b_2$ (that is, t lies on the upper edge of H) then go to Step 7A otherwise go to Step 7B.*

The following test steps are now the real test steps. They are all based on the relationships between the slopes of a triangle edge and line segments. For

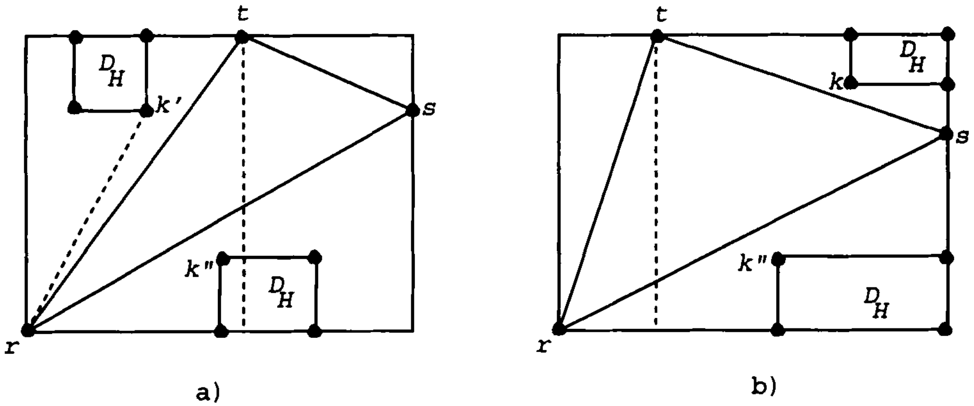


Figure 5.18: Slope relationships

example, if $sl(r, k') > sl(r, t)$ then the point k' is above the triangle edge from r to t, and hence it follows that D_H and T cannot intersect, cf. Fig. 5.18a).

Step 7A. If $k_1 \leq t_1$ (that is, k is to the left of t) go to (i) else go to (ii).

(i) If $sl(r, k') > sl(r, t)$ or $sl(r, s) > sl(r, k'')$ then $D \cap T = \emptyset$ otherwise $D \cap T \neq \emptyset$ (Fig. 5.18a)). End.

(ii) If $k_2 \geq s_2$ (that is, k is above s) then

if $sl(t, k) > sl(t, s)$ then $D \cap T = \emptyset$ else $D \cap T \neq \emptyset$

else

if $sl(r, s) > sl(r, k'')$ then $D \cap T = \emptyset$ else $D \cap T \neq \emptyset$ (Fig. 5.18b)). End.

Step 7B. If $k_2 \geq t_2$ (that is, k is above t) then go to (i) otherwise go to (ii).

(i) If $l_1 \leq t_1$ (that is, k' is to the left of t) then

if $k' \neq t$ then $D \cap T = \emptyset$ else $D \cap T \neq \emptyset$

else

if $[sl(r, s) > sl(r, k'')$ or $sl(t, k') > sl(t, s)]$ then $D \cap T = \emptyset$ else $D \cap T \neq \emptyset$ (Fig. 5.19a)). End.

(ii) If $[sl(r, k') > sl(r, t)$ or $sl(r, s) > sl(r, k'')]$ then $D \cap T = \emptyset$ else $D \cap T \neq \emptyset$ (Fig. 5.19b)). End.

Remarks.

1. The overall number of arithmetic operations and comparisons is rather low. We count at most 8 subtractions, 4 divisions and a few comparisons.

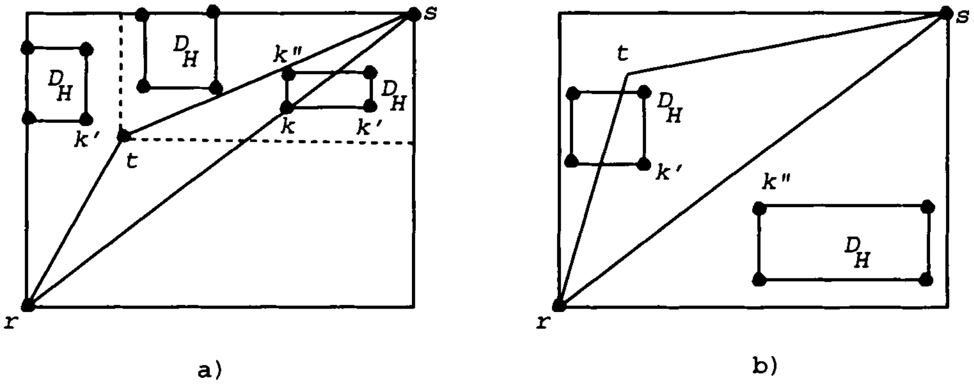


Figure 5.19: k above t and k below t

2. The *completeness* of Alg. 12 is clear from the completeness of the cases that occur, and the correctness can be verified from the geometric configuration, i.e. the figures, directly.

In contrast to Alg. 11, Alg.12 does not determine what type of intersection (i.e. overlapping or $D \subseteq T$) it is. Only the case $T \subseteq D$ is covered by Step 2. Supplementary tests are required in order to get a decision for either overlapping of D and T or for the decision $D \subseteq T$. Hence if it has already been shown in Step 7 that $D \cap T \neq \emptyset$ then

$D \subseteq T$ iff,
in case of Step 7A(i):
$sl(r,k'') \leq sl(r,t), sl(t,k''') \leq sl(t,s)$ and $sl(r,s) \leq sl(r,k')$,
in case of Step 7A(ii):
$sl(t,k''') \leq sl(t,s)$ and $sl(r,s) \leq sl(r,k')$,
in case of Step 7B(i):
$k' = k''$ (in case $l_1 \geq t_1$),
$sl(t,k'') \leq sl(t,s)$ and $sl(r,s) \leq sl(r,k')$ (in case $l_1 > t_1$),
in case of Step 7B(ii):
$sl(r,k'') \leq sl(r,t)$ and $sl(r,s) \leq sl(r,k')$ (in case $k_1 \leq t_1$),
$sl(t,k'') \leq sl(t,s)$ and $sl(r,s) \leq sl(r,k')$ (in case $k_1 > t_1$).

The additional cost for checking whether a triangle and a rectangle overlap or whether the rectangle is contained in the triangle after the intersection has been established is at most 2 slope computations and 2 comparisons. This means that the total number of operations is at most 12 subtractions, 6 divisions and a couple of comparisons are required.

In the same manner as in Alg. 11 machine interval arithmetic can be used to control the problems that might occur when the algorithm is implemented using floating point arithmetic. This would make this algorithm more expensive

since all the real calculations would have to be replaced by machine interval arithmetic calculations.

5.5 Box-Tetrahedron Intersection Testing

Many applications in computer graphics and solid modelling require the determination of whether two objects intersect or not or if one object is strictly contained in another (see, for example, [64, 156, 153]). Typical situations where this occurs are in determining whether a tool will touch an object, whether two objects in a scene do not intersect such that the scene is realistic or whether a robot can maneuver through a maze.

The procedure, following [220], is a generalization of the work in the previous section.

It turned out that Greene's [81] approach and the one used in [215] for 2D used almost the same geometric idea. Hence in generalizing [215] to 3D it is sufficient to refer to Greene [81] for details regarding the geometric background. The tests and the algorithms developed here and those used by Greene remain significantly different, in particular with respect to the requirement that the computations should result in a guaranteed answer.

The geometric key for the test is just the well-known fact that a box B and a tetrahedron T do not intersect iff there exists a plane separating B and T which is parallel to

(i) a face of B or

(ii) a face of T or

(iii) an edge of B and an edge of T,

cf. Greene [81], p. 78. It follows that B and T intersect iff

α) the three axis-parallel projections of B and T intersect and

β) for any closed halfspace H which contains T where a face of T lies in the boundary of H it holds that B and H intersect.

In contrast to Greene who uses this criterion directly and applies it to detect polyhedra and box intersections, we introduce interval barycentric coordinates and operate with a reduced box. The box is seen as a three dimensional interval and is treated as one object. The interval barycentric coordinates give additional information due to their geometric properties without increasing the computational costs. The box B is furthermore reduced to that part, B_H, which lies in the box hull, $H = H(T)$, of T where $H(T)$ is the smallest axis-parallel box that contains T, see also Figure 5.20. The advantage of dealing with B_H instead of B is, that the intersection properties remain unchanged and it is a larger chance that the test resp. the algorithm will lead to an earlier decision.

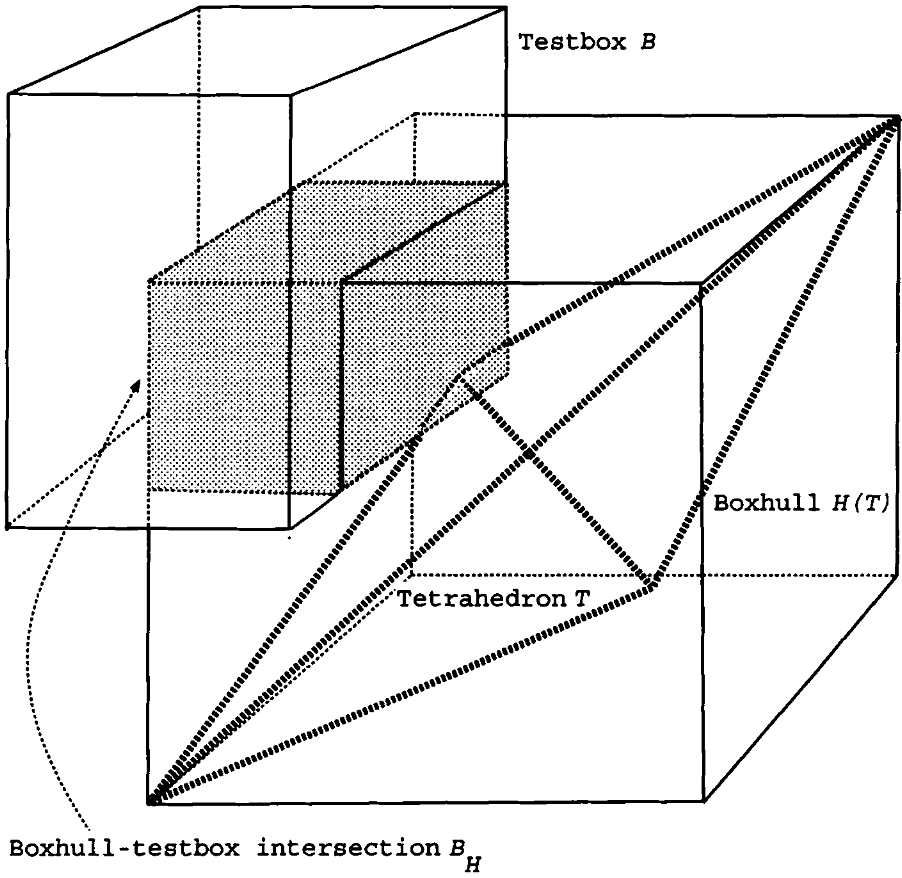


Figure 5.20: The bounding box and smallest axis-parallel box

A crucial part of the procedure is based on the use of interval arithmetic to evaluate a set of interval barycentric coordinates. The use of these coordinates allows us to decide a number of cases without further computations as well as to control the errors in the computations.

The central part of the procedure is based on Skelboe's principle from section 2.9 which is applied here to functions $f(x)$, $x = (x_1, x_2, x_3)$ of 3 real variables.

The value of the principle is that the complete box B, that is, the set of all the points of it, can be interpreted as exactly *one* interval arithmetic expression when described by barycentric coordinates. This implies that only *one* comparison is required in order to determine, provided some plane is given, whether the box lies on the positive or negative side of the plane or whether plane and box intersect. Nevertheless, some theoretic background is needed to make the construction of the interval arithmetic expression mentioned understandable, although the construction itself is simple. This, together with the introduction of three-dimensional interval barycentric coordinates, is done in the next section.

Interval tools also make it easy to distinguish between overlapping of two objects and one containing the other. It is, certainly, not difficult at all to detect whether T lies in B, but conversely, a couple of comparisons is needed to decide whether B and T overlap or whether B is contained in T.

The aim of this section is to develop and verify a procedure for establishing the possible box-tetrahedron relationships determining the conditions for:

1. containment of the tetrahedron in the box,
2. containment of the box in the tetrahedron,
3. when the tetrahedron and the box overlap,
4. when the tetrahedron and the box do not intersect.

In order to be complete we note that there are a variety of other techniques that can be applied to the problem discussed here. As an example we note that the results of Chazelle-Dobbin [26] can be specialized to our problem. In their paper, the detection of the intersection between convex polyhedra is studied and results asymptotic in the number of polyhedron vertices are presented. The intersection testing problem is reduced in dimension introducing the idea of drums (slices through vertices parallel to a coordinate plane) and recursion between chains in two dimensions. Bimodal search is used to improve the efficiency of subproblems. Clearly, slicing a general tetrahedron parallel to a coordinate plane through vertices creates three drums on the average with up to eight vertices each. The intersections of these drums with the box then have to be computed. Just the generation of extra vertices makes this method uncompetitive with our method since our method is specifically designed for

the case of small numbers of vertices. Another possible algorithm would be the computation of the maximum and minimum value of each barycentric coordinate occurring at box vertices [247] so that interval barycentric coordinates are not required. These approaches would again violate the requirement that the computations should result in a guaranteed answer.

5.5.1 Three-Dimensional Interval Barycentric Coordinates

If a non-degenerated tetrahedron $T \subseteq R^3$ is given, the barycentric coordinates w.r. to T are often used to describe the location of a point $q \in R^3$ w.r. to T. If the point q is replaced with a box B being a 3-dimensional interval, we speak of 3-dimensional interval barycentric coordinates. They are an excellent and simple means to describe the location of B w.r. to T and are therefore appropriate as basic expressions for an intersection test.

We consider four points $a, b, c, d \in R^3$ forming the vertices of a non-degenerate tetrahedron denoted by T. The barycentric coordinates of a point $p \in R^3$ with respect to the tetrahedron are computed as

$$\left.\begin{aligned} \gamma_1(p) &= \frac{\mathrm{vol}(p,b,c,d)}{\mathrm{vol}(a,b,c,d)}, \\ \gamma_2(p) &= \frac{\mathrm{vol}(a,p,c,d)}{\mathrm{vol}(a,b,c,d)}, \\ \gamma_3(p) &= \frac{\mathrm{vol}(a,b,p,d)}{\mathrm{vol}(a,b,c,d)}, \\ \gamma_4(p) &= \frac{\mathrm{vol}(a,b,c,p)}{\mathrm{vol}(a,b,c,d)} \end{aligned}\right\} \tag{5.19}$$

where

$$\mathrm{vol}(r,s,t,u) = \frac{1}{6}\begin{vmatrix} r_1 & r_2 & r_3 & 1 \\ s_1 & s_2 & s_3 & 1 \\ t_1 & t_2 & t_3 & 1 \\ u_1 & u_2 & u_3 & 1 \end{vmatrix}$$

for $r, s, t, u \in R^3$, cf. for instance, Hanson [98]. Although this definition is complete we note that $\mathrm{vol}(r, s, t, u)$ is the volume of the oriented tetrahedron with vertices r, s, t, u (in this order). It can be degenerate.

Barycentric coordinates have a number of interesting properties. Here we need the following two:

$$p \in T \text{ iff } 0 \le \gamma_i(p) \le 1,\ i = 1, \ldots, 4, \tag{5.20}$$

$$\sum_{i=1}^{4} \gamma_i(p) = 1, \text{ for } p \in R^3. \tag{5.21}$$

Hence, for example, the points p lying on the plane spanned by the vertices a, b, c, or, which is the same, spanned by that face of T which has a, b, c as

vertices, are characterized by $\gamma_4(p) = 0$. In a similar manner, the points p lying in that closed halfspace of the plane just mentioned in which T lies, are characterized by $\gamma_4(p) \geq 0$.

In the same manner as in [215] we now consider a box $B = B_1 \times B_2 \times B_3$, also denoted by $B = (B_1, B_2, B_3), B_i \in I, i = 1, 2, 3$. Then

$$B \subseteq T \text{ iff } 0 \leq \gamma_i(p) \leq 1, i = 1, \ldots, 4 \text{ for all } p \in B. \tag{5.22}$$

The statement "for all" $p \in B$ simply means that all points in the range of $\gamma_i(p)$ over B , defined as $\Box\gamma_i(B) = \{\gamma_i(p) | p \in B\}$, must have the property (5.22). In other words, the box B is included in T iff all barycentric coordinates are in the range $[0, 1]$ i.e. we need to evaluate $\Box\gamma_i(B)$, $i = 1, \ldots, 4$ and then to test the inclusion in $[0, 1]$.

The exact evaluation of the range of a function is in general a difficult and time consuming task (see for example [212]). The present case has, however, some special properties which we now exploit.

For simplicity we only consider the case for $i = 1$ since the other cases are similar and we generalize the techniques used in [215]. Letting $i = 1$ we have that

$$\gamma_1(B) = \frac{\text{vol}(B, b, c, d)}{\text{vol}(a, b, c, d)} \supseteq \Box\gamma_1(B) \tag{5.23}$$

by (2.17) using interval arithmetic for the natural interval extension of γ_1 over B. Since the denominator is only a scalar, we only need to consider the numerator of (5.23) for the purpose of obtaining the extent of the range and we get

$$\begin{aligned} \text{vol}(B, b, c, d) &= \frac{1}{6} \begin{vmatrix} B_1 & B_2 & B_3 & 1 \\ b_1 & b_2 & b_3 & 1 \\ c_1 & c_2 & c_3 & 1 \\ d_1 & d_2 & d_3 & 1 \end{vmatrix} \\ &= \left.\begin{array}{c} \{(B_1 - d_1)((b_2 - d_2)(c_3 - d_3) - (b_3 - d_3)(c_2 - d_2)) \\ -(B_2 - d_2)((b_1 - d_1)(c_3 - d_3) - (b_3 - d_3)(c_1 - d_1)) \\ +(B_3 - d_3)((b_1 - d_1)(c_2 - d_2) - (b_2 - d_2)(c_1 - d_1))\} / 6 \end{array}\right\} (5.24) \end{aligned}$$

and the principle previously mentioned applies since each interval variable occurs only once. This means that for this particular expression we have

$$\Box\text{vol}(B, b, c, d) = \text{vol}(B, b, c, d).$$

With the determinants evaluated in this manner and using the point value of $\text{vol}(a, b, c, d)$ it follows that $\gamma_i(B)$, $i = 1, \ldots, 4$ can be used in the procedure to make a definite decision as to the tetrahedron-box relationship for the following ad-hoc cases:

1. Inclusion of the box in the tetrahedron. This occurs when $\gamma_i(B) \subseteq [0, 1]$ for $i = 1, \ldots, 4$ or already when $\gamma_i(B) \geq 0$ [resp. ≤ 1] for $i = 1, \ldots, 4$ by (5.21).

2. The box is contained in one of the regions defined by $\gamma_i(q) < 0$ or $\gamma_i(q) > 1$ for some i, $i = 1, \ldots, 4$. That is, if $\gamma_i(B) < 0$ or $\gamma_i(B) > 1$ for some i, $i = 1, \ldots, 4$ then clearly $B \cap T = \emptyset$.

Certainly, not every box-tetrahedron relationship is covered by these two sample cases.

5.5.2 The Algorithm

It was already mentioned earlier that we use Greene's observation as the geometric background for the procedure. This observation says that the box B and the tetrahedron T intersect iff B and T intersect in all three projections parallel to the main axes and that B does not lie entirely outside any planes spanned by a face of T.

We do not, however, pursue Greene's procedure which first searches for the box corners that are farthest (regarding a positive as well as a negative direction) from the planes spanned by the faces of T. The procedure then observes those corners to determine on which side of the related planes they are in order to force a decision about box-plane intersection. If no decision has been reached so far, the projections of B and T are checked for their intersections. This last step cannot be avoided in our procedure either, and we again use 2D-barycentric coordinates, whereas Greene uses his algorithm focused on 2D. The reasons for the choice of 2D-barycentric coordinates also in 2D are the same as for choosing the 3D-barycentric coordinates for the 3D problem treated in this section.

In this subsection, we first formulate the geometric idea with interval barycentric coordinates such that a test based on this would already be complete and correct. We then add a couple of further test conditions that need almost no additional computations. These additional test conditions lead in general to earlier decisions than if they were not incorporated.

Test 1 *for checking intersection between B and T.*

1. If $\gamma_i(B) < 0$ for some $i = 1, \ldots, 4$, then B and T are disjoint.

2. If $p_j(B)$ and $p_j(T)$ are disjoint for some $j = 1, 2, 3$, then B and T are disjoint (p_1, p_2, p_3 are the 3 axis-parallel projections).

3. In the remaining cases, B and T intersect.

Note that $p_j(B)$ is a rectangle and $p_j(T)$ consists of one or the union of two nondegenerate triangles. Detecting their intersection property can be performed with the test given in [7] or in Greene [81].

It is quite simple to distinguish between the intersection cases for overlapping and complete containment:

Let H be the axis-parallel box hull of T, that is, the smallest axis-parallel box that contains T.

Test 2 *for checking overlapping and containment of B and T.*

1. $T \subseteq B$ iff $H = H \cap B$ (i.e., $H \subseteq B$).

2. $B \subseteq T$ iff $0 \leq \gamma_i(B)$ for $i = 1, \ldots, 4$.

3. In all the other cases of intersection (Test 1) B and T overlap.

Proof. The correctness of 1. and 3. is obvious. In 2., if $B \subseteq T$, no point p of B lies outside T, i.e. no point p of B satisfies $\gamma_i(p) < 0$ for any i. Since $\gamma_i(B) = \{\gamma_i(p) : p \in B\}$, we get $0 \leq \gamma_i(B)$. The converse direction uses the same argument. □

In order to establish the algorithm, we add a few additional steps to the two tests, even though they are already complete. The reason is that, on the average, an earlier decision is forced via the additional test steps and yet the added numerical effort is almost negligible. In particular, it makes sense to first check how many vertices of T lie in B. This can be executed just by coordinate comparisons. Then one needs to apply the test only for the case that no vertex of T lies in B. We then restrict the test to that part of B which lies in the box hull, H, of T rather than to the whole box B. There are two reasons for doing this. The first reason is that this part, denoted by $B_H = B \cap H$, has the same intersection relationship with T as with B, except for one degenerate case where B and T touch each other on an edge but no vertex of T lies in B. (This will be considered in Step 5 of the Algorithm.) The second reason is that by dropping the parts outside H, earlier test-steps can bring a decision. For example, it may be that $\gamma_1(B_H) < 0$ such that disjointness of B and T is proven, but $\gamma_i(B) < 0$ need not be the case such that it would be necessary to compute the projections, etc., cf. Test 1. Finally, it makes sense to add test questions to see if

$$\gamma_i(B) > 1$$

for some $i = 1, \ldots, 4$ is valid which also raises the effectivity of the procedure. For points $p \in R^3$, checking the comparison $\gamma_i(p) > 1$ would be completely superfluous after having checked the conditions $\gamma_i(p) < 0$ for $i = 1, \ldots, 4$ (cf. Step 1 of Test 1). This is due to property (4) which implies that $\gamma_i(p) > 1$ for some $i = 1, \ldots, 4$ assumes $\gamma_j(p) < 0$ for some $j = 1, \ldots, 4$, and converse. However, this situation cannot be transferred to interval barycentric coordinates. Hence, $\gamma_i(B) > 1$ for some $i = 1, \ldots, 4$ just says that $\gamma_i(p) > 1$ for all $p \in B$ holds. This means, by (4), that for each $p \in B$ an index j exists such that $\gamma_j(p) < 0$. Since the numbers j can vary, one cannot conclude that $\gamma_j(p) < 0$ for all $p \in B$ (with constant j) such there is no relationship with $\gamma_j(B) < 0$ (cf. Step 1 of Test 1).

ALGORITHM 13 *(For testing overlapping, containment and disjoint relationship between B and T.)*

Step 1. *Check the number of vertices of T lying in B:*
if 4 then $T \subseteq B$,
if 2, or 3 then T and B overlap.

Step 2. *Compute H, the axis-parallel box hull of T,*
compute $B_H = H \cap B$.

Step 3. *If $B_H = \emptyset$ then B and T are disjoint.*

Step 4. *For $i = 1, \ldots, 4$,*
(i) compute the interval barycentric coordinates $\gamma_i(B_H)$,
(ii) if $\gamma_i(B_H) < 0$ or $\gamma_i(B_H) > 1$ then B and T are disjoint.

Step 5. *If $\gamma_i(B_H) \geq 0$ for $i = 1, \ldots, 4$ then*
$B \subseteq T$, if $B_H = B$, otherwise B and T overlap.

Step 6. *For $j = 1, 2, 3$:*
if $p_j(T)$ and $p_j(B_H)$ are disjoint then T and B are disjoint.

Step 7. *If no decision was made in the previous steps then B and T overlap.*

Step 6 of the algorithm 13can be executed by the procedure given in the previous section which also is based on 2D interval barycentric coordinates.

Remarks.

1. The algorithm can certainly be used to check for "intersection" or "disjointedness" only. In this case, the results $T \subseteq B$, $B \subseteq T$ and "overlapping" are to be interpreted as "B and T intersect".

2. The algorithm is complete and correct. When it is implemented on a machine, however, correctness can be weakened due to rounding errors. This can be mitigated if machine interval arithmetic is implemented with "outward rounding" (see [5] or [8]). Then the numerical execution remains correct so far that if the algorithm delivers a definite decision it is guaranteed.

3. **Computational costs.** We briefly compare the computational costs of Greene's [81] and algorithm 13. The algorithm of Chazelle-Dobbin [26] and related procedures are incomparable because they aim to behave asymptotically best w.r. to polyhedra and the number of their vertices. Hence the advantages of these methods do not apply to our case.

   The numerical effort of Greene's and our approach is almost the same if worst cases are considered. To determine the planes spanned by the faces is comparable with the computation of the interval barycentric coordinates. Working with barycentric coordinates has the small advantage of

getting the orientation without additional effort. The number of inequalities is a bit higher in our case due to the enrichment of the algorithm by further steps.

The advantage of our algorithm is the incorporation of the additional steps (being superfluous from the standpoint of logic), which lead to an earlier decision. For instance, one only has to apply the main body of algorithm 13 if no vertices of T lie in B.

4. If one wants to detect intersection of a box B with a polyhedron P, one can use almost the same algorithm. The geometric background is again the observation that B and P do not intersect iff there exists a separating plane that is (i) parallel to a polyhedron face, or (ii) parallel to a box face, or (iii) parallel to a polyhedron edge and a box edge (Greene [81]). Again, this is equivalent to the following criterion: B and P intersect iff a) B does not lie entirely outside the plane of any face of P, and b) the (axis parallel) projections of P and B intersect (Greene [81]). The projections of P and B give a (solid) polygon p and a rectangle D. Dropping one dimension, the criterion reduces to the following one: D and p intersect iff a) D does not lie entirely outside the lines spanned by any edge of p, and b) D intersects the rectangle hull of p (Greene [81], p. 77). Hence it is obvious that tests 1, 2 and also algorithm 13 can easily be adapted to a box-polyhedron intersection test, where again the relationship between P and B or p and D can be described by interval barycentric coordinates. In order to determine the interval barycentric coordinates which need to be related to a tetrahedron (or a triangle in the 2D case) by definition, a subdivision of P into tetrahedrons is not advised since, in general, always one face of such a tetrahedron would be needed to check the inequality $\gamma_i(B) < 0$. Hence, any vertex of P not lying on the plane spanned by the face in question will do it. If, however, one additionally wants to incorporate the knowledge of $\gamma_i(B) > 1$ properly one has to search for a vertex of P which is farthest from the face in question.

5.5.3 Examples

As an example consider the tetrahedron T defined by the four points $a = (1,1,4)$, $b = (1,1,1)$, $c = (4,1,1)$ and $d = (1,4,1)$. The axis-parallel box hull is $H = ([1,4],[1,4],[1,4])$.

Example 1. Consider the box $B = ([1.3,1.6],[1.3,1.6],[1.3,1.6])$. Since Step 1 of algorithm 13 does not apply and since $B_H = B \neq \emptyset$ we calculate the interval barycentric coordinates as $\gamma_1(B_H) = [0.01, 0.2]$, $\gamma_2(B_H) = [0.4, 0.7]$, $\gamma_3(B_H) = [0.01, 0.2]$ and $\gamma_4(B_H) = [0.01, 0.2]$. Since $B_H \geq 0$, $i = 1, \ldots, 4$ it

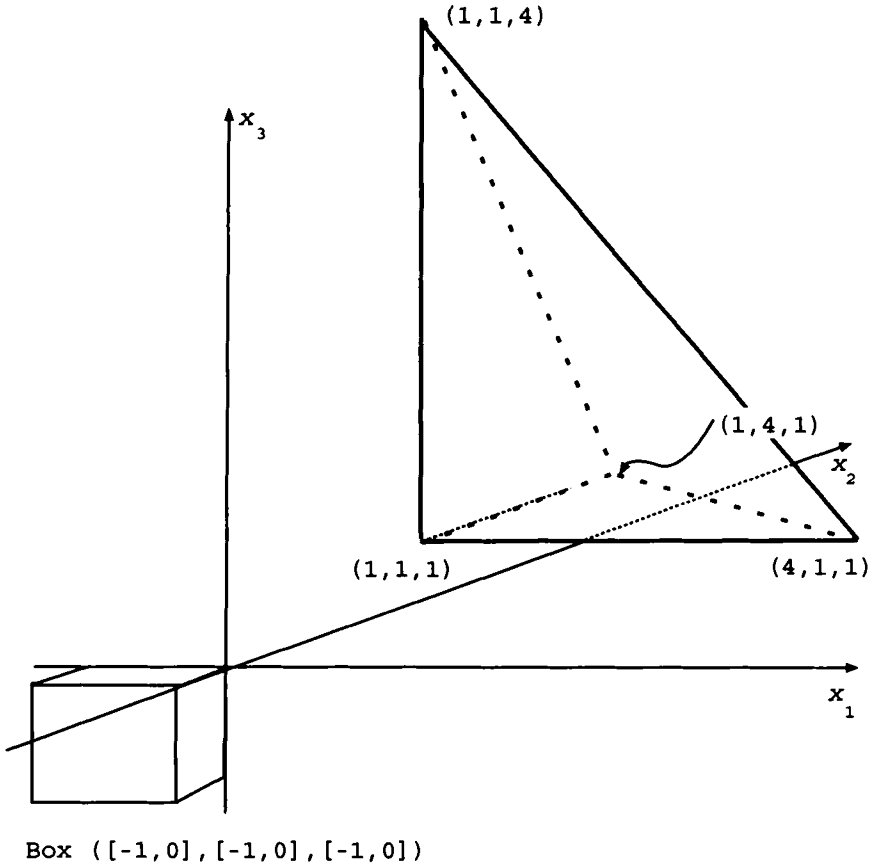


Figure 5.21: Box-tetrahedron test

follows that $B \subseteq T$.

Example 2. Consider the box $B = ([2,3],[2,3],[0,2])$. Since Step 1 does not apply we calculate $B_H = B \cap H = ([2,3],[2,3],[1,2])$. Now, $\gamma_1(B_H) = [0,1/3]$, $\gamma_2(B_H) = [-2/3,1/3]$, $\gamma_3(B_H) = [1/3,2/3]$ and $\gamma_4(B_H) = [1/3,2/3]$. Hence, neither Step 4 nor Step 5 of algorithm 13 lead to a decision. The same holds for Step 6. Hence we are left with Step 7 which means that B and T overlap.

Example 3. Consider the box $B = ([-1,0],[-1,0],[-1,0])$. Since $B_H = \emptyset$ it follows immediately that T and B do not intersect. This example is shown in Figure 5.21.

5.6 Ellipse-Rectangle Intersection Testing

We now present an efficient algorithm, also described in [224] for testing whether a solid rectangle and a solid ellipse intersect. The algorithm requires at most two evaluations of the quadratic polynomial that defines the ellipse and a few simple arithmetic expression executions. Convexity and monotonicity properties of this polynomial are the main tools for the design of the algorithm.

Testing for intersection between a rectangle and an ellipse can for example occur in computer graphics in windowing operations [54], in solid modeling with boolean operations [280] and in databases while executing neighborhood queries [143]. Implicitly the test is required for these applications in 3D. We know of no efficient test for an ellipse - rectangle intersection although such a test is found as a primitive in Class TGEllipse [264]. It works with an axes-aligned rectangle and an arbitrary ellipse. In the case of a non-axes-aligned rectangle, rectangle and ellipse have to be rotated to make the rectangle axis-aligned.

Let the rectangle $\mathcal{R}$ to be tested be defined by its four corners,

$$(x_L,\ y_L),\ (x_R,\ y_L),\ (x_R,\ y_R) \quad \text{and} \quad (x_L,\ y_R)$$

with $x_L \leq x_R$ and $y_L \leq y_R$. Thus $\mathcal{R}$ is a Cartesian product,

$$\mathcal{R} = X \times Y \tag{5.25}$$

where $X = [x_L,\ x_R]$ and $Y = [y_L,\ y_R]$.

An ellipse curve can be described as the zero set of a second degree polynomial,

$$\begin{aligned} f(x,\ y) &= ax^2 + bxy + cy^2 + dx + ey + f && (5.26)\\ &= (ax + by + d)x + (cy + e)y + f && (5.27) \end{aligned}$$

with

$$4b^2 < ac, \tag{5.28}$$

see for example [52] and we assume that

$$a > 0 \tag{5.29}$$

without loss of generality.

The solid ellipse corresponds to the point set $E = \{(x, y) : f(x, y) \leq 0\}$. Hence, a point $(x,\ y)$ of the plane lies in the ellipse (including its edge) iff $f(x,\ y) \leq 0$. Throughout this section, we will say ellipse instead of filled ellipse.

Further discussions of ellipses in a graphics setting are found in [229], pp. 236-242, [105] and in [146] or [54] where conversion from a representation using major, minor axes, orientation and origin to an implicit form is discussed.

Conditions (5.28) imply that f is *convex* and hence, that f is *convex over each straight line* in the plane. Convexity properties are connected with monotonicity properties of f, and we will make intense use of them.

In contrast to our method, the geometrically most straightforward method for deciding whether E and $\mathcal{R}$ do intersect is probably the direct geometric method, which might begin checking whether one of the 4 edges of $\mathcal{R}$ cuts the boundary of E, etc. Despite the simple logical structure of this test it is quite expensive since it requires the solution of four quadratic equations in the worst case as well as the execution of several comparisons and arithmetic expressions. The correct implementation of a quadratic equation solver is surprisingly complex, cf. [58] and more accessible in [199].

No quadratic equation need to be solved in our method.

5.6.1 Analytical Tools Needed

In this subsection, we collect the prerequisites that are needed in order to develop our test without too many interruptions.

A. The Midpoint of the Ellipse

Let $mE = (x^*,\ y^*)$, the *midpoint* of E be defined as the unique minimizer of f, cf.[36]. Then mE is the solution of the equation $\nabla f(x,\ y) = (0,\ 0)$ where

$$x^* = \frac{be - 2cd}{4ac - b^2},\quad y^* = \frac{bd - 2ae}{4ac - b^2}.$$

Because of the convexity of f and the minimum property of the midpoint, f is monotonically increasing on each ray leaving mE. This fact plays a key role in the logical structure of the algorithm.

B. Monotonicity

The knowledge of the monotonicity properties of f on the edges of $\mathcal{R}$ is also an essential part of the intersection test.

Let $\mathcal{I}$ be the lower or upper edge of $\mathcal{R}$, that is, $\mathcal{I} = (X,\ y_0)$ with $y_0 = y_L$ or $y_0 = y_R$, cf. Fig. 5.22. Then f is monotonically increasing [resp. decreasing] on $\mathcal{I}$ iff $\nabla_x f(x,\ y_0) \geq 0$ [resp. ≤ 0] for any $x \in X$ holds, where $\nabla_x f$ denotes the partial derivative of f with respect to x. This infinite set of inequalities will be written concisely as $\nabla_x f(X,\ y_0) = 2aX + by_0 + d \geq 0$ [resp. ≤ 0]. We use interval arithmetic to compute these expressions. Similarly, f is monotonically increasing [resp. decreasing] on an edge $(x_0,\ Y)$, where $x_0 = x_L$ or $x_0 = x_R$ iff $\nabla_y f(x_0,\ Y) = 2cY + bx_0 + e \geq 0$ [resp. ≤ 0].

In order to conveniently express increasing or decreasing function values if one moves from one corner of the rectangle to an adjacent corner, the following notation will be useful:

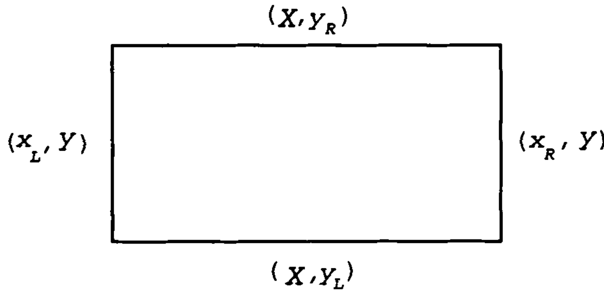


Figure 5.22: Rectangle

Let (x_0, y_0) be one of the four corners of $\mathcal{R}$. We set

$$\widetilde{\nabla}_X = \widetilde{\nabla}_X(x_0, y_0) = \begin{cases} \nabla_x f(X, y_0) & \text{if } x_0 = x_L, \\ -\nabla_x f(X, y_0) & \text{if } x_0 = x_R, \end{cases}$$

$$\widetilde{\nabla}_Y = \widetilde{\nabla}_Y(x_0, y_0) = \begin{cases} \nabla_y f(x_0, Y) & \text{if } y_0 = y_L, \\ -\nabla_y f(x_0, Y) & \text{if } y_0 = y_R. \end{cases}$$

For example, the condition $\widetilde{\nabla}_X \geq 0$ indicates that if the point (x, y) moves on the edge (X, y_0) from the endpoint (x_0, y_0) to the other endpoint on this edge, the function values increase monotonically .

C. Inexpensive Computation of Discriminants

In a few cases of the test it is necessary to know whether an axis parallel straight line meets the ellipse or not. This can be decided by considering the sign of the discriminant of f over this line. We will keep the computation of the discriminant inexpensive by expressing it in terms that are already known at the stage of the test where the discriminant is needed.

We note that a quadratic equation in one variable,

$$\alpha x^2 + \beta x + \gamma = 0$$

is solvable in $\mathcal{R}$ iff its *discriminant*, $D = \beta^2 - 4\alpha\gamma$, is nonnegative.

Hence, the system of equations,

$$\left. \begin{array}{l} f(x, y) = 0 \\ y = y_0 \end{array} \right\} \tag{5.30}$$

which is equivalent to the equation in x,

$$f(x, y_0) = ax^2 + (by_0 + d)x + (cy_0^2 + ey_0 + f) = 0,$$

has a solution in $\mathcal{R}$ (that is, the line $y = y_0$ meets E) iff

$$D(y_0) = (by_0 + d)^2 - 4a(cy_0^2 + ey_0 + f) \geq 0 \tag{5.31}$$

where $D(y_0)$ is the discriminant of $f(x, y_0) = 0$. Reformulating (5.31) results in

$$\begin{aligned} D(y_0) &= (2ax_L + by_0 + d)^2 - 4a[ax_L^2 + bx_L y_0 + cy_0^2 + dx_L + ey_0 + f] \\ &= [\nabla_x f(x_L, y_0)]^2 - 4af(x_L, y_0) \\ &= [\inf \nabla_x f(X, y_0)]^2 - 4af(x_L, y_0) \\ &= [\inf \tilde{\nabla}_X(x_0, y_0)]^2 - 4af(x_0, y_0) \text{ if } x_0 = x_L. \end{aligned}$$

Similarly, we obtain

$$\begin{aligned} D(y_0) &= (2ax_R + by_0 + d)^2 - 4a[ax_R^2 + bx_R y_0 + cy_0^2 + dx_R + ey_0 + f] \\ &= [\sup \nabla_x f(X, y_0)]^2 - 4af(x_R, y_0) \\ &= [\inf \tilde{\nabla}_X(x_0, y_0)]^2 - 4af(x_0, y_0) \text{ if } x_0 = x_R. \end{aligned}$$

Putting these two reformulations together, we can express the solvability of (5.30) as follows:

$$\begin{aligned} &\text{(5.30) has no solution in } \mathcal{R} \text{ iff} \\ &D(y_0) = [\inf \tilde{\nabla}_X(x_0, y_0)]^2 - 4af(x_0, y_0) < 0. \end{aligned} \tag{5.32}$$

If we interchange the variables x and y, and apply the previous considerations to the system

$$\left.\begin{aligned} f(x, y) &= 0 \\ x &= x_0 \end{aligned}\right\} \tag{5.33}$$

we get analogously that

$$\begin{aligned} &\text{(5.33) has no solution in } R \text{ iff} \\ &D(x_0) = [\inf \tilde{\nabla}_Y(x_0, y_0)]^2 - 4cf(x_0, y_0) < 0. \end{aligned} \tag{5.34}$$

Testing the conditions (5.32) and (5.34) in the algorithm will be inexpensive since the values $\tilde{\nabla}_X$, $\tilde{\nabla}_Y$, and $f(x_0, y_0)$ as well as $4a$ (or $4c$) would already have been evaluated at that stage of the computations. In fact, only 4 or 5 arithmetic operations are required additionally in order to evaluate the discriminants.

5.6.2 The Algorithm

In this subsection, an algorithm is established that tests whether a given solid ellipse E in general position and a given solid axis-parallel rectangle do or do not intersect. The input data is E represented by the coefficients of the function f, and $\mathcal{R}$ represented by the 4 necessary coordinates x_L, x_R, y_L, y_R. It is assumed that $\mathcal{R}$ is non-degenerate.

If E and $\mathcal{R}$ intersect then the algorithm does not tell whether $\mathcal{R}$ is contained in E or vice versa. Appropriate tests are provided in the next section if such additional information is requested. They can be merged with the algorithm at the indicated points.

ALGORITHM 14 *(Ellipse-rectangle intersection test.)*

Step 1. *Compute the midpoint* $mE = (x^*, y^*)$ *of* E.

Step 2. *If* $mE \in \mathcal{R}$ *then*

[$\mathcal{R} \cap E \neq \emptyset$ *(Options:* $E \subseteq \mathcal{R}$ *Test,* $\mathcal{R} \subseteq E$ *Test). STOP].*

Step 3. *Let* (x_0, y_0) *be a corner of* $\mathcal{R}$ *which is nearest to* mE *and* (x_f, y_f) *be one which is farthest away from* mE.

Step 4. *Compute* $f(x_0, y_0)$.
If $f(x_0, y_0) \leq 0$ *then*

[$\mathcal{R} \cap E \neq \emptyset$ *(Option:* $\mathcal{R} \subseteq E$ *Test). STOP].*

Step 5. *Cases:*

Case A : $x^* \in X$

(*i*) *Compute* $\tilde{\nabla}_X = \tilde{\nabla}_X(X, y_0)$,
(*ii*) *if* $\tilde{\nabla}_X \geq 0$ *then [*$\mathcal{R} \cap E = \emptyset$*, STOP],*
if $\tilde{\nabla}_X \leq 0$ *then*
α) *compute* $f(x_f, y_0)$,
β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $f(x_f, y_0) > 0$*, STOP,*
else
α) *compute* $D(y_0)$,
β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $D(y_0) < 0$*, STOP.*

Case B : $y^* \in Y$

(*i*) *Compute* $\tilde{\nabla}_Y = \tilde{\nabla}_Y(x_0, y_0)$.
(*ii*) *if* $\tilde{\nabla}_Y \geq 0$ *then [*$\mathcal{R} \cap E = \emptyset$*, STOP],*
if $\tilde{\nabla}_Y \leq 0$ *then*
α) *compute* $f(x_0, y_f)$,
β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $f(x_0, y_f) > 0$*, STOP,*
else
α) *compute* $D(x_0)$,
β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $D(x_0) < 0$*, STOP.*

Case C : $x^* \notin X, y^* \notin Y$

(*i*) *Compute* $\tilde{\nabla}_X = \tilde{\nabla}_X(x_0, y_0)$,
if $\tilde{\nabla}_X \leq 0$ *then*
α) *compute* $f(x_f, y_0)$,
β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $f(x_f, y_0) > 0$*, STOP,*
if $\inf \tilde{\nabla}_X < 0$ *then*

α) *compute* $D(y_0)$,

β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $D(y_0) < 0$, *STOP.*

(*ii*) *Compute* $\tilde{\nabla}_Y = \tilde{\nabla}_Y(x_0, y_0)$,

if $\tilde{\nabla}_Y \geq 0$ *then* [$\mathcal{R} \cap E = \emptyset$, *STOP*],

if $\tilde{\nabla}_Y \leq 0$ *then*

α) *compute* $f(x_0, y_f)$,

β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $f(x_0, y_f) > 0$, *STOP,*

else

α) *compute* $D(x_0)$,

β) *the result is* $\mathcal{R} \cap E = \emptyset$ *iff* $D(x_0) < 0$, *STOP.*

THEOREM 10 *The algorithm is correct and complete.*

Proof

Completeness. The algorithm is complete when each ellipse-rectangle constellation which is admitted by the general assumptions will be processed correctly by the algorithm. That the algorithm meets this definition, follows directly from its logical structure, which can be reformulated as a nested sequence of *if-then-else* statements and *cases* in the following form (computations of expressions can be suppressed):

If A1 **then** Res1 **else**
  If A2 **then** Res2 **else**
    Case 1 **or** *Case 2* **or***Case 3*

Since the three cases cover each geometric constellation which has been left so far, each selection of input data will be processed by one of the statements A1, A2, *Case 1*, *Case 2*, or *Case 3*.

Correctness. An algorithm is correct if it assigns the right result (boolean, in our case) to any possible input data. In order to prove the correctness of the algorithm we have to go through all its branches and to input them geometrically. The discussion of the options will be postponed to the next section.

Step 2: Intersection is obvious, cf. Fig. 5.23a.

Step 4: $f(x_0, y_0) \leq 0$ means $(x_0, y_0) \in E$ and intersection follows, cf. Fig. 5.23b.

Step 5: Because of the previous steps, we have the following assumptions in the sequel,

$$mE \notin \mathcal{R} \text{ and } (x_0, y_0) \notin E.$$

Case A(ii) says $x^* \in X$ and $\tilde{\nabla}_X \geq 0$, that is, the function f is monotonically increasing if f passes from x_0 to x_f on the edge (X_0, y_0). Since $f(x_0, y_0) > 0$,

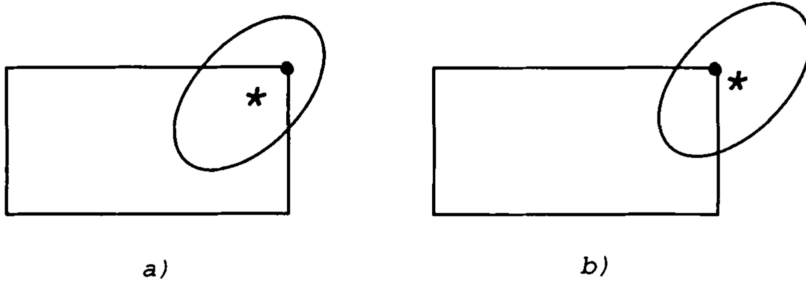


Figure 5.23: The initial easy cases of intersection[3]

we get $f(X, y_0) > 0$ for all $x \in X$, so that the whole edge is outside E. Further, each point $(x, y) \in \mathcal{R}$ lies on a ray which leaves mE and passes through the edge (X, y_0) before it reaches (x, y). Since the function values increase monotonically on the ray, $f(x, y) > 0)$ follows. That is, E is outside $\mathcal{R}$. No intersection, cf. Fig. 5.24a.

Case A(iii) The conditions $\tilde{\nabla}_X \leq 0$ says that f is monotonically increasing if f moves on the edge (X, y_0) from x_f to x_0. Hence, if $f(x_f, y_0) > 0$ then $f(x, y_0) > 0$ for all $x \in X$, and this edge is outside E. For the same reason as in *Case A(ii)*, the whole ellipse is outside E, cf. Fig. 5.24b.

If $f(x_p, y_0) \leq 0$, then (x_f, y_0) is an ellipse point, and $E \cap \mathcal{R} \neq \emptyset$, cf. Fig. 5.24c.

The *else*-clause is addressed if neither $\tilde{\nabla}_X \geq 0$ nor $\tilde{\nabla}_X \leq 0$ holds. In this case, f is not monotone on the edge (X, y_0) and one has to check the discriminant condition directly for an intersection with this edge, which implies the intersection result for E and $\mathcal{R}$ because of $x^* \in X$, cf. Fig. 5.24e and 5.24c.

Case B is symmetric to *Case A* (swap x with y).

Case C(i) Besides $x^* \in X$, this case is analogous to *Case A(iii)*.

Case C(ii) Besides $y^* \in Y$, this case is analogous to *Case B(iii)*.
□

5.6.3 Optional Inclusion Tests

It was already mentioned that the aim of Algorithm 14 is to decide whether $E \cap \mathcal{R} \neq \emptyset$ or $E \cap \mathcal{R} = \emptyset$ holds. In case of $E \cap \mathcal{R} \neq \emptyset$, it is sometimes also of interest to know whether $\mathcal{R} \subseteq E$ or $E \subseteq \mathcal{R}$ or if neither of the two inclusions hold. In this section we therefore suggest auxiliary tests which are

[3]The midpoint of the ellipse is denoted by $\star$ and (x_0, y_0) by $\bullet$.

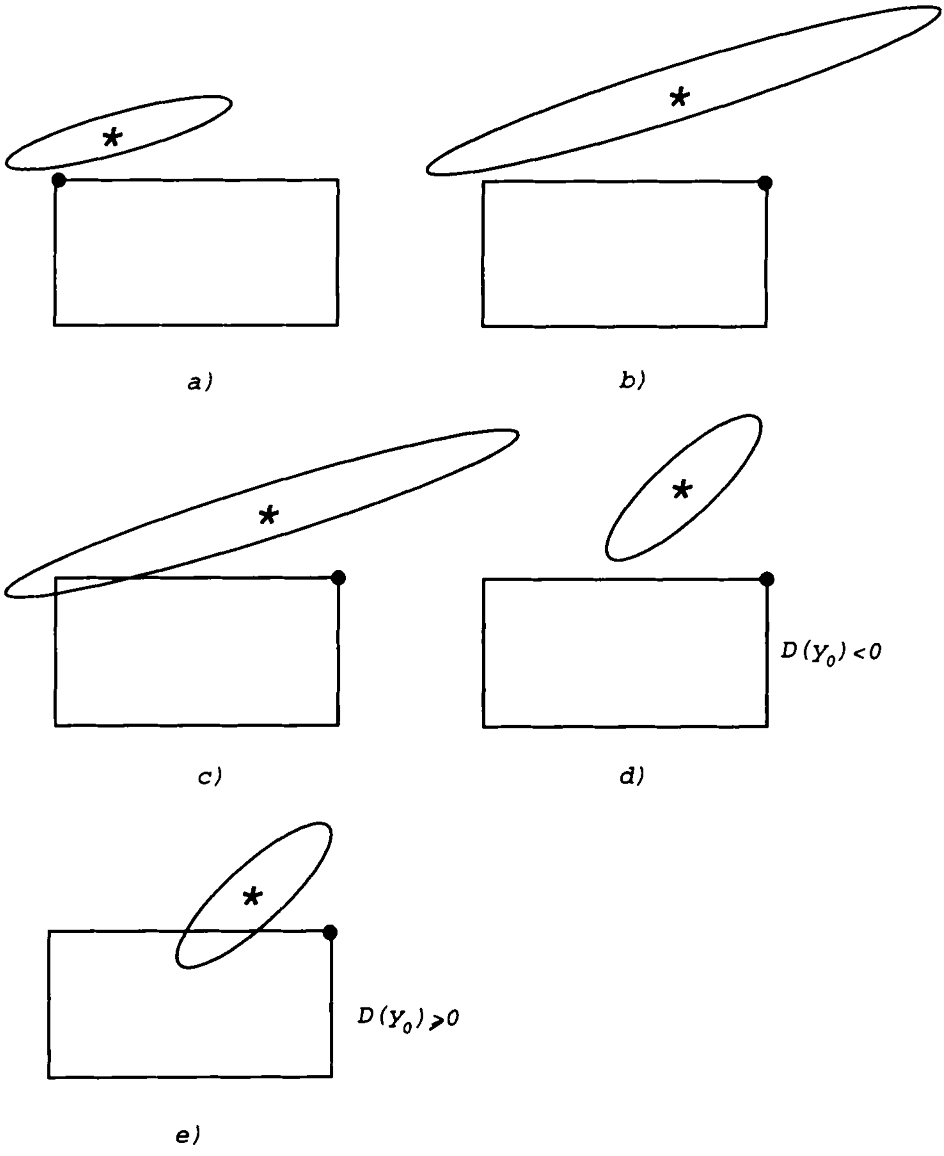


Figure 5.24: The remaining cases when $mE \notin \mathcal{R}$ and $(x_0, y_0) \notin E$

to be inserted into Algorithm 14 at the positions marked.

The optional $E \subseteq \mathcal{R}$ test only has to be executed in Step 2 since $mE \in \mathcal{R}$ is a necessary assumption for the validity of the inclusion $E \subseteq \mathcal{R}$. Hence, if the $E \subseteq \mathcal{R}$ test is not invoked during the execution of Algorithm 14, $E \subseteq \mathcal{R}$ does not hold.

Optional $E \subseteq \mathcal{R}$ Test

(valid only as part of Step 2 in Alg. 14)

Step 1'. Determine $(x_0,\ y_0)$ as in Step 3.

Step 2'. Compute $f(x_0,\ y_0)$.
If $f(x_0,\ y_0) \leq 0$ then $[E \not\subseteq \mathcal{R}$, STOP$]$.

Step 3'. Compute $\tilde{\nabla}_X = \tilde{\nabla}_X(x_0,\ y_0)$.
If $\inf \tilde{\nabla}_X < 0 < \sup \tilde{\nabla}_X$ then

(*i*) compute $D(y_0)$,
(*ii*) if $D(y_0) > 0$ then $[E \not\subseteq \mathcal{R}$, STOP$]$.

Step 4'. Compute $\tilde{\nabla}_Y = \tilde{\nabla}_Y(x_0,\ y_0)$.
If $\inf \tilde{\nabla}_Y < 0 < \sup \tilde{\nabla}_Y$ then

(*i*) compute $D(x_0)$,
(*ii*) the result is $E \subseteq \mathcal{R}$ iff $D(x_0) \geq 0$,

else

$E \subseteq \mathcal{R}$.

The optional $\mathcal{R} \subseteq E$ test is less tied to the position in Algorithm 14 where it is invoked. The reason is that there is almost no other possibility except to compute the four function values at the corners of $\mathcal{R}$. It is not possible to decide the $\mathcal{R} \subseteq E$ test only with gradient or discriminant considerations because one can construct examples with identical gradient and discriminant information but with different $\mathcal{R} \subseteq E$ behavior. Again, as is the case with the $E \subseteq \mathcal{R}$ test, the inclusion $\mathcal{R} \subseteq E$ does not hold if the $\mathcal{R} \subseteq E$ test is not invoked in Step 2 or Step 4 during the execution of Algorithm 14.

Optional $\mathcal{R} \subseteq E$ Test

Step 1. Compute the function values $f(x_L,\ y_L)$, $f(x_L,\ y_R)$, $f(x_R,\ y_L)$, $f(x_R,\ y_R)$, which are not yet known (for example, $f(x_0,\ y_0)$ will be known already).
It follows that $\mathcal{R} \subseteq E$ iff none of these values are positive.

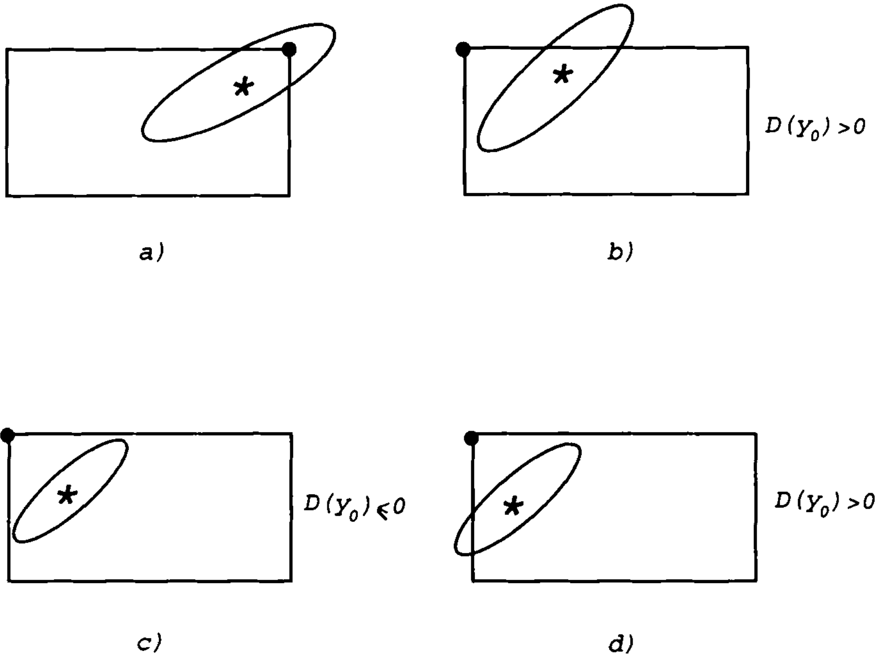


Figure 5.25: Cases for the $E \subseteq \mathcal{R}$ test

THEOREM 11 *Algorithm 14 enlarged by the inclusion tests is correct and complete.*

Proof. *Completeness.* The logical structure of Algorithm 14 is changed only marginally by adding the inclusion test. I.e., each admitted geometric rectangle-ellipse constellation will be recognized by the enlarged algorithm, and the completeness is kept.

Correctness. The correctness of the $\mathcal{R} \subseteq E$ test is obvious. Regarding the $E \subseteq \mathcal{R}$ test, it is first clear that $mE \in \mathcal{R}$ is a necessary condition for $E \subseteq \mathcal{R}$. Hence the test is attached to Step 2 only, that is, a confirmation that $E \subseteq \mathcal{R}$ holds arises from Step 2 only. If there is no such confirmation, $E \subseteq \mathcal{R}$ is not valid.

The correctness of the steps of the test rely on simple geometric considerations':

Step 2'. Condition $f(x_0, y_0) \leq 0$ says that (x_0, y_0) lies in E, so that there are adjacent points of E that lie outside $\mathcal{R}$, cf. Fig. 5.25a.

Step 3'. Because of the missing monotonicity of f on the edge (X, y_0), one has to check the discriminant condition for an intersection of E with the edge.

In case of a proper inclusion, that is, $D(y_0) > 0$, and a part of (X, y_0) will be outside E, cf. Fig. 5.25b, and $E \not\subseteq \mathcal{R}$.

Step 4' treats the remaining case, $D(y_0) \leq 0$. Even though there is no proper intersection of E with the edge (X, y_0) and intersection of E with the edge (x_0, Y) can well happen. If no monotonicity of f on the edge (x_0, Y) is given, E intersects $\mathcal{R}$ properly iff $D(x_0) > 0$, cf. Fig. 5.25c and 5.25d.

If monotonicity of f on the edge (x_0, Y) is given, the ellipse cannot pass this edge, and hence, since (x_0, y_0) is nearest to mE and $(x_0, y_0) \notin E$ through no other edge.

□

5.6.4 Complexity and Rounding Errors

The operational-logical structure of the algorithm 14 is a tree with several branches and subbranches, where the height of the tree is very low. The computational execution of the test is therefore very inexpensive. Summing up the algorithm needs:

1. the computation of mE, requiring 14 arithmetic operations,
2. 0, 1 or 2 function evaluations, requiring 10 arithmetic operations each (using the form (5.27)),
3. 0, 1 or 2 partial interval derivatives (if 2 are needed, only 1 function evaluation and no discriminants are needed), requiring 3 arithmetic and 2 interval arithmetic operations each (which is equivalent to 7 arithmetic operations each),
4. 0 or 1 discriminant evaluations (if 1 is needed only 1 function evaluation is needed), requiring 4 to 5 arithmetic operations each

and a few additional comparisons and sign checks.

The decision for intersection or not is therefore obtained after 2 inexpensive evaluations (mE, f) and 2 very inexpensive evaluations ($\widetilde{\nabla}$, D) or after 3 inexpensive evaluations and one very inexpensive evaluation.

If the optional inclusion tests are incorporated into algorithm 14 then the costs will not increase since the algorithm stops with the 2nd or 4th step if the inclusion tests are called up. Thus, if the optional tests are not invoked because the computation does not go through their branch, the cost analysis of the algorithm is still valid.

Hence, we sum up the overall costs if the optional tests are invoked (we again drop comparisons):

1. Computation of mE (for Algorithm 14),

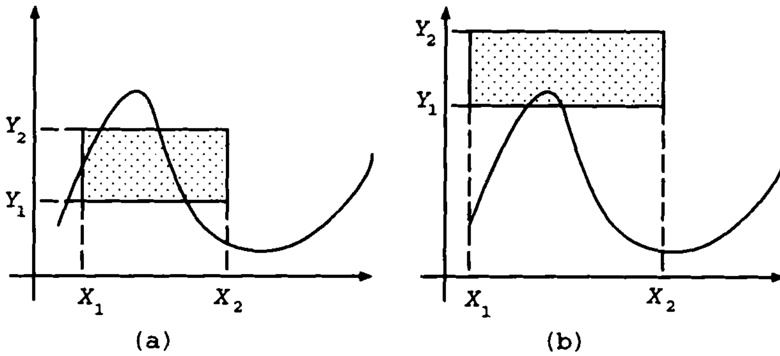


Figure 5.26: Intersection testing of curve and rectangle

2. 1, 2, 3 or 4 function evaluations (one for the $E \subseteq \mathcal{R}$ test, at most 3 for the $\mathcal{R} \subseteq E$ test),
3. 0, 1 or 2 partial interval derivative evaluations (for the $E \subseteq \mathcal{R}$ test),
4. 0, 1 or 2 discriminant evaluations (for the $E \subseteq \mathcal{R}$ test).

5.7 Intersection Between Rectangle and Explicitly Defined Curve

In this section some computational aspects of dealing with real curves are discussed. Objects bounded by curves occur frequently in computer graphics and solid modeling. When a windowing operation is performed then one wants to know whether such objects appear completely in the window, or partially, or perhaps not at all. This leads to a development of algorithms for testing whether a curve and a rectangle representing the frame of a window intersects. Similarly interference testing between objects whose boundaries are defined by curves and curved surfaces leads to curve-curve and surface-surface intersection testing.

We consider explicitly defined plane curves and whether they intersect an axes-parallel rectangle. The curves are either of the type $(x, f(x))$ or they are more generally defined via a parametrization, that is, of the type $(x(t), y(t))$.

First let the rectangle $\mathcal{R} = X \times Y$ and the curve $y = f(x)$, $x \in X_0$ be given. Here, $X = [x_1, x_2]$, $Y = [y_1, y_2]$, and X_0 are intervals. We assume that X and X_0 overlap, otherwise the rectangle and curve do not intersect. Only the area common to X and X_0 is meaningful for the test and we therefore replace $X \cap X_0$ by X as the working area. With this new notation, we have

$$\mathcal{R} = X \times Y \text{ and } y = f(x), \; x \in X,$$

cf. Fig. 5.26.

We further need an inclusion function F of f for the validated computation where

$$w(F(Z)) - w(\square f(Z)) \to 0 \quad \text{as} \quad w(Z) \to 0. \tag{5.35}$$

for subintervals Z of X. A real number $\epsilon > 0$ is needed as termination parameter.

We provide two different intersection tests for this type of curve. The first test needs no derivative information and is thus applicable to curves which are not necessarily continuously differentiable, for example, to curves which are only piecewise smooth. The principle underlying the algorithm is subdivision combined with function value comparisons in the subintervals. It is easy to write the code but the computation might be time consuming.

In this connection the condition (5.35) has to be explained. It says that the excess-width of the inclusions $F(Z)$ tends to zero as the width of the underlying interval Z does, that is, the approximation of the range of f over Z by the inclusion $F(Z)$ improves as the width of Z becomes smaller. Without this property the iterated subdivision which is pursued throughout the algorithm and which makes the subintervals smaller and smaller would be useless. This explanation is also valid for the second test.

The second test we provide is considerably faster and requires that the curve is smooth. This test is largely based on the interval Newton method.

In the *first test*, X is systematically subdivided into subintervals $Z = [z_1, z_2]$, and the intervals $F(Z)$ and $F(z)$ where $z \in Z$ for some z are evaluated. All intervals Z satisfying $F(Z) \subseteq R \setminus Y$ are excluded since they cannot contain points which are rectangle as well as curve points. The computation *terminates* when

(i) a point z or an interval Z is found with $F(z) \subseteq Y$ or $F(Z) \subseteq Y$ (intersection is confirmed),

(ii) points z and z' are found with
$F(z) \leq y_1 \leq F(z')$ or $F(z) \leq y_2 \leq F(z')$ (intersection is confirmed),

(iii) all the subintervals which are available so far have width smaller than ϵ (no decision has been possible up to now within the prescribed accuracy),

(iv) the list of available subintervals becomes empty (no intersection is confirmed).

The idea of the algorithm is to demonstrate that there is a point of the curve that is guaranteed to be in the box or that is guaranteed that no point of the curve is in the box. The algorithm therefore first computes an inclusion of the curve over the domain X. If this inclusion is outside Y then there is no intersection. Similarly if the inclusion is contained in Y then an inclusion is guaranteed. If neither of the above situations are valid then the algorithm proceeds by evaluating inclusions of the curve at the endpoints of the domain X.

If a decision is not reached for the whole rectangle, the rectangle is recursively subdivided and the process repeated till either a point of the curve lies in a subrectangle or no curve point lies in no subrectangle which gives a validated decision. If not decision is reached till all subrectangles are smaller than a give width, the algorithm is terminated with an uncertain result.

Input parameters are the inclusion function F for f over X which satisfies (5.35), the rectangle $\mathcal{R}$ and a termination parameter $\epsilon > 0$.

ALGORITHM 15 *(Rectangle curve intersection test)*

Step 1. *Compute $F(X)$.*

Step 2. *If $F(X) \cap Y = \emptyset$ then STOP (no intersection is confirmed).*

Step 3. *If $F(X) \subseteq Y$ then STOP (intersection is confirmed).*

Step 4. *Compute $F_1 = F(x_1)$ and $F_2 = F(x_2)$. If $F_1 \subseteq Y$ or $F_2 \subseteq Y$ then STOP (intersection is confirmed).*

Step 5. *If $F_1 \vee F_2 \geq y_2$ set $G(X) = F(X)$, $Z = Y$ and set $r = lb(F_1 \vee F_2)$ goto 8.*

Step 6. *If $F_1 \vee F_2 \leq y_1$ set $G(X) = -F(X)$, $Z = -Y$ and set $r = -ub(F_1 \vee F_2)$ go to 8.*

Step 7. *STOP (intersection is confirmed).*

Step 8. *Initialize list $\mathcal{L} = \{(X, r, 1)\}$.*

Step 9. *Denote by (W, q, d) the first element of $\mathcal{L}$ and remove this element from $\mathcal{L}$.*

Step 10. *If $G(W) \cap Z = \emptyset$ go to 14.*

Step 11. *If $G(W) \subseteq Z$ then STOP (intersection is confirmed).*

Step 12. *Subdivide W into W_1 and W_2.*

Step 13. *For $i = 1, 2$*

1. *If $ubG(mid\ m(W_i)) < z_2$ then STOP (intersection is confirmed).*
2. *Enter $(W_i, lbG(mid\ (W_i)), d + 1)$ onto the list $\mathcal{L}$ so that the list is first ordered by increasing third component then by increasing second component.*

Step 14. *If $\mathcal{L} = \emptyset$ then STOP (no intersection is confirmed).*

Step 15. *If $w(W) < \epsilon$ for all $(W, q, d) \in \mathcal{L}$ then STOP (all the subintervals W_i in the final (nonempty) list have a width smaller than ϵ).*

Step 16. *Goto 8.*

The algorithm behaves as binary search in the worst case. However, in general it only needs to subdivide until a decision has been reached. This may happen after just a few steps.

It is obvious that the algorithm as well as its numerical realization is complete, i. e., it deals with every possible configuration. The algorithm and the numerical realization are correct as a wrong result never can arise. The computation terminates after a finite number of steps since the subdivision process bisects the current subintervals uniformly so that their widths become finally smaller than ϵ, provided no earlier termination took place.

The *second test* can be considered in two parts. The first part tests for simple configurations which can be dealt with easily. In the second part the interval Newton procedure is applied in order to localize intersections of the curve with the lower or upper horizontal side of $\mathcal{R}$ after which the existence test is applied to provide guaranteed results.

Input parameters are the inclusion function F of f over X which satisfies (5.35), the rectangle $\mathcal{R} = X \times Y$ and the termination parameter $\epsilon > 0$, as well as the data which are required for running the interval Newton algorithm (the parameter ϵ' of the interval Newton algorithm must be larger than the parameter ϵ here.)

ALGORITHM 16 *(Improved rectangle curve intersection test)*

Step 1. *Initialize list $\mathcal{L} = \{X\}$. Remove X from list.*

Step 2. *Compute $F(x_1), F(x_2)$ and $F(X)$.*

Step 3. *If $F(x_1) \subseteq Y$ or $F(x_2) \subseteq Y$, then STOP (intersection is confirmed).*

Step 4. *If $F(X) \subseteq R \backslash Y$, then*

(i) if $\mathcal{L}$ is empty then STOP (no intersection is confirmed)

(ii) delete X (since it does not contain parameter values with curve points in $\mathcal{R}$)

and go to Step 8 (select next interval).

Step 5. *If $F(x_1) \leq y_2$ and $y_1 \leq F(x_2)$*

or

if $F(x_2) \leq y_2$ and $y_1 \leq F(x_1)$

then STOP (intersection is confirmed).

Step 6. *If $y_1 \in F(x_1) \vee F(x_2)$ or if $F(x_1), F(x_2) < y_1$,*
then apply interval Newton method to $f - y_1$ over X.

If during the interval Newton computation

(i) a point $z \in X$ or a subinterval Z is found with

$F(z) \subseteq Y$ *or* $F(Z) \subseteq Y$

then STOP (intersection is confirmed),

(ii) a subinterval Z is found with $F(Z) < y_1$ then Z can be deleted since it contains no intersection points with the curve,

(iii) the processing list of the interval Newton method becomes empty then STOP if $\mathcal{L}$ is empty (no intersection is confirmed) otherwise go to Step 8 (select next interval).

If the interval Newton computation terminates because all the subintervals Z_i in the final (nonempty) processing list of the interval Newton method are sufficiently small then apply the existence test to each of these boxes. As soon as the existence test verifies that a zero of the function $f - y_1$ exists, that is a solution of the equation $f = y_1$ in one of the these boxes then STOP (intersection is confirmed). Apply the statements (i), (ii), and (iii) also to the existence test computation.

Bisect those intervals which remained in the procession list of the interval Newton algorithm (existence test included) if their width is larger than $w(X)/2$ and put them at the end of the list $\mathcal{L}$. Put those intervals the width of them is already smaller than or equal to $w(X)/2$ at the end of the list $\mathcal{L}$ without bisecting them.

Step 7. *If $y_2 \in F(x_1) \vee F(x_2)$ or if $F(x_1), F(x_2) > y_2$, then apply interval Newton to $f - y_2$ and proceed symmetrically to Step 6.*

Step 8. *Remove the first interval from the list $\mathcal{L}$ that has width larger than or equal to ϵ and denote it by $X = [x_1,\ x_2]$. If the width of all intervals of the list (which cannot be empty) is smaller than ϵ then STOP (no decision about the intersection has been possible yet with respect to the chosen accuracy).*

Alg. 16 as well as its numerical realization are complete, correct and terminate after a finite number of steps. These properties are due to the logical construction of the algorithm and the properties of the interval Newton method.

More general, we now deal with a parametrized curve $z(t) = (x(t), y(t))$ with $t \in [0,1]$, and $z(t) \in R^2$ and wish to check for an intersection with the rectangle $\mathcal{R}$. We need an inclusion function $Z(T) = (X(T), Y(T)) \in I^2$ for $z(t)$, with $T \in I([0,1])$, where X and Y are inclusion functions for x and y. In analogy with the previous tests, the conditions

$$\left.\begin{array}{l} w(X(T)) - w(\Box x(T)) \to 0 \text{ as } w(T) \to 0, \\ w(Y(T)) - w(\Box y(T)) \to 0 \text{ as } w(T) \to 0 \end{array}\right\} \qquad (5.36)$$

are required for subintervals T of the parameter interval $[0, 1]$. Geometrically, Z(T) can best be interpreted as the rectangle $X(T) \times Y(T)$. A termination parameter $\epsilon > 0$ is needed as before.

The following algorithm checks whether the rectangle $\mathcal{R}$ is intersected by the curve $z(t)$. Similarly to Alg. 15, the parameter interval is subdivided, until it is recognizable that curve points are in $\mathcal{R}$ or that none of the subintervals contains any curve points. If the widths of all the subintervals are smaller than ϵ, the computation is terminated without a conclusive result. Interval Newton steps are not incorporated since these steps can only be reasonably applied when a well determined search interval is available. Theoretically, this would be possible, but practically, it would lead to a large number of cases which are not worth the effort.

The input parameters are the rectangle $\mathcal{R} = A \times B$, the curve $z(t) = (x(t), y(t))$ with $t \in [0, 1]$, the inclusion function Z as described above and which satisfies (5.36), and a termination parameter ϵ.

ALGORITHM 17 *(Testing for intersection between rectangle and parametric curve)*

Step 1. *Set* $T = [0, 1], t_1 = 0$, *and* $t_1 = 1$,

Initialize list $\mathcal{L} = \{T\}$. *Remove* T *from list.*

Step 2. *Compute* $Z(t_1)$, $Z(t_2)$ *and* $Z(T)$.

Step 3. *If* $Z(t_1) \subseteq \mathcal{R}$ *or* $Z(t_2) \subseteq \mathcal{R}$, *then STOP (intersection is confirmed).*

Step 4. *If* $Z(T) \subseteq R^2 \backslash \mathcal{R}$, *then*

(i) if list $\mathcal{L}$ *ist empty then STOP (no intersection is confirmed)*

(ii) delete T *(since* T *does not contain points with function values in* $\mathcal{R}$*) and go to Step 7 (select the next interval).*

Step 5. *(i) If* $Y(t_1) \leq b_2$ *and* $b_1 \leq Y(t_2)$

or

if $Y(t_2) \leq b_2$ *and* $b_1 \leq Y(t_1)$

then STOP (intersection is confirmed).

(ii) If $X(t_1) \leq a_2$ *and* $a_1 \leq X(t_2)$

or

if $X(t_2) \leq a_2$ *and* $a_1 \leq X(t_1)$

then STOP (intersection is confirmed).

Step 6. *Split* T *into two subintervals of equal length and put them at the end of the list.*

Step 7. *Remove the first interval from the list, denote it by T and its endpoints by t_1 and t_2. If $w(T) < \epsilon$ (then all intervals of $\mathcal{L}$ have width smaller than ϵ) then STOP (no decision about the intersection has been possible with respect to the chosen accuracy).*

Step 8. *Go to Step 2.*

Alg. 17 as well as its numerical realization are complete, correct and terminate after a finite number of steps. These properties are due to the logical construction of the algorithm.

Condition (5.36) says that the excess-widths of the inclusions $X(T)$ and $Y(T)$ tend to zero as the width of the underlying interval T does, that is, the approximation of the current range of x and y over T by the inclusions $X(T)$ and $Y(T)$ becomes better and better. Without this property the iterated subdivision which is pursued throughout the algorithm would be pointless.

5.8 Box-Sphere Intersection Test

5.8.1 Introduction

Testing for the intersection of a circle with an axis-parallel rectangle or for the intersection of a sphere with an axis- parallel box is a computer graphics primitive that occurs in a number of applications. In the field of solid modeling such tests are applicable as a preprocessor for many surface-surface intersection algorithms. In particular, the implementation of the tests in this section responds directly to the concerns expressed in [51] where it is stated that

> ...the numerical schemes available so far for dealing with offsets and intersections fail to meet the minimum standards of reliability, accuracy and efficiency...

The rectangle-circle test is used to decide whether a circle or disk overlaps a region in the view-plane and the sphere-box test is used in spatial subdivision techniques in ray tracing.

Arvo [10] describes a series of such tests where the midpoint m and the radius r act as sphere parameters. The tests determine the smallest or largest distance from m to the box and compare it with r. This enables [10] to solve several variants of the intersection problem such as solid box vs. solid sphere, solid box vs. hollow sphere, etc. The tests are independent of the dimension. Moreover, Guibas-Stolfi [85] propose an Incircle Test where the circle is defined by three points $a, b, c \in R^2$. The test recognizes whether a query point $e \in R^2$ lies inside this circle or not and it is based on the determination of the oriented volume of a 3-dimensional tetrahedron whose vertices lie on a rotation paraboloid. This leads to the computation of a 4 by 4 determinant. If now e is

replaced by a rectangle (being the Cartesian product of two intervals), and if the determinant is evaluated in a well-defined manner with interval arithmetic then a circle-rectangle intersection test results [215].

The *aim of this section*, based on [216], is therefore first to show that Arvo's box-sphere tests can be unified and simplified using interval techniques such that only one (interval-) function evaluation is needed to perform the tests. Then, the test in [215] is lifted to dimension 3 such that another box-sphere test results. We give a new proof that avoids arguing with tetrahedrons on hyper-rotation paraboloids (which would be the case if the test in [85] was generalized directly) and that is - with the exception of a sign - independent of the dimension such that it also provides a simpler proof for the test in [85]. It finally turns out that Arvo's and the extension of the test by Guibas-Stolfi are mathematically almost identical as the underlying formulas differ by just a constant. An algorithm for one of the tests is given and further numerical examples are provided.

We need a small *observation*. Let

$$f(x) = ax^2 + bx + c. \tag{5.37}$$

Then the range of $f(x)$ over $X = [x_L, x_R]$, denoted by $\square f(X)$ can be found by first computing $t_1 = f(x_L)$, $t_2 = f(x_R)$ and if $a \neq 0$ then $t_3 = f(-b/(2a))$ finally obtaining

$$\square f(X) = \begin{cases} [\min(t_1,t_2,t_3), \max(t_1,t_2,t_3)] & \text{if } -b/(2a) \in X \text{ and } a \neq 0 \\ [\min(t_1,t_2), \max(t_1,t_2)] & \text{otherwise.} \end{cases} \tag{5.38}$$

5.8.2 Midpoint and Radius as Sphere Parameters

Let a sphere S be defined by its *midpoint* $m \in R^3$ and its *radius* $r > 0$. Hence

$$S = \left\{x \in R^3 : \|x - m\| = r\right\}.$$

Let us now introduce a *distance comparing function*

$$F(x) = \|x - m\|^2 - r^2 \text{ for } x \in R^3. \tag{5.39}$$

Then, if the $\cdot$ stands for the standard inner product,

$$\begin{aligned} F(x) &= (x-m)\cdot(x-m) - r^2 \\ &= x \cdot x - 2x \cdot m + m \cdot m - r^2 \\ &= \sum_{i=1}^{3}(x_i^2 - 2x_i m_i) + m \cdot m - r^2. \end{aligned}$$

$F(x)$ compares the distance from x to m with the distance from the points of S to m, expressed in squares to avoid square roots. A point x is therefore inside S iff $F(x) < 0$.

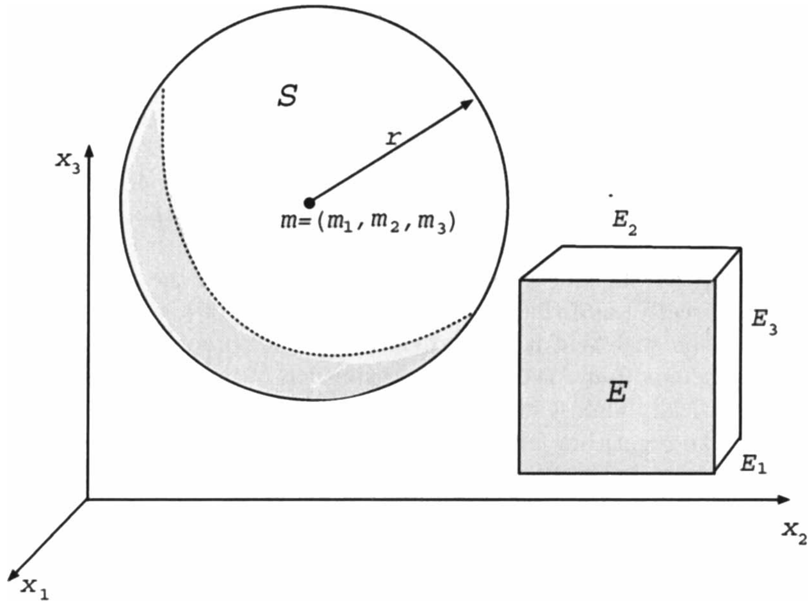


Figure 5.27: Box-sphere configuration

Let $E = (E_1, E_2, E_3)$ be an axis-parallel box, where $E_1, E_2, E_3 \in I(R)$. This box will be compared with S. A typical situation is shown in Figure 5.27. The comparison can be made if we take the distance comparison function F together with set-value considerations. For example, if

$$F(x) < 0 \text{ for all } x \in E$$

which is equivalently expressed by $\square F(E) < 0$, one can be sure that E lies inside the sphere. One only has to find a way to determine $\square F(E)$. This can be done easily in interval arithmetic:

Let $f_i(x_i) = x_i^2 - 2m_i x_i$ for $x_i \in R, i = 1, 2, 3$. Then f_i is of the form (5.37) and $\square f_i(E_i)$ can be evaluated by (5.38). By (5.39), $F(x) = \sum_{i=1}^{3} f_i(x_i) + m \cdot m - r^2$ for $x \in R^3$ and finally, $\square F(E)$ can be evaluated directly as

$$\square F(E) = \sum_{i=1}^{3} \square f_i(E_i) + m \cdot m - r^2$$

with interval arithmetic.

This leads to the following:

Test for the intersection behavior of the box E and the sphere S:

1. Evaluate $\square F(E)$ as explained above.

2. (i) $\square F(E) < 0$ iff E inside S (no intersection),
   (ii) $\square F(E) > 0$ iff E outside S (no intersection),
   (iii) $0 \in \square F(E)$ iff E and S intersect.

Example 1. As an example to illustrate the above consider a sphere S defined by $m = (0,1,0)$ and $r = 1$ together with a box $E = ([3,4],[0,2],[-1,1])$. Then $f_1(x_1) = x_1^2$, $f_2(x_2) = x_2^2 - x_2$ and $f_3(x_3) = x_3^2$ from which $\square f_1(E_1) = [9,16]$, $\square f_2(E_2) = [-1/4, 2]$ and $\square f_3(E_3) = [0,1]$ using (5.38). From this $\square F(E) = [8.75, 19] > 0$ showing that E is outside S.

The test is concerned with E as a solid object and with S as a hollow one. One can immediately derive variants - as considered by [10] - with S solid, E hollow, etc. For example, if E and S are solid, $\square F(E) \leq 0$ is equivalent to E inside S (which means here that $E \subseteq S$), and intersection is given. By the way, if a decision is needed to whether $S \subseteq E$ with E solid it is not necessary to compute $\square F(E)$. One can check this directly by computing

$$S \subseteq E \text{ iff } E_{iL} \leq m_i - r \leq m_i + r \leq E_{iR} \quad (i = 1,2,3)$$

where $E_i = [E_{iL}, E_{iR}]$.

When *numerically computing* $\square F(E)$ values $\square F(E)_{NUM}$ are calculated due to rounding errors. In this case there is an extremely small percentage chance that the test leads to a wrong decision due to the deviation of $\square F(E)_{NUM}$ from $\square F(E)$. Nevertheless, if machine interval arithmetic is used it is still possible to save almost all of the conclusions of the test. In this case, instead of the interval $\square F(E) = [u, v]$ only including intervals for the endpoints u and v can be calculated

$$u \in [u_L, u_R], \quad v \in [v_L, v_R],$$

and we have the following "guaranteed" conclusions summed up:

Numerical version of the Test (E solid, S hollow):

(i) $v_R < 0$ implies $\square F(E) < 0$ (i.e., E inside S),
$\square F(E) < 0$ implies $v_L < 0$.

(ii) $u_L > 0$ implies $\square F(E) > 0$ (i.e., E outside S), $\square F(E) > 0$ implies $u_R > 0$,

(iii) $u_R \leq 0 \leq v_L$ implies $0 \in \square F(E)$ (i.e., E and S intersect),
$0 \in \square F(E)$ implies $u_L \leq 0 \leq v_R$.

Remark 1. It is obvious that the considerations of this section can be formulated for each dimension.

5.8.3 Four Peripheral Points as Sphere Parameters

In [85] an *Incircle Test* is presented which determines whether a query point is inside a circle given by 3 of its points. We generalize this test to dimension 3 which is also of interest for the computation of three-dimensional Voronoi diagrams [85, 198]. The resulting *Insphere Test* accordingly determines whether a query point e is inside the sphere S that is now given by four of its peripheral points. We provide a new proof, which avoids hyper-geometric interpretations and which can be performed in any dimension. We also single out the close mathematical connections of the test with the distance comparing functions, $F(x) = (x-m)\cdot(x-m)-r^2$, introduced in Subsec. 5.8.2. After this, we go one step further and replace the query e by a box E and the Insphere Test becomes a test for box-sphere intersections, if some basic rules of interval arithmetic are considered.

Let $a, b, c, d \in R^3$ be affinely independent points. Then there exists a unique sphere, S, such that the given points lie on the sphere. We again denote its midpoint by m and the radius by r. The generalization of [85], the Boolean predicate Insphere(a, b, c, d, e) will be introduced and be defined to be true iff $e \in R^3$ is inside the sphere. Whereas the Incircle Test requires the determination of a 4 by 4 determinant the Insphere Test leads to the determinant

$$\mathcal{D}(a,b,c,d,e) = \begin{vmatrix} a_1 & a_2 & a_3 & a\cdot a & 1 \\ b_1 & b_2 & b_3 & b\cdot b & 1 \\ c_1 & c_2 & c_3 & c\cdot c & 1 \\ d_1 & d_2 & d_3 & d\cdot d & 1 \\ e_1 & e_2 & e_3 & e\cdot e & 1 \end{vmatrix}. \tag{5.40}$$

Let $\beta = b - a$, $\gamma = c - a$, $\delta = d - a$, $\epsilon = e - a$ and

$$V = \begin{pmatrix} \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \\ \delta_1 & \delta_2 & \delta_3 \end{pmatrix}.$$

Then

$$\mathcal{D}(a,b,c,d,e) = \left|\begin{array}{ccc:c} & & & b\cdot b - a\cdot a \\ & V & & c\cdot c - a\cdot a \\ & & & d\cdot d - a\cdot a \\ \hdashline \epsilon_1 & \epsilon_2 & \epsilon_3 & e\cdot e - a\cdot a \end{array}\right|. \tag{5.41}$$

In order to avoid orientation assumptions and dealing with oriented volumes (one way to proceed would be the assumption that β, γ, δ are positively oriented) we introduce

$$\sigma = \mathrm{sign}(|V|).$$

The determinant $|V|$ need not be evaluated separately since it occurs as a minor of (5.41). Since $a, b, c, d \in S$,

$$\|x - m\| = (x - m) \cdot (x - m) = r^2 \text{ for } x = a, b, c, d. \tag{5.42}$$

Subtracting the equation (5.42) for a from the corresponding equations for b, c, d yields

$$\begin{aligned} b \cdot b - a \cdot a &= 2m \cdot \beta, \\ c \cdot c - a \cdot a &= 2m \cdot \gamma, \\ d \cdot d - a \cdot a &= 2m \cdot \delta. \end{aligned} \tag{5.43}$$

This leads to a system of linear equations for the components of m,

$$V \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} b \cdot b - a \cdot a \\ c \cdot c - a \cdot a \\ d \cdot d - a \cdot a \end{pmatrix}. \tag{5.44}$$

Inserting the values of m into one of the equations (5.42) gives the value of r.

LEMMA 4 *(i)* $\mathcal{D}(a, b, c, d, e) = 16|V|F(e)$

(ii) Insphere(a, b, c, d, e) *is true iff* $\sigma\mathcal{D}(a, b, c, d, e) < 0$.

Proof. (i) By (5.42) and the definition of $F(x)$, we get

$$\begin{aligned} e \cdot e - a \cdot a &= e \cdot e + m \cdot m - a \cdot a - m \cdot m \\ &= (e - m) \cdot (e - m) + 2e \cdot m - (a - m) \cdot (a - m) - 2a \cdot m \\ &= F(e) + 2\epsilon \cdot m. \end{aligned}$$

Hence, by (5.43),

$$\mathcal{D}(a, b, c, d, e) = 16 \left| \begin{array}{ccc:c} & & & m \cdot \beta \\ & V & & m \cdot \gamma \\ & & & m \cdot \delta \\ \hdashline \epsilon_1 & \epsilon_2 & \epsilon_3 & m \cdot \epsilon + F(e) \end{array} \right|$$

$$= 16 \left| \begin{array}{ccc:c} & & & 0 \\ & V & & 0 \\ & & & 0 \\ \hdashline \epsilon_1 & \epsilon_2 & \epsilon_3 & F(e) \end{array} \right| \tag{5.45}$$

$$= 16|V|F(e). \tag{5.46}$$

(The determinant (5.45) results from the preceding one by multiplying the first column by $-m_1$, the second by $-m_2$, the third by $-m_3$ and then adding it to the fourth column.)

(ii) Now, e is inside S is equivalent to $F(e) < 0$, cf. Subsec. 5.8.2, which together with (i) - gives the result. □.

The lemma enables us to compare the two approaches to perform an *Insphere Test* if four affinely independent points $a, b, c, d \in R^3$ are given (that is, if $|V| \neq 0$):

1. Compute m, r and $F(e)$. If $F(e) < 0$ then e is inside S.

2. Compute σ and $\mathcal{D}(a, b, c, d, e)$ (advisably not directly as 5 by 5 matrix) and apply point (ii) of lemma.

Both approaches show about the same level of numerical costs and stability, even if the number of arithmetic operations of the first approach is a bit smaller.

The conclusions attained so far can immediately be transformed to box-sphere intersection tests based on determinants in the same manner as the Incircle Test in [85] formed the basis for the circle-rectangle test in [215]. According to our range notation we set $\square\mathcal{D}(a, b, c, d, E) = \{\mathcal{D}(a, b, c, d, e) : e \in E\}$ for a box $E \subseteq R^3$. Furthermore, the Boolean predicate Insphere(a, b, c, d, E) shall be true iff E lies inside S. Then the lemma gets the following form

COROLLARY 1 *(i)* $\square\mathcal{D}(a, b, c, d, E) = 16|V|\square F(E)$.

(ii) Insphere(a, b, c, d, E) is true iff $\sigma\square\mathcal{D}(a, b, c, d, E) < 0$.

The Corollary enables us to compare the two approaches to perform an *Insphere Test for boxes E*, if affinely independent points $a, b, c, d \in R^3$ are given:

1. Compute m, r and $\square F(E)$. If $\square F(E) < 0$, E lies inside S.

2. Compute σ and $\square\mathcal{D}(a, b, c, d, E)$ and apply point (ii) of the Corollary.

Remark 2. Because of the close connection between F and $\mathcal{D}$ shown in point (i) of the corollary, $\square\mathcal{D}(a, b, c, d, E)$ can be used in the same manner as $\square F(E)$ for several variants of the Insphere Test (box outside sphere, box intersects sphere, etc.) cf. Subsec. 5.8.2.

Remark 3. If one wants to evaluate $\square\mathcal{D}(a, b, c, d, E)$ without determining m and r, the following procedure is suggested:

Developing the determinant (5.41) by the fourth row yields

$$\mathcal{D}(a, b, c, d, e) = -\epsilon_1 w_1 + \epsilon_2 w_2 - \epsilon_3 w_3 + (e \cdot e - a \cdot a) w_4$$

where the w_i's are the appropriate minors. Let

$$z = (b \cdot b - a \cdot a, \;\; c \cdot c - a \cdot a, \;\; d \cdot d - a \cdot a)^T$$

and let V_i denote the ith column of V, then the minors are

$$w_1 = |V_2 \;\; V_3 \;\; z|, \;\; w_2 = |V_1 \;\; V_3 \;\; z|, \;\; w_3 = |V_1 \;\; V_2 \;\; z|, \;\; w_4 = |V|. \tag{5.47}$$

From $e \cdot e - a \cdot a = \epsilon \cdot \epsilon + 2\epsilon \cdot a$ we get

$$\mathcal{D}(a,b,c,d,e) = \sum_{i=1}^{3} f_i(\epsilon_i) = \sum_{i=1}^{3} f_i(e_i - a_i) \tag{5.48}$$

where

$$f_i(x_i) = w_4 x_i^2 + ((-1)^i w_i + 2a_i w_4)x_i. \tag{5.49}$$

One notices that $f_i(x_i)$ is of the form (5.37) such that $\Box f_i(E_i - a_i)$ can be calculated with formula (5.38). This gives

$$\Box\mathcal{D}(a,b,c,d,E) = \sum_{i=1}^{3} \Box f_i(E_i - a_i). \tag{5.50}$$

Example 2. We continue Example 1 by considering S as being defined by the four points $a = (1,1,0)$, $b = (0,1,1)$, $c = (0,2,0)$ and $d = (0,0,0)$. Then

$$\begin{aligned} \beta &= b - a = (-1,0,1), \\ \gamma &= c - a = (-1,1,0), \\ \delta &= d - a = (-1,-1,0) \end{aligned}$$

so that

$$V = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & -1 & 0 \end{pmatrix}.$$

We also calculate $z = (0,2,-2)^T$. Now

$$w_1 = \begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ -1 & 0 & -2 \end{vmatrix} = 0,$$

$$w_2 = \begin{vmatrix} -1 & 1 & 0 \\ -1 & 0 & 2 \\ -1 & 0 & -2 \end{vmatrix} = -4,$$

$$w_3 = \begin{vmatrix} -1 & 0 & 0 \\ -1 & 1 & 2 \\ -1 & -1 & -2 \end{vmatrix} = 0$$

and finally $w_4 = |V| = 2$. From these calculations (or using Gaussian elimination directly on (5.44)) we could recreate m and r using (5.42) and (5.44) then proceed as in Example 1. Since our aim is to illustrate the above theory we elect instead to proceed as above with equation (5.49) obtaining the separable components of $\mathcal{D}$ as

$$\begin{aligned} f_1(x_1) &= 2x_1^2 + 4x_1, \\ f_2(x_2) &= 2x_2^2, \\ f_3(x_3) &= 2x_3^2. \end{aligned}$$

Finally we can compute $\Box\mathcal{D}(a,b,c,d,E) = \Box f_1([2,3]) + \Box f_2([-1,1]) + \Box f_3([-1,1]) = [16,30] + [0,2] + [0,2] = [16,34] > 0$ from (5.50). This shows that E is outside S.

5.8.4 Algorithm

We now present an algorithm for the case when the sphere S is defined by four affinely independent points using the equations suggested in Remark 3.

ALGORITHM 18 *(Sphere-box intersection test)*

Step 1. *Four affinely independent points a, b, c and d defining the sphere and $E = (E_1, E_2, E_3)$ defining the box are entered as data.*

Step 2. *Calculate vectors β, γ and δ and form matrix V.*

Step 3. *Compute w_i, $i = 1, \ldots, 4$ from (5.47).*

Step 4. *Evaluate the coefficients of $f_i(x_i)$, $i = 1, 2, 3$ using (5.49).*

Step 5. *Evaluate $\Box\mathcal{D}(a,b,c,d,E)$ using (5.50).*

Step 6.

1. *$\Box\mathcal{D}(a,b,c,d,E) < 0$ implies that the box E is contained in the sphere S.*
2. *$\Box\mathcal{D}(a,b,c,d,E) \ni 0$ implies that the box E and the sphere S intersect.*
3. *$\Box\mathcal{D}(a,b,c,d,E) > 0$ implies that the box E and the sphere S are disjoint.*

5.8.5 Numerical Examples

We now present three more examples with less detail than the first two to illustrate the possible cases further. Let the sphere S be defined by $a = (2,0,0)$,

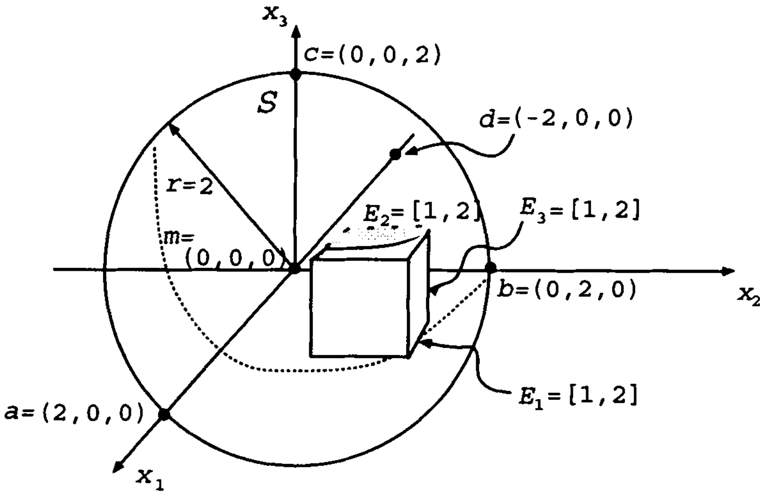


Figure 5.28: Box partly intersecting sphere

$b = (0,2,0)$, $c = (0,0,2)$, $d = (-2,0,0)$. Then the midpoint is easily found to be $m = (0,0,0)$ with the radius $r = 2$.

Example 3. The sphere S defined above is compared with the box E defined by $E_1 = E_2 = E_3 = [1,2]$. The configuration is shown in Figure 5.28. The approach in Subsec. 5.8.3 yields the distance comparing function in (5.39) as $F(x) = x_1^2 + x_2^2 + x_3^2 - 4$ which has the separable components $f_i(x_i) = x_i^2$. This gives $\square F(E) = [-1,8]$ which shows that S and E intersect.

The approach of Subsec. 5.8.3 yields $|V| = -16$, $\sigma = -1$ and minors $w_1 = w_2 = w_3 = 0$, $w_4 = -16$. The separable components of $\mathcal{D}$ are $f_1(x_1) = -16x_1^2 - 64x_1$, $f_2(x_2) = -16x_2^2$, $f_3(x_3) = -16x_3^2$.

If x_i is replaced by $E_i - a_i$ $(i = 1,2,3)$ in each case (use formula (5.38)), we find that $\square\mathcal{D}(a,b,c,d,E) = [-16,128]$. Again, the calculation shows that S and E intersect which confirms the previous result.

Example 4. The sphere S is as in Example 3 with E defined by $E_1 = E_2 = E_3 = [2,3]$. This results in the configuration in Figure 5.29. Proceeding as in Example 3 we get $\square F(E) = [-1,8]$ and $\square\mathcal{D}(a,b,c,d,E) = [128,368]$. In both cases the results show that S and E are disjoint.

Example 5. The sphere S is as in Example 3 with E defined by $E_1 = E_2 = E_3 = [2,3]$. This results in the configuration in Figure 5.30. Proceeding as in Example 3 we get $\square F(E) = [-4,-1]$ and $\square\mathcal{D}(a,b,c,d,E) = [-64,-16]$. Here both results show that E is included in S.

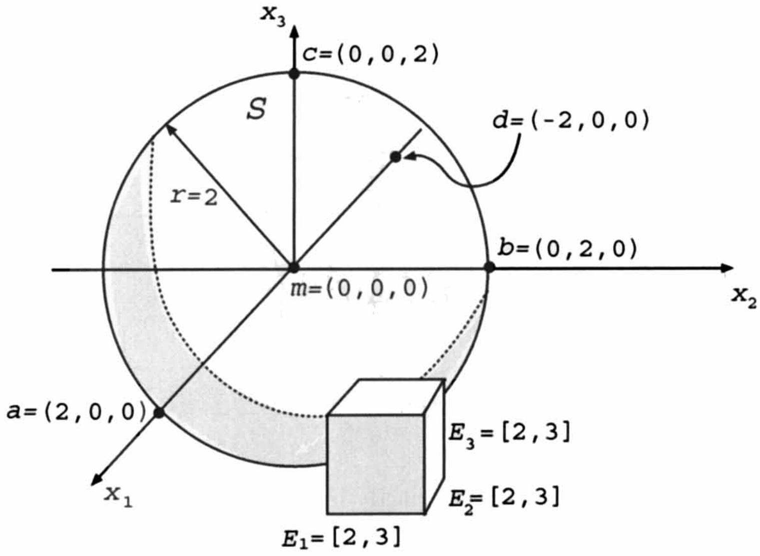


Figure 5.29: Box outside sphere

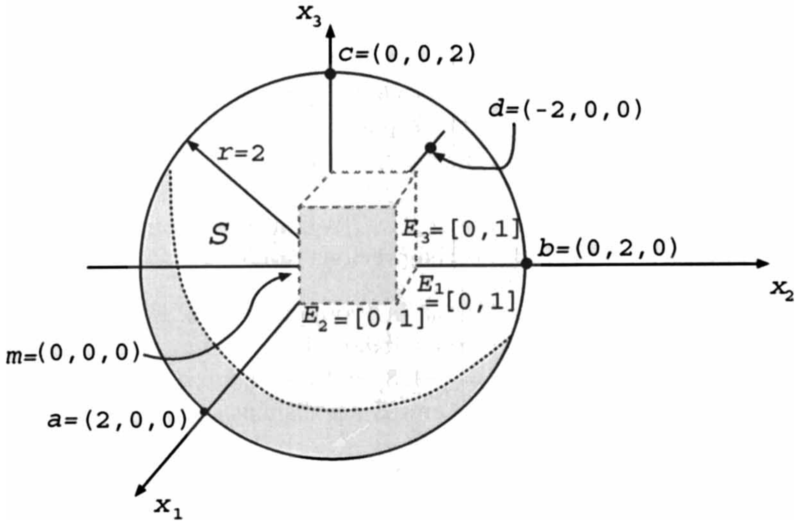


Figure 5.30: Box included in sphere

Chapter 6

The SCCI-Hybrid Method for 2D-Curve Tracing

6.1 Introduction

A hybrid method for plotting 2-dimensional curves, which are defined implicitly by equations of the form $f(x, y) = 0$ based on [225], is developed in this chapter. The method is extremely robust and completely reliable and consists of Space Covering techniques, Continuation principles and Interval analysis, and it is called *SCCI-hybrid method*. The space covering, based on iterated subdivision, guarantees that no curve branches or isolated curve parts are lost (which can happen if grid methods are used). The continuation method is initiated in a subarea as soon as is proven that the subarea contains only one smooth curve segment. Such a subarea does not need to be further subdivided which means that the computation is accelerated as far as possible with respect to the subdivision process.

The crux of the SCCI-hybrid method is the intense use of the implicit function theorem for controlling the steps of the method. The formulas which are required in this theorem are evaluated using interval arithmetic so that the application of the theorem gives logically and analytically correct results which are not contaminated by rounding or approximation errors. Although the implicit function theorem has a rather local nature, it is empowered with global properties by evaluating it in an interval environment. This means that the theorem can provide global information about the curve in a subarea such as existence, non-existence, uniqueness of the curve or even the presence of singular points. The information obtained from the theorem allows the above-mentioned control of the subarea and to decide how it is processed further, i.e. deleting it, subdividing it, switching to the continuation method or preparing the plotting of the curve in this subarea. The curves can be processed mathe-

matically in such a manner, that the derivation of the plotted curve from the exact curve is as small as desired (modulo the screen resolution).

Let an equation

$$f(x,y) = 0 \tag{6.1}$$

be given, where f is a smooth real function in two variables, x and y. The aim of curve or contour tracing is to graphically represent the set of real solutions, (x, y), of (6.1). The set of solutions of (6.1) is called the *contour* of the curve or also the *curve* itself which is implicitly defined by (6.1). Even if f is a polynomial, the curve can behave very badly with respect to a plotting or to a preceding computational phase. The curve need not be connected, it can have forks and other nonsmooth parts, and it can happen that there are isolated parts, that consist of one point only. This means that the numerical and graphical treatment of equation (6.1) can be nontrivial.

The most important tool for obtaining explicit representations of the contour is the implicit function theorem which is well-known in analysis. Its application is however rather restricted and only valid if, besides a few technical conditions, the curve locally admits an explicit representation. For example, the theorem does not work for singular points such as the point $(0,0)$ of the simple function $f(x,y) = xy$.

Standard traditional methods for numerical solutions of (6.1) are continuation methods, simplicial methods, homotopy methods and space covering methods. Surveys of the first three methods can be found in, for example, Zangwill-Garcia [279], Schwetlick [240], Rheinboldt [227], Allgower-Georg [8], Guddat [84]. Space covering methods are well-known in interval analysis. As far as we know, they were first applied to contour tracing as part of computer-aided geometric processing by Mudur-Koparkar [174]. Further approaches to contour tracing with interval methods can be found in Neumaier [178], Suffern-Fackerell [258], Duff [37], Snyder [253], Kearfott [126], Kearfott-Xing [130], Schramm [239], [22] and others. These papers, however, differ from ours either by the problem statement or the methods used. The affine arithmetic used by de Figueiredo-Stolfi [53] is a generalization of interval arithmetic appropriate for some contouring problems. Grid techniques are also commonly used when it is required to plot the curves (see, for instance, [251], [53]). Martin et al. [154, 155] compared the accuracy and the costs of several variants of hybrid methods, included are variants that use derivative information.

The article by Taubin [265] and references therein considers the specific problem of rendering implicit curves on *raster devices* which are rectangular arrays of domains called *pixels* having the same colour or shading. A variant of this problem is discussed in [267] where the colouring of the pixels contains information about the existence of solution points in the pixel.

Typical examples of curves that would pose problems for many standard methods include

$$f(x,y) = (x^2 + y^2 - 1)(x^2 + y^2 - 1 - \epsilon) = 0 \tag{6.2}$$

for small ϵ or

$$f(x,y) = (x+y)(x-y) = 0 \tag{6.3}$$

if the curves are defined implicitly as above and no other information such as "the equation defines two intersecting lines" or "the equation defines two close, but disjoint circles" is available. Continuation methods might overlook the outer circle when starting with the interior circle of (6.2) or might break down at the point $(0,0)$ with (6.3). Grid methods might also have trouble in identifying this point in (6.3) with sufficient accuracy.

We return to these prototype examples in Sec. 6.3. where we show that the SCCI-hybrid methods have no difficulties at all in discovering these situations and to plot curves with almost arbitrary accuracy.

The SCCI approach extremely robust and completely reliable and uses

- space covering,
- Moore's exclusion test and
- computationally verifiable global versions of the implicit function theorem

as basic principles.

In our approach, the area in which the contour has to be plotted is usually a rectangle and it will be recursively subdivided into sub-rectangles. A rectangle will not be subdivided further when this rectangle

(i) is proven not to contain any contour points or

(ii) is appropriate for applying the continuation method or

(iii) has reached a prescribed size which is appropriate for submitting the contour in the rectangle to a plotting operation or to a plotting stack. Frequently, these rectangles are called plotting cells.

The interval arithmetic proof of (i) that a rectangle does not contain any contour points is generally supported by the *exclusion test*. For this purpose one needs an *inclusion function*, $F(X,Y)$, of $f(x,y)$ as introduced in Ch. 2. This means that

$$f(x,y) \in F(X,Y) \quad \text{for any} \quad x \in X,\ y \in Y, \tag{6.4}$$

where X, Y are compact intervals which means that $X \times Y$ is a rectangle with edges X and Y. We return to appropriate inclusion functions with a more detailed development in Sec. 6.2. Now, the *exclusion test* can be applied to the rectangle $X \times Y$ which means to select an inclusion function F of f, to evaluate the interval $F(X,Y)$ and to check for

$$0 \notin F(X,Y). \tag{6.5}$$

If (6.5) is satisfied, there is no point (x, y) in the rectangle $X \times Y$ with $f(x, y) = 0$ because of (6.4). Hence (6.5) guarantees, that the rectangle $X \times Y$ cannot contain any contour points of f. A more sophisticated procedure can be found in Neumaier [178] who develops an inclusion function by using a mean-value formula and transforms the question of whether $0 \in F(X, Y)$ to the solvability of a linear homogeneous interval system. Neumaier calls this procedure a "generalized Gauss-Seidel method".

The most striking feature of the SCCI-hybrid method is a process which decides whether the current rectangle should be subdivided further or whether the continuation method can be successfully applied in the rectangle, cf. (ii). (The continuation method is the least expensive method for computing the path, hence the earlier it is applicable the faster the computation proceeds.) This decision is simply derived from the above mentioned implicit function theorem as soon as it is globally valid in the rectangle. Then the curve in the rectangle can be represented uniquely as a function $y(x)$ or as a function $x(y)$. Certainly, explicit formulas for $y(x)$ or $x(y)$ need not be developed, however, the knowledge that these explicit functions do exist means that the continuation theorem can be safely applied without worrying about pathological cases. In order to check the validity of the theorem, it suffices to evaluate the partial derivatives of f over the rectangle, cf. Sec. 6.2. The partial derivatives, computed so far, make it possible to draw conclusions about the monotonicity behavior of $y(x)$ or $x(y)$ without further computational costs.

The continuation method itself corresponds to predictor corrector steps. The predictor steps are immediately obtained from the partial derivatives already computed and the correction steps can be done with arbitrary accuracy using interval methods.

There is still a variety of further interval based resources, which seemingly have not yet been used in connection with contour tracing. These are mostly tools which, together with point methods, make it possible to collect global information about the curves in a rectangle. Examples of this are existence, nonexistence, connectedness (i.e., one path of the contour), entry and exit points of the contour w.r.t. the rectangle, existence or nonexistence of singular points. All these features can be obtained using simple tools of interval arithmetic such as inclusion functions of f, partial derivatives of f, and the interval Newton method (which also could be avoided if the user is afraid to use it).

Summing up: The main features of the SCCI method, that is, its robustness and complete reliability, stem from the interval tests that are incorporated and the relative speed stems from the procedural control gained from the global information. A disadvantage of the method is the high computational effort which is caused by the fact that the curve is not given explicitly.

In Sec. 6.2, we will first outline the SCCI method and the tools used. This is followed by a more detailed description of the various parts of the method. In Sec. 6.3, the application of the presented method to the two critical prototype

examples mentioned before is discussed.

6.2 The Parts of the SCCI-Hybrid Method

The SCCI-hybrid method is presented here as a complete method consisting of the following parts:

A. The subdivision process

B. The standard exclusion test

C. The determination of the global shape of the contour

D. The continuation method

E. The plotting

F. Consideration fo rounding errors.

Parts A, B, D and E are based on known material whereas part C contains new material. Each of the parts are described below. The main discussion will however focus on parts C and D because of the large influence of part C to part D. The other parts are only described in a level of detail necessary for the development of the overall method. It should also be noted that the parts are mostly independent of each other, except for the dependency of part C on part D. The subdivision process, the standard exclusion test and the plotting process will therefore only be briefly discussed allowing us to focus on the new material.

Let X_0, Y_0 be two compact intervals forming the search rectangle $Z_0 = X_0 \times Y_0$ and let the smooth function $f : Z_0 \to R$ be given. Our aim is to plot the contour which is defined by $f(x, y) = 0$. Recursively, we also have to deal with sub-rectangles Z of Z_0, i.e., $Z = X \times Y$ where $X \subseteq X_0$, $Y \subseteq Y_0$ are compact intervals. We look only for those solutions of the equation $f(x, y) = 0$ which lie in Z.

As mentioned, we first need an *inclusion function*, F of f, that is a function

$$F : I(Z_0) \to I(R)$$

which satisfies

$$f(x,y) \in F(X,Y) \quad \text{for all} \quad (x,y) \in X \times Y.$$

In order to obtain reasonable results, one should only choose inclusion functions which satisfy the condition

$$w(F(X,Y)) \to 0 \quad \text{as} \quad w(X,Y) \to 0. \tag{6.6}$$

For example, if

$$f(x,y) = x \sin y + y \cos x$$

then $F(X,Y) = X \sin Y + Y \cos X$ is an inclusion function that satisfies (6.6), where $\sin Y$ and $\cos X$ are the ranges of sin and cos over Y and X, resp. (Ranges of standard functions can be computed with almost all interval software packages available.)

In contrast, the function

$$F(X,Y) = X[-1,1] + Y[-1,1]$$

is also an inclusion function of f, since the values of sin and cos are in $[-1,1]$. But condition (6.6) is not satisfied: Set, for example, $X = 1$ and $Y = 1$. Then $w(X) = w(Y) = 0$. But $F(X,Y) = 1[-1,1] + [-1,1] = [-2,2]$ and has therefore width equal to 4.

If $w(X)$ and $w(Y)$ are small, say, not larger than $1/4$, the mean value form should be used as inclusion function instead of the natural interval extension since it has a higher convergence order than inclusions based on the natural interval extensions (cf. Sec. 2.12). The meanvalue form is defined as

$$F_m(X,Y) = f(c,d) + (X - c,\ Y - d)^T\ \nabla f(X,Y)$$

for $X \in I(X_0)$, $Y \in I(Y_0)$, where c, d are the midpoints of X and Y, respectively, and $\nabla f(X,Y)$ is an inclusion function for the gradient which should satisfy (6.6), see for example [212], or any other book of interval analysis.

For a better understanding of the SSCI-hybrid method we first give an informal and then a more detailed description.

ALGORITHM 19 *(The SCCI-hybrid method (for plotting the contour of the equation $f(x,y) = 0$ in the rectangle Z_0).)*

The input parameters are Z_0, an inclusion function F of f, and the plotting cell size. The method works with two lists, a waiting list, WL for the mathematical processing and a plotting list, PL, which contains the final plotting information.

Step 1. *Initialize the waiting list, WL, by entering Z_0 onto the list.*

Step 2. *If WL is empty, terminate the computation (and start the plotting). Otherwise, get the "next" rectangle from WL and denote it by Z.*

Step 3. *Apply the standard exclusion test to Z. If Z is discarded by the test, go to Step 2.*

Step 4. *Determine the global shape of the contour in Z as specified later. Then*

α) If it turns out that there are no contour points in Z, discard Z and go to Step 2.

β) *If Z has already plotting size (supplied with the global shape information), Z is put on the plotting list. Go to Step 2.*

γ) *If the information on the global shape of the contour is appropriate, the continuation method is initiated for Z. I.e., Z is replaced by plotting cells, which are put on PL. If the continuation method cannot reasonably be applied to some part of Z (Z included), denote this part by Z and go to Step 5.*

δ) *In the remaining cases, go to Step 5.*

Step 5. *Subdivide Z, put the two halves on WL. Go to 2.*

After the termination, the proper plotting procedure can begin, that is, the plotting list, PL, can be forwarded to the plotting device.

We now start with a detailed discussion of the individual parts which are needed in the overall method.

A. The Subdivision Process

The subdivision process is a known technique in many areas of numerical mathematics, optimization, CAGD, etc. It has already been applied in Sec. 2.13, hence a brief description is sufficient. In general the aim of this process is to subdivide a given area recursively into smaller pieces (mainly rectangles, cubes, simplices, etc.) until there is enough information available for the current sub-area to commence a proper investigation or treatment of the problem or parts of the problem in this sub-area (for instance, zero search, search for extremum points, contour sampling, etc.).

In our case, we start with the initial rectangle, $Z_0 = X_0 \times Y_0$. This rectangle is subdivided and each sub-rectangle $Z = X \times Y$ which occurs in this process has to pass several tests regarding the behavior of f over Z. When a test renders a definite answer (no curve patch in Z, or, curve passes through an edge, or, when it is promising to switch to the continuation method) or if Z has already reached a size predescribed by the user (plotting cell size), then this rectangle will not be subdivided further. Instead it will be submitted to the appropriate mathematical or procedural treatment (such as commencing the plotting or deleting the rectangle or commencing the continuation method). Otherwise, the rectangle Z is subdivided into 2 (or 4) equal-sized sub-rectangles. The subdivision bisects the longest edge. The two halves are put on the list WL to be processed as mentioned above. There are various possible proposals for ordering the rectangles on the list. Examples of the orderings are the various types of linear orderings and tree structure orderings. In our experience the following linear ordering has been found to be the best:

1. Get the *first* rectangle, Z, from the already initialized, nonempty list (in our case, the waiting list, WL) and perform the various tests and procedures on Z.

2. If Z has not yet been discarded or forwarded to some other treatment, Z is subdivided, and the two halves are inserted at the head of the list, i.e., the left half before the right half, and the lower half before the upper half.

3. (Optionally) Install a directed "adjacent" relationship structure, from left to right, from below to above.

The advantage of this kind of "last in - first out" principle as opposed to uniform subdivision is that one can roll up the process along desired directions. This facilitates the incorporation of optional data structures. Furthermore, uniform subdivision needs, in general, more storage.

The adjacency structure is thought of as an accelerating device, however, it does not have any influence on the mathematical performance. This means if it is already known that a box has curve points on an edge, this information can be carried over to the adjacent box which share that edge with the aid of the structure. We do not mention the details of how to implement an adjacency structure since they are not part of this method.

The neighbor finding techniques which are described in Ch. 3 of Samet [238] may also be used.

B. The Standard Exclusion Test

This test has already been described in Sec. 6.1. The test stated that no curve points can lie in $Z = X \times Y$ if $0 \notin F(X, Y)$ which means that Z can be removed from further processing. If $0 \in F(X,Y)$, then Z is sent to Step 4 (global shape determination), which is the next processing step.

C. The Global Shape of the Contour in a Rectangle $Z = X \times Y$

Information about the global shape of the contour in Z is obtained from a set of computer executable queries. The information obtained from the queries is used for further decisions as to how to continue the computations. The advantage of the global shape knowledge obtained by interval methods in contrast to sampling based information is that the knowledge is guaranteed to be correct.

In addition to the inclusion function F of f, we need inclusion functions F_x and F_y of the partial derivatives, f_x and f_y, of f, respectively. Again, it is reasonable to take the natural interval extensions of f_x and f_y as inclusion functions, such that condition (6.6) for F_x and F_y holds automatically. Returning to our former example, $f(x,y) = x \sin y + y \cos x$, we get $f_x(x,y) = \sin y - y \sin x$, $f_y(x,y) = x \cos y + \cos x$ as partial derivatives of f which means that the inclusion functions are $F_x(X,Y) = \sin Y - Y \sin X$ and $F_y(X,Y) = X \cos Y + \cos X$.

The queries which establishes the global information depend on
a corner sign check (signs of f at the corners of Z),
the values of F at Z,
the values of F at the edges of Z,
the values of F_x, F_y at Z,
the values of F_x, F_y at the edges of Z.

Since the global shape investigation is only executed when Z does not pass the standard exclusion test, we can assume that

$$0 \in F(X,Y) \tag{6.7}$$

in the sequel. We now distinguish several cases:

$$\text{I.}\quad 0 \notin F_x(X,Y),\; 0 \notin F_y(X,Y)$$

First, we consider the condition

$$0 \notin F_y(X,Y). \tag{6.8}$$

Since F_y is an inclusion function for the partial derivative f_y, condition (6.8) implies that $f_y(x,y) \neq 0$ for any point $(x,y) \in Z$. Therefore, the implicit function theorem is applicable, and provided, a point $(x_0, y_0) \in Z$ with $f(x_0, y_0) = 0$ exists, there is a unique function $h(x)$ with the property $y_0 = h(x_0)$ and $f(x, h(x)) = 0$ for all x belonging to some neighborhood of x_0.

Since $0 \notin F_x(X,Y)$, one also has that $f_x(x,y) \neq 0$ for any $(x,y) \in Z$. Therefore h can be globally extended to a strictly monotone function in a neighborhood of Z and is at least well-defined as long as $(x, h(x)) \in Z$. This *function then represents the contour* of $f(x,y) = 0$ in Z, and *there are no further contour points in* Z.

The derivative of h is

$$h'(x) = -\frac{f_x(x,h(x))}{f_y(x,h(x))}, \tag{6.9}$$

where the strict monotonicity of h can be seen directly.

One also can practically decide whether h is increasing or decreasing, since we know that the intervals $F_x(X,Y)$ and $F_y(X,Y)$ do not contain zero. Hence, h is *strictly monotonically increasing*, if these intervals have different signs, otherwise h is *strictly monotonically decreasing*. Fig. 6.1 shows a few samples of possible configurations. Fig. 6.2 shows a few configurations that cannot occur.

It remains to be settled whether there is a curve in Z at all i. e. it is possible to have situations as shown in Fig. 6.1c. In contrast to non-interval methods, it is very simple to decide: If there is any curve point in Z, the curve must have an entry and exit point in Z, since singular points are excluded in

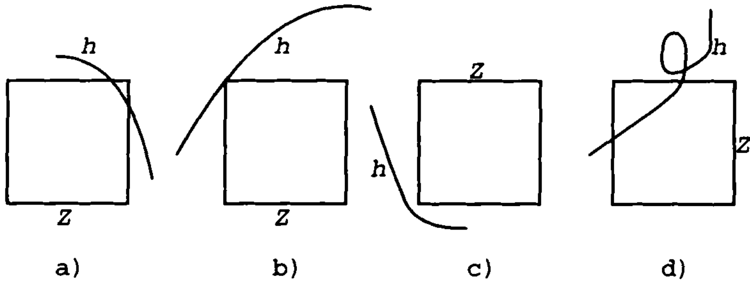


Figure 6.1: Sample configurations

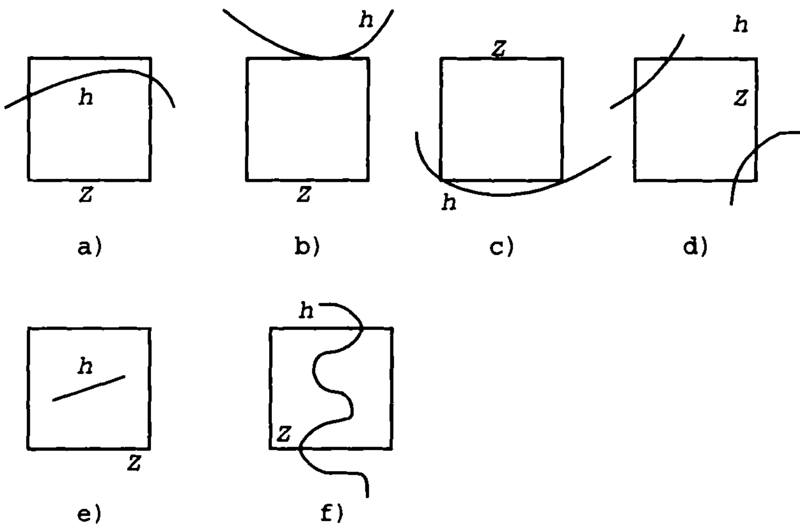


Figure 6.2: Impossible configurations

case I. (A singular point satisfies $f_x(x,y) = f_y(x,y) = 0$.) The entry and exit points can coincide, which then gives a corner, cf. Fig. 6.1b. Hence, in order to finally find out, whether Z contains contour points or not, the signs of the function values at the corners of Z are considered:

1. If the four signs are equal then Z cannot contain any contour points (and Z can be removed from further processing). Clearly any curve in Z has to enter Z somewhere. This can only happen if the function is 0 at some edge point. It would then follow that one of the adjacent corners had a positive function value and the other a negative value due to the monotonicity properties of f. The curve can not pass through a corner since the function value at such a corner would be zero (and by assumption the other three corners have function value zero) which contradicts the monotonicity properties.

2. Suppose that three signs are equal and the fourth one is zero. Then because of the derivative information, the corner with value zero is the only solution point of equation (6.1). Hence this point can be plotted (and entered as a degenerate plotting cell onto the plotting list PL), and Z can be discarded.

3. For the remaining possible sign distributions (3 positive and 1 negative, 2 positive and 2 negative, 1 positive and 3 negative, 2 zero, 1 positive and 1 negative, 1 zero, 2 positive and 1 negative, and 1 zero, 1 positive and 2 negative) the unique curve enters and leaves via different edges which are easy to determine from the sign distributions. All together, the ideal situation is reached for applying the continuation method since the entry and exit point can be computed with no danger of bifurcations or other singular points and there are no further isolated curve segments in Z. The entry point of the curve in Z is determined, where we think of the process as progressing from left to right. (This orientation is not really necessary for the mathematical part of the method, but it facilitates the algorithmic design.) With respect to this orientation, the entry point is either on the left edge of Z, or on the upper or lower edge of Z. I.e., it is on the upper edge, if $F_x(X,Y)$ and $F_y(X,Y)$ have the same sign (since then $h'(x) < 0$), or on the lower edge otherwise (since then $h'(x) > 0$). We use the interval Newton method to compute the entry point, if it is not already at a corner, since it has quadratic convergence under the given circumstances, as well as safe bounds for the entry point, cf. Ch. III. It is clear that the interval Newton method is applied to f restricted to the edge in question so that the zero search is one dimensional.

We restate the main steps:

a) The entry point is determined,

b) the process is switched to the continuation method,

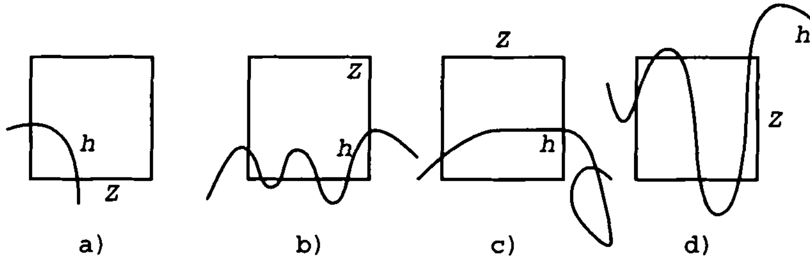


Figure 6.3: Possible configurations when $0 \in F_x(X,Y)$, $0 \notin F_y(X,Y)$

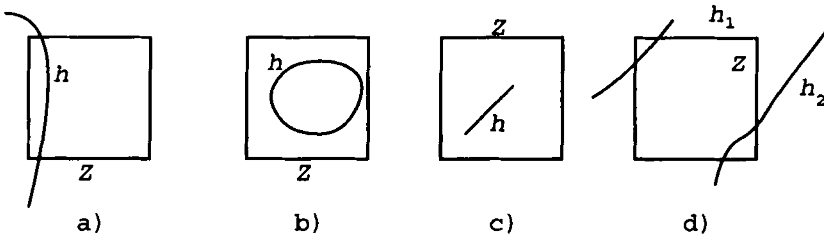


Figure 6.4: Impossible configurations when $0 \in F_x(X,Y)$, $0 \notin F_y(X,Y)$

c) Z is removed from further processing.

II. $0 \in F_x(X,Y)$, $0 \notin F_y(X,Y)$

The geometric interpretation of the assumption $0 \notin F_y(X,Y)$ is that, given any $x \in X$, there exists at most one $y \in Y$ with $f(x,y) = 0$. As in case I, the implicit function theorem is applicable if curve points exist in Z, so that the curve in Z can again be described by a function $h(x)$ where

$$h'(x) = -\frac{f_x(x,h(x))}{f_y(x,h(x))}.$$

Singular points do not exist in Z, hence there are no bifurcation or curve branches that begin or end in the interior of Z. However, it is possible that several branches of the contour go through Z with several entry and exit points. Fig. 6.3 shows a few examples of possible curves, and Fig. 6.4 shows some impossible configurations.

We will briefly explain why the configuration shown in Fig. 6.4d is impossible. We first denote the two branches of h by h_1 and h_2. Then we have to distinguish between two principally different cases (further cases are slight variations of the two main cases):

(i) The function values of f below h_1 are positive (precisely, $f(x,y) > 0$ if $(x,y) \in Z$ and $y < h_1(x)$) and the function values above h_2 are negative. Since f is continuously differentiable, each line leading from the negative part to the positive part has to pass a point (x,y) with $f(x,y) = 0$. There are no such points and it follows that the situation described can not occur.

(ii) The function values of f in the region between h_1 and h_2 are all positive. This implies that the function values above h_1 are negative (because of $0 \notin F_y(X,Y)$) and the values below h_2 are negative which means, that $f_y(x,y) < 0$ for points $(x,y) \in Z$ with $y = h_1(x)$ because of the function values decrease in the y-direction and that $f_y(x,y) > 0$ for points $(x,y) \in Z$ with $y = h_2(x)$ because of the function values increase in the y-direction. Since $F_y(X,Y)$ contains all values $f_y(x,y)$ with $(x,y) \in Z$ and since $F_y(X,Y)$ is an interval and therefore convex, $F_y(X,Y)$ contains 0, which is a contradiction. Hence the situation described here can not occur as well.

Two things are learned for the design of the program from this:

(i) The left edge of Z as well as the right edge contains as most one curve point each.

(ii) The lower and the upper edge of Z can contain more than one curve point each. But if a curve branch leaves the upper [lower] edge (seen in direction from left to right), another branch can enter only on the upper [lower] edge.

Based on the information gleaned so far, the processing of the rectangle Z will be as follows (and we again think of a curve parametrization from left to right):

Case 1. The signs of the function values at the corners of the left edge of Z are different.

Then the curve enters Z at this edge at exactly one point. This point is determined by the interval Newton method, except when the point is already a corner. Z is then sent to the *continuation method* which is applied until the curve leaves Z (for the first time). If the curve leaves Z at a point with x-coordinate $x' \in X = [x_1, x_2]$, that part of Z with x-coordinate $x \in [x_1, x']$ need not be processed further, since it cannot have another curve separated from the first. Hence, Z has to be replaced by the sub-rectangle, $Z' = X' \times Y$, where $X' = [x', x_2]$. If $x' = x_2$ already, Z' can be dropped. If Z' is sent back to the waiting list, WL, it is beneficial to send the information already available along with Z', that is,

α) $0 \notin F_y(X',Y)$.

β) $f(x', y_1) = 0$ or $f(x', y_2) = 0$ (but not both), where $Y = [y_1, y_2], x' < x_2$.

γ) there is no further entry point of the curve on the left edge.

It can also happen that when X is reduced to X' the condition $0 \in F_x(X, Y)$ is changed to $0 \notin F_x(X', Y)$. In this case Z' satisfies the conditions for case I, and is *returned* to this case.

Case 2. The signs of the values of f at the corners of the left edge of Z are equal (only "+" and "-", but not 0 are possible).

In this case, no curve enters through the left edge of Z, and, it can only possibly enter through the upper or lower edge. Let

$X_u = (X, y_2)$ be the upper edge of Z,

$X_l = (X, y_1)$ be the lower edge of Z.

Then, one evaluates F over X_u and X_l in the following manner:

(i) If $0 \notin F(X_l)$ and $0 \notin F(X_u)$, no curve points are on these two edges, hence no curves enter Z, and Z can be *discarded*.

(ii) If $0 \notin F(X_l)$, $0 \in F(X_u)$ only X_u might have entry points. It is reasonable to separate the easy from the more involved cases. An involved behavior is, for example shown by

$$h(x) = y_2 + (x + \epsilon - x_1)^2 \sin \frac{1}{x + \epsilon - x_1} \quad \text{if} \quad x + \epsilon - x_1 \neq 0,$$
$$h(x) = 0 \text{ otherwise},$$

where $0 < \epsilon << x_2 - x_1$. In this case it is almost impossible to get an overview of the global shape of h only using numerical tools.

We therefore distinguish between the following cases:

a) If $0 \notin F_x(X_u)$, there can be at most one curve point on X_u which can be shown by checking the signs of f at the corners of X_u:

a1) If the function values at the corners have the same sign (0 is again not possible), then no curve goes through X_u, and Z can be *discarded*.

a2) If the function values at the corners have different signs, then there is exactly one curve branch going through Z, it enters through X_u and leaves through the right edge of Z. This means that Z behaves nicely and Z is sent to the *continuation method*.

b) If $0 \in F_x(X_u)$ then the curve can cross X_u from zero to an infinite number of times. It would certainly be theoretically possible to determine the exact number of crossing points if there is only a finite number of them (this could be done first, using the interval Newton method for separating them, and second, by showing with

uniqueness tests, that in each separated area there is exactly one solution point). It seems, however, more reasonable just to proceed with ***subdividing*** Z (and to store the actual data for keeping the computational costs low) due to the many situations that might occur.

(iii) If $0 \in F(X_l)$, $0 \notin F(X_u)$, proceed analogously to (ii). For example, swap X_l and X_u in (ii), to get (iii).

(iv) If $0 \in F(X_l)$, $0 \in F(X_u)$ then there are again too many entry and exit points possible. Hence we will turn to the continuation method only if some monotonicity properties can be established:

a) If $0 \notin F_x(X_l)$, $0 \notin F_x(X_u)$ and

a1) if the function signs at the 4 corners of Z are equal then there are no curve points in Z, and Z can be ***discarded***,

a2) if the function signs at the corners are not equal, two different edges of Z are involved with a unique entry point and a unique exit point of the curve. The entry point is determined using the interval Newton method and then a switch is made to the ***continuation*** method.

b) If $0 \in F_x(X_l)$ or $0 \in F_x(X_u)$ it might be possible to find the curve points on X_l and X_u. This does not seem reasonable due to the possibly large computational effort involved and we therefore recommend that the process is continued by ***subdividing*** Z.

III. $0 \notin F_x(X,Y)$, $0 \in F_y(X,Y)$

This case is analogous to case II. One may swap the x and y coordinates in order to transform this case to case II.

IV. $0 \in F_x(X,Y)$, $0 \in F_y(X,Y)$

This is the case where all can happen, starting from a reasonable curve behavior as in case I (cf. Fig. 6.5a) to the occurrence of not connected curves or singular points (forks, crossings, the start of new paths), cf. Fig. 6.5 b-g. The reason that reasonable curve behavior can also be included in this case is due to the overestimation of the range of the derivatives, f_x and f_y, by their inclusions, F_x and F_y. Therefore it might happen that $0 \in F_x(X,Y)$ is computed even though $f_x(x,y) \neq 0$ for all $(x,y) \in Z$.

It certainly would be possible to investigate further details computationally, such as the existence of singular points in Z (just apply interval Newton method to the point equation system $f_x(x,y) = f_y(x,y) = 0$ in Z and check whether the solution satisfies $f(x,y) = 0$). There are, however, still too many different

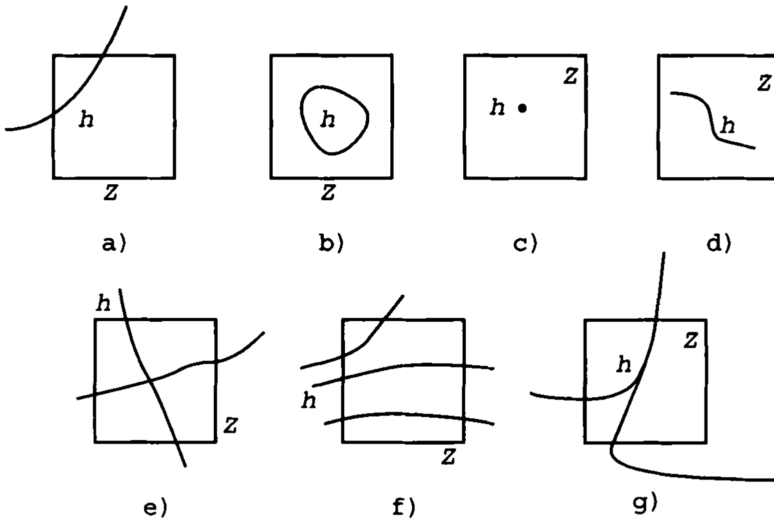


Figure 6.5: Possible configurations when $0 \in F_x(X,Y)$, $0 \in F_y(X,Y)$

situations possible such that it is best to reduce the size of the rectangles, until one obtains one of the more convenient cases in I to III since smaller rectangles have a smaller chance of containing awkward situations. Even so, it is clear that a singular point in a rectangle remains a singular point independent of how small the rectangle is.

Summing up, the rectangle Z is sent back to be ***subdivided*** further.

D. The Continuation Method

The continuation method would be accepted as the most reasonable and efficient method to design contours, if one did not have to deal with the uncertainty of whether there are perhaps easily overlooked disconnected curve pieces "near" to the smooth path one is just plotting. In this connection the question also arises how to jump from one path to another if they are disconnected. See, for example, [84]. Both of the two disadvantages just mentioned are meaningless when using the SCCI-hybrid method because of the availability of global information. Accordingly, we switch from the subdivision process to the continuation method as soon as the global information indicates that the above-mentioned flaws are excluded. Nevertheless, in some cases we switch back to the subdivision process, i.e., when the computational effort for going ahead with the continuation method would be large.

Hence, if a rectangle, Z, is destined for the continuation method it is guaranteed that there is only one path in Z. The path may fall into several disconnected pieces *in* Z, which does not matter, since these pieces are then connected by the edge of Z so that they are always under control, cf., for example, Fig.

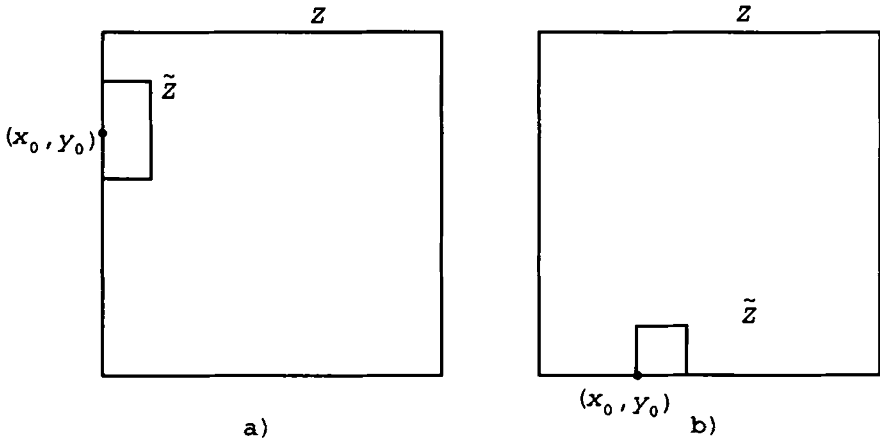


Figure 6.6: Plotting cells $\tilde{Z}$

6.3b or 6.3d.

The continuation process itself is simple and easy, and comparable to marching methods: We start with the entry point and we continue to keep track of the path, until it leaves Z, and we check whether there is a re-entry of the curve (on the same edge). The track keeping is nothing but the covering of the path with plotting cells, one after the other, from left to right. Then the curve is generally only interpolated in the plotting cells, in our case, linearly.

Thus, let $(x_0, y_0) \in Z$ be the entry point as well as the starting point. Since the whole method progresses from left to right, (x_0, y_0) either lies on the left edge of Z, or on the upper or lower edge of Z. In the latter cases, there is no curve point to the left of (x_0, y_0). Let d be the width of a plotting cell, prescribed by the user. The value of d can be fixed, or variable, or can have different values in the two coordinate directions. We choose d as a fixed constant.

Let

$$\begin{aligned} \tilde{X} &= [x_0,\ x_0 + d] \cap X, \\ \tilde{Y} &= [y_0 - d,\ y_0 + d] \cap Y, \\ \tilde{Z} &= \tilde{X} \times \tilde{Y}, \end{aligned}$$

cf. Fig. 6.6.

The area $\tilde{Z}$ is the rectangle where the plotting in form of a linearization is intended. (This paper does not discuss other interpolation types, since they have no influence on the SCCI method itself.) As the continuation method is developed from left to right, we need not consider the area left of x_0. But we have to consider the area below and above y_0, since the curve can move down as well as up.

In order to get a reasonable linear interpolation, we just look at the exit

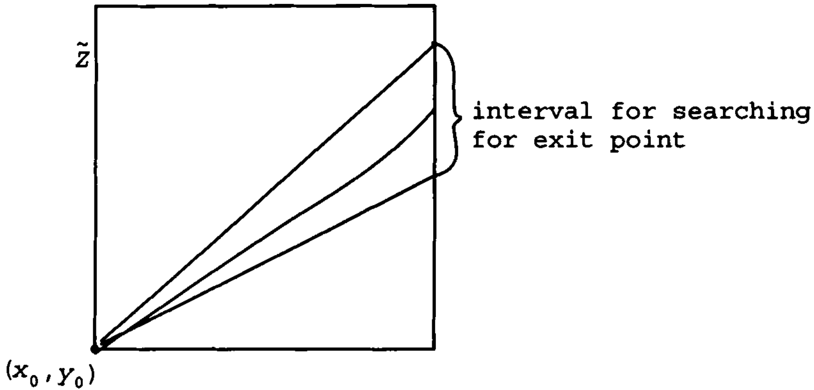


Figure 6.7: Cone containing contour

point of the curve from $\tilde{Z}$ which then will be the second point for the linear interpolation, if appropriate. If there are only a few further entry and exit points, one can continue to interpolate, or one can colour the box black, if there are too many.

Now, first of all, the path in $\tilde{Z}$ will be included in a cone with vertex (x_0, y_0). The cone is obtained from the partial derivatives. (A similar idea was already used by Hoffmann [107] who used Lipschitz constants. This was, however, difficult to realize since methods for determining the Lipschitz constants required were not available. - Note that our method delivers a way to determine a Lipschitz constant implicitly and almost automatically.) Since the contour lies in the cone, the contour leaves $\tilde{Z}$ also within the cone. Hence it is in general not too difficult to localize the (first) departure point, cf. Fig. 6.7.

Since we switched to the continuation method from the part of the algorithm where the global shape of the contour in Z was explored the following conditions are satisfied:

$$0 \notin F_x(\tilde{X}, \tilde{Y}) \quad \text{or} \quad 0 \notin F_y(\tilde{X}, \tilde{Y}). \tag{6.10}$$

These conditions are carried over from Z to $\tilde{Z}$ because of the inclusion $\tilde{Z} \subseteq Z$.

Since (6.10) holds not only for the initial plotting cell we have considered so far, but also for all other plotting cells lying in Z, we will no longer distinguish between the different cells and denote them all by $\tilde{Z} = \tilde{X} \times \tilde{Y}$. It is not essential whether $\tilde{Z}$ is now the initial cell in Z or not. Accordingly, we let (x_0, y_0) be the entry point of the curve in $\tilde{Z}$. Note that (x_0, y_0) lies on the left edge of $\tilde{Z}$ or on the upper or lower edge of $\tilde{Z}$ due to our left to right orientation.

We now distinguish between two cases:

I. $0 \notin F_y(X,Y)$

As we have seen above the curve in Z can be completely described by an explicit function $h(x)$ as long as $(x,h(x)) \in Z$. In general we do not know this function, but we know the derivative

$$h'(x) = -\frac{f_x(x,h(x))}{f_y(x,h(x))}$$

as discussed in (6.9). This is enough to get the cone, and that $h(x)$ is unknown does not matter as we will see below.

An inclusion function for h' could be

$$H'(\overline{X},\overline{Y}) = -\frac{F_x(\overline{X},\overline{Y})}{F_y(\overline{X},\overline{Y})} \text{ for } \overline{X} \in I(X), \overline{Y} \in I(Y).$$

Although H' is an inclusion function for $h'(x)$, it is reasonable to carry along $\overline{Y}$ as a second (interval) variable in order to have a better control of the bounds of $h(x)$ such as $h(x) \in \tilde{Y}$. This helps to obtain tighter inclusions. This means that $H'(\overline{X}, \overline{Y})$ is an inclusion of $h'(x)$ as long as $(x,h(x)) \in \overline{X} \times \overline{Y}$ which is exactly the situation we have to deal with in the plotting cells $\tilde{Z}$.

The inclusion function H' is convenient to work with. In many cases, it would however be better to find a direct inclusion of $h'(x)$ by looking first for an appropriate expression of $h'(x)$ (in terms of x and $h(x)$) and then to choose the natural interval extension of this expression by replacing x by $\overline{X}$ and $h(x)$ by $\overline{Y}$. The reason is that different expressions for one and the same function in general lead to different inclusion functions as discussed in Sec. 2.8. Experience has shown that the function H' as defined above does not produce the tightest bounds.

The inclusion function H' (or an improved one) is now used to define a cone which covers the curve in the plotting cell $\tilde{Z}$. The cone is

$$C = \{(x,y) : x \geq x_0,\ y \in y_0 + H'(\tilde{X},\tilde{Y})(x - x_0)\}.$$

The point (x_0, y_0), which is the entry point of the curve in $\tilde{Z}$, is the vertex of C, and C is built up with the slopes of all tangents of the graph of h in Z.

Hence, by the meanvalue theorem, any point of the contour that lies in $\tilde{Z}$, lies in C, cf. Fig. 6.8.

Let

$$\begin{aligned} c_{\max}(x) &= y_0 + (\max H'(\tilde{X},\tilde{Y}))(x - x_0), \quad x \geq x_0 \\ c_{\min}(x) &= y_0 + (\min H'(\tilde{X},\tilde{Y}))(x - x_0), \quad x \leq x_0. \end{aligned}$$

Then $c_{\max}$ and $c_{\min}$ describe the upper and the lower boundary of the cone. We have 2 cases:

(i) Both half-lines, $c_{\max}$ and $c_{\min}$ intersect the right edge of $\tilde{Z}$, cf. Fig. 6.8a, for instance. Then, the curve leaves $\tilde{Z}$ on this edge between the two

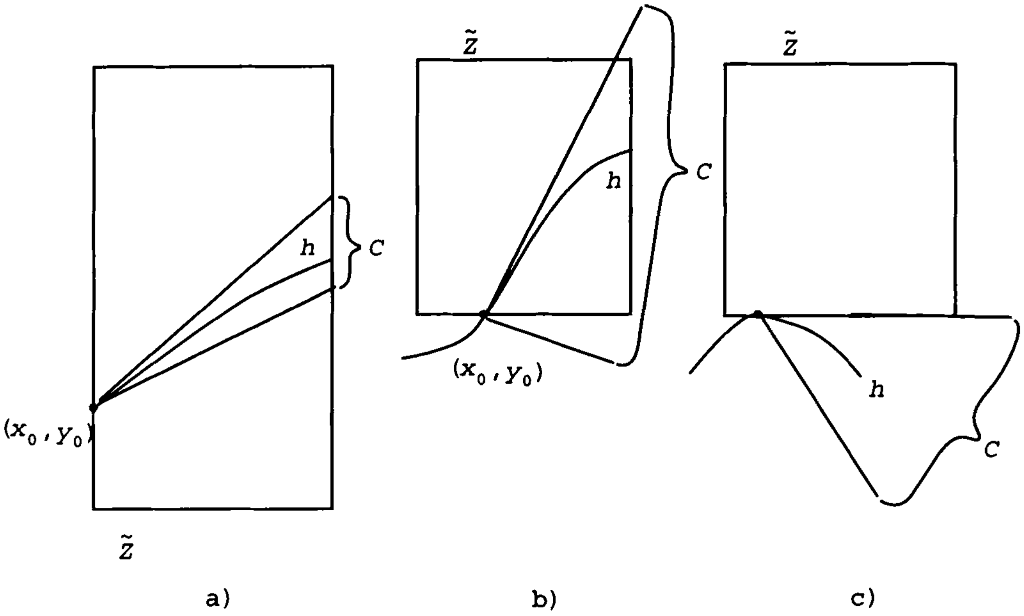


Figure 6.8: Possible cones

intersection points. Simply, apply the interval Newton method to the function $f(\bar{x}_0, y)$ with $\bar{x}_0 = \min(x_0 + d, x_2)$ as a function in the variable y on the interval bounded by the intersection points in order to determine the (unique) solution point, say $(\bar{x}_0, \bar{y}_0)$. This point will then be the initial point for the continuation step in the next plotting cell or the exit point of the curve out of Z, if $\bar{x}_0 = x_2$. If this exit point has, in fact, been reached, the processing of the curve in Z is finished, all the interpolation points for the curve or their plotting cells are on the plotting list, and Z can be removed from further processing.

(ii) At least one of the half-lines $c_{\max}$ or $c_{\min}$ intersects with the upper or lower edge of Z, cf. for example, Fig. 6.8b. The chance for a simple continued " marching" as was the case with case (i) has already decreased (theoretically, there could be still an infinite number of curve entries or exits on the upper or on the lower edge), however, one could be lucky and one should therefore exhaust all possibilities for recognizing a convenient plotting behavior if such a behavior is present at all. Otherwise we have to deal with a plotting crash (black box colouring) or to plot by sampling as most non-interval based algorithms do.

We consider the most extensive case only, that is, $c_{\max}$ goes through the upper edge of $\tilde{Z}$ and $c_{\min}$ goes through the bottom edge of $\tilde{Z}$, cf. Fig. 6.9.

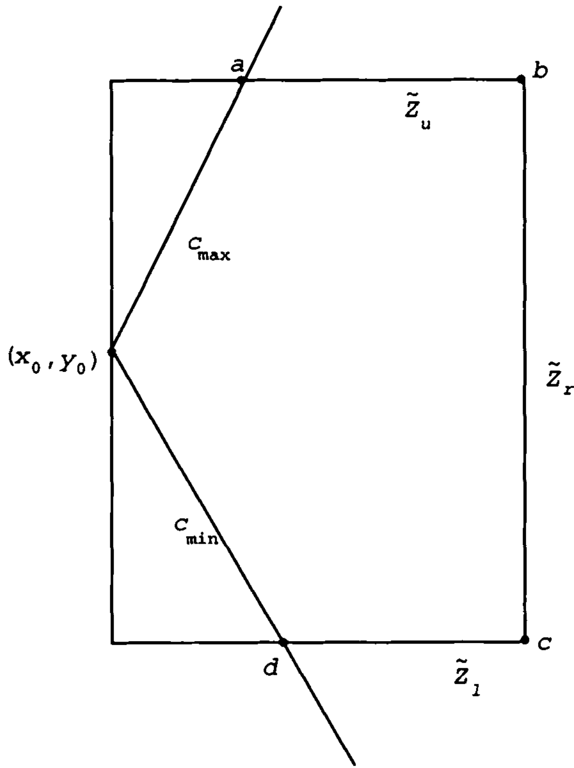


Figure 6.9: The general case for a cone

The other cases that occur are when $c_{\max}$ and $c_{\min}$ both go through the upper edge of $\tilde{Z}$, or both go through the lower edge, or $c_{\max}$ or $c_{\min}$ (but not both) go through the right edge of $\tilde{Z}$. These variants can easily be derived from the most general case from which one may learn how to incorporate the information about the upper and lower edge. The case that $c_{\min}$ coincides with the lower edge or that $c_{\max}$ coincides with the upper edge should also be clear.

We return to the general case sketched in Fig. 6.9. We only need to consider those parts of the edges that lie in $\tilde{Z}$. Hence the edges to be considered are, cf. Fig. 6.9,

$\tilde{Z}_u$ (with endpoints a and b) as that part of the upper edge of $\tilde{Z}$ that lies in C,

$\tilde{Z}_r$ (with endpoints b and c) as that part of the right edge of $\tilde{Z}$ that lies in C, and

$\tilde{Z}_l$ (with endpoints d and c) as that part of the lower edge of $\tilde{Z}$ that lies in C.

As mentioned before we wish to exhaust all reasonable possibilities to recognize a convenient and true plotting behavior. For this reason, the following tests are performed.

1. If $0 \notin F_x(\tilde{Z})$, there is a unique exit of the curve through one of the three edges, $\tilde{Z}_u$, $\tilde{Z}_l$ or $\tilde{Z}_r$. The edge, which has the exit, is easily determined by comparing the function values at the points a, b, c and d. The exact position is then found by the interval Newton method. Linear interpolation suffices for plotting.

2. If $0 \notin F_x(\tilde{Z}_u)$ and $0 \notin F_x(\tilde{Z}_l)$, there is also a unique exit as in Test 1. One proceeds as in Test 1.

3. If $0 \notin F(\tilde{Z}_u)$ and $0 \notin F(\tilde{Z}_l)$ there is a unique exit of the curve through $\tilde{Z}_r$. Proceed as in Test 1.

4. If $0 \notin F(\tilde{Z}_l)$ and $0 \notin F_x(\tilde{Z}_u)$, there exists a unique exit of the curve through $\tilde{Z}_u$ or $\tilde{Z}_r$. Proceed as in Test 1.

5. If $0 \notin F(\tilde{Z}_u)$ and $0 \notin F_x(\tilde{Z}_l)$. there exists a unique exit of the curve through $\tilde{Z}_l$ or $\tilde{Z}_r$. Proceed as in Test 1.

If these 5 tests are not passed successfully, one can make a last attempt and try the following:

If $0 \in F(\tilde{Z}_u)$ and $0 \in F_x(\tilde{Z}_u)$ bisect $\tilde{Z}_u$ iteratively to a depth of 3 or 4 obtaining 8 or 16 subintervals of $\tilde{Z}_u$. If the following condition is satisfied for each subinterval I_ν,

$$0 \notin F(I_\nu) \quad \text{or} \quad 0 \notin F_x(I_\nu), \tag{6.11}$$

then each I_ν has at most one intersection point with the curve, and the signs of f at the endpoints of I_ν clarify matters.

An analogous partition is done if $0 \in F(\tilde{Z}_l)$ and $0 \in F_x(\tilde{Z}_l)$. If the subinterval condition (6.11) is satisfied for the appropriate subinterval, one has sufficient information for the plotting (either linear interpolation, if points on different edges need to be connected, or quadratic or piecewise linear interpolation, if points on the same edge need to be interpolated). Note, there can be no crossings or saddlepoints. Furthermore, if the curve leaves via one edge, a re-entry is only possible via the same edge. Further, if $\tilde{F}(b)$ and $\tilde{F}(c)$ have the same sign, the curve will finally leave through $\tilde{Z}_u$ or $\tilde{Z}_l$, otherwise through $\tilde{Z}_r$.

If the plotting cells are sufficiently small and if more than 4 entry or exit points lie on at least 3 different edges, it makes sense to colour the whole cell or parts of it black (for example, the rectangle hull or convex hull of the points).

If all these efforts for assigning a convenient plotting behavior to the curve in $\tilde{Z}$ have been without success (practically, this will never occur, and we discuss this point only, in order to have a strategy suggestion for the worst-case enthusiasts), we terminate the continuation process (one does not know, where to continue). It is best to proceed as follows:

1. Discard that part of Z, which is to the left of $\tilde{Z}$, since the continuation method already worked fine there.

2. The remaining part has still to be processed and is then destined for the subdivision part. The continued subdivisions will generate smaller rectangles, where the continuation method can be again applied partially, or until one again winds up with the necessity of colouring some plotting cells black (but then not being forced "to continue" the curve from this cell).

II. $0 \notin F_x(X, Y)$

This condition means that to each $y \in Y$ at most one $x \in X$ exists with $f(x, y) = 0$. This means that if there is a contour in $Z = X \times Y$, it is unique w.r.t. the x-coordinate. Therefore, the application of the continuation method is still possible, but not from left to right. It is then best to swap the two coordinate directions and to apply the procedure as discussed under case I, that is the case $0 \notin F_y(X, Y)$. Then it is, in fact, possible to use the left-to-right-trend, cf. Fig. 6.10.

In order to have no plotting discontinuities at the connection points with other cells, it is reasonable to have a bookkeeping of the swapping incorporated in the overall data structure, if it is used at all.

If the continuation process is interrupted due to an involved plotting behavior, cf. the discussion in the former case, one has to re-swap the part which has not yet processed for further treatment.

Remark. The exit points will not be determined exactly when the interval Newton method is applied. Instead they will only be enclosed by some bounds. For example, if the exit is on the right edge of $\tilde{Z}$, we get bounds like

$$\overline{y}_l \leq \overline{y}_0 \leq \overline{y}_u$$

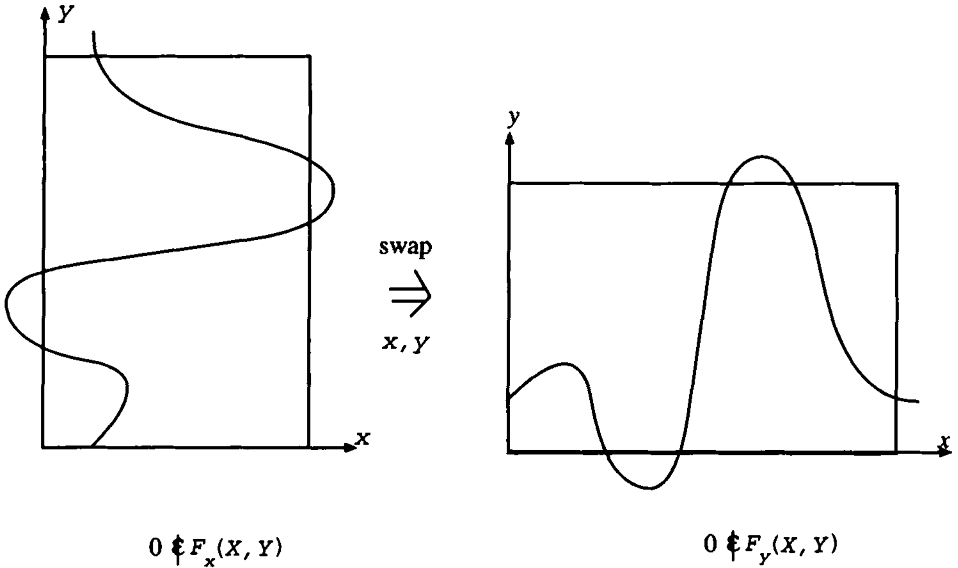


Figure 6.10: Swapping for left-to-right trend

with a prescribed maximum accuracy for the bound distance $\overline{y}_u - \overline{y}_l$. Thus, for the next plotting cell, the cone will not have a sharp vertex but only a flat vertex such as

$$C = \{(x, y) : \; y \in [\overline{y}_l, \overline{y}_u] + H'(\tilde{X}, \tilde{Y})(x - \overline{x}_0), \; x \geq x_0\},$$

where $\tilde{Z} = \tilde{X} \times \tilde{Y}$ already denotes the "next" cell, and the entry point is $\overline{y}_0$. The exit point of the curve from $\tilde{Z}$ is then again to be searched at one of the edge parts of $\tilde{Z}$ within C, and again with the interval Newton method. It is important to mention that there is no error accumulation by using the cones with the flat vertices as is the case when solving differential equations, since the tolerance $\overline{y}_u - \overline{y}_l$ for the determination of $\overline{y}_0$ is not affected by the inaccuracies accumulated in the former step. The accuracy depends solely on the tolerance predescribed for the bound difference which is the termination parameter for the interval Newton method and this can be made as small as possible. When the algorithm is implemented on a computer then inaccuracies due to rounding errors cause coarsening and this can also be handled with the cone idea and it does not matter, whether the inaccuracy comes from Newton or rounding errors.

E. The Plotting

It is difficult to make any suggestions how to draw the contours in the plotting cells. By this, we do not mean the technical process of drawing, which is

left to the plotting machine, but rather the decision as to how to approximate the contours in a cell by simpler lines which are then actually plotted. This decision depends on the size of the cells chosen and on the optical accuracy required. Some suggestions and hints are found in the literature, see, for example, Hoschek and Lasser [109], Suffern [257], Sutcliff [263]. Some of the main points of discussion seem to be,

> linear or quadratic approximation of the contour, if only two edge points are discovered or "sampled" in the cell, how to draw, if more than two edge points are found in the cell, when to colour the whole cell black.

We will not contribute to this discussion, since the choice of the type of approximations is not up to interval methods. Nevertheless, interval methods are able to support these decisions, because of the localization of the contour in the cell and in particular because of the localization of the points on the cell edges. This does not depend on the sampling, which is an uncertain and statistically completely unreliable procedure since contour points can easily be lost, but on the interval recipe, which can be made as reliable as necessary.

The global information part about the contour in the plotting cell that we are able to bring out of the interval theory, may include (besides the basic knowledge that 0 lies in $F(Z)$, where Z is the current cell) the computation of

> the values of F_x and F_y at Z,
> the values of F at the corners and edges of Z,
> the values of F_x and F_y at the edges of Z,
> interval Newton based knowledge
>> (number of zeros of f on each of the edges, zeros of the gradient, ∇f, in Z (if there is no such zero in Z, no singular points like bifurcations, crossings, or isolated points or contour pieces lie in Z)).

The geometric meaning of these expressions is described in Sec. 2 and may be used directly to control and direct the plotting procedure. To unify all these issues is a combinatorial puzzle rather than a mathematical activity and it is up to the user's requirements which of these issues are worthy of being included. Therefore we restrict ourselves to give a few examples.

Before we do this, it is reasonable to distinguish between *touching points* and *crossing points* of the contour. By touching points, we mean a point where the contour only touches the edge, but remains locally on the same side of the edge, i.e., outside or inside the cell, on the same side of or on the edge. By crossing points, we mean that the contour enters the interior of the cell from the outside or vice versa. Generally, one can distinguish between the two cases computationally and figure out their number on each edge of the plotting cell, if the situation is not too involved.

We now give a few examples for the plotting:

1. Touching points which are also singular points, are plotted as points. Touching points which are not singular points are part of an interpolation procedure for curves outside or inside the cell, or perhaps on the edge itself, and are plotted at this overriding level.

2. If there are exactly two crossing points lying on two different edges and if $0 \notin \nabla F(Z)$, (where $\nabla F(Z)$ is an abbreviation for $(F_x(Z),\ F_y(Z))$), we plot a straight connection line between the two points.

3. If there are exactly two crossing points lying on the same edge and if $0 \notin \nabla F(Z)$, we do a rough search for a third point in the cell lying on the symmetry line between the two edge points and use piecewise linear approximation (resulting in a cone) or quadratic interpolation.

4. If there are exactly two crossing points and one inside touching point on the edges, and if $0 \notin \nabla F(Z)$, again a cone is plotted by connecting the touching point with each of the crossing points.

5. If there are exactly four crossing points on the edges and if $0 \notin \nabla F(Z)$, we have a case which may cause headache for users of the sampling strategy. In contrast, this case can be solved easily with the interval mode.

   Without loss of generality, we assume that $0 \notin F_y(Z)$. (Otherwise the x-and y-coordinates have to be swapped.) We have already seen that then the equation $f(x, y) = 0$ can be resolved for y in Z, such that the curve in Z can be represented by a function $y = h(x)$ with a derivative $h'(x) = -f_x(x, h(x))/f_y(x, h(x))$, cf. (6.9). This means that the slope can never reach an infinite value such that contours of the type shown in Fig. 6.4a) are excluded. We have further seen, that if the curve leaves the cell at the upper [lower] edge, a reentry can happen only on the same edge, cf. Fig. 6.3d, so that the configurations as shown in Fig. 6.4d cannot happen. This leads to following definite answer:

   Enumerate the four crossing points on the edges by increasing x-values, that is, z_1, z_2, z_3, z_4, when $x_1 < x_2 < x_3 < x_4$. Then the curve in Z is split into two isolated parts, where one part connects z_1 and z_2, the other one connects z_3 and z_4, cf. Fig. 6.11

   Hence, the plotting instructions are obvious. If the two points connected lie on different edges, connect them with a straight line, as also done in Ex. 2.

   Otherwise connect them with a piecewise straight line (cone) or a quadratic curve, as done in Ex. 3.

These few examples could show the great variety of cases that could be recognized by interval computations. The user has finally to decide yet, when

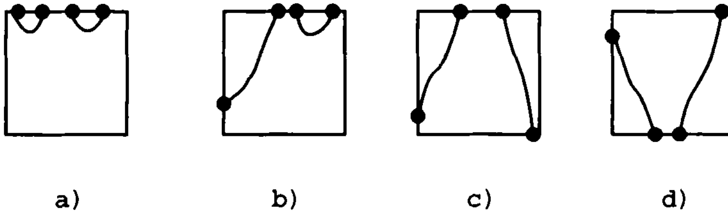


Figure 6.11: Two curve pieces

to start to colour the whole cell black. We do it, for example, if more than four crossing points or inside touching points are in the cell, provided $0 \notin \nabla(Z)$.

Up to now, we avoided the case $0 \in \nabla F(Z)$, since the computational effort is very high in order to reach decisions that are as definite as in the case $0 \notin \nabla F(Z)$. Certainly, one could continue the subdivision (for improving the mathematical precision and not for the plotting) until $0 \notin F(Z)$ or $0 \notin \nabla F(Z)$ is arrived at. For the remaining cells, a search for singular points (satisfying $\nabla f(z) = 0$ and $f(z) = 0$) would make sense and also to clarify which kind of singular point had been found. In our code, we chose a very simple procedure in the case $0 \in \nabla F(Z)$ being aware that we are confronted with a rare case: We just subdivide Z into 4 or 9 congruent sub-rectangles, and we colour a sub-rectangle, Z_i black when $0 \in F(Z_i)$, that is, if we cannot exclude that Z_i contains no curve points.

F. Consideration of rounding errors

Up to now, the development of the parts of the hybrid method took place in the space of reals, that is, in the real plane. If the calculations are performed in a floating point environment, the occurrence of rounding errors will falsify the results. The execution of the computations with machine interval arithmetic with its outward rounding helps to control the rounding errors as usual.

The inclusions which occurred above and which were checked whether they contain zero or not were primarily $F(X,Y)$, $F_x(X,Y)$, and $F_y(X,Y)$.

As an illustrative example, let us focus on the inclusion $F(X,Y)$. Let the machine interval arithmetic evaluation of this inclusion be $F_M(X,Y)$. Then the outward rounding implementation guarantees that $F(X,Y) \subseteq F_M(X,Y)$ holds. So if a computation results in $0 \notin F_M(X,Y)$, one knows that $0 \notin F(X,Y)$, and the discussions for the case $0 \notin F(X,Y)$ are applicable.

Conversely, if a computation results in $0 \in F_M(X,Y)$, it is likely that $0 \in F(X,Y)$ holds, but one cannot rely on it. But this does not matter, because not even the situation $0 \in F(X,Y)$ allows to conclude that 0 is in the range of the function $f(x,y)$ over the rectangle $X \times Y$. This was the reason for introducing the check of the signs at the corners of the rectangle in order

to get a definite answer. These corner checks have to be done now as well.

However, the evaluation of the function values at the corners is also subject to floating point errors. This means that for a corner (x_1, y_1) of the rectangle $X \times Y$ the true but usually unknown function value $f(x_1, y_1)$ is contained in an including machine interval, say $f_M(x_1, y_1)$. In most of the cases this interval will be strictly positive or strictly negative so that the true but unknown function value $f(x_1, y_1)$ will be of the same sign as its including interval. But what happens if, for example, $f_M(x_1, y_1)$ is equal to $[-10^{-3}, +10^{-4}]$? Practically, all can happen: A zero could be the corner exactly, it could lie outside the rectangle, and it could lie inside the rectangle.

In such cases it is reasonable to subdivide the rectangle as was done in case $0 \in F_x(X, Y)$ and $0 \in F_y(X, Y)$, cf. case IV of part C and to put the halves onto the list WL for further processing. But if the rectangle $X \times Y$ has already cell width one can either

(i) use data (like signs of the derivatives and signs of corners) of adjacent rectangles to get a decision about the site of the zero, which is frequently possible,

or

(ii) put it on the plotting list, PL, with a mark that indicates a suspected zero. Such a mark could be, for instance, different colouring of the cell or plotting a cross instead of filling the cell black. Another way is to plot the cell like a zero. In this case the interpretation of the contour should be so that no zero is lost in the plotting, but that not every plotted point is really a zero. Those problems arise only with functions which are very sensitive to evaluate, but nevertheless, such situations cannot be excluded.

A similar procedure could be provided for the two other inclusions, $F_x(X, Y)$ and $F_y(X, Y)$. One only has to keep in mind that the computational result that, for instance, 0 is contained in the machine interval arithmetic evaluation of $F_x(X, Y)$, means only that it is not proven that $F_x(x, y) \neq 0$ for all $x \in X, y \in Y$.

6.3 Examples

We present a few numerical and graphical results out of a larger collection of mainly ill-posed problems.

For each of the examples the input information provided is the function defining the contour implicitly, the function gradient, the function domain which is of interest, and the cell-widths for controlling the quality of the plotting. No other information is made available to the SCCI-hybrid method. If further information about the class of objects to be plotted would be provided the method probably could be replaced by a more effective one. For example

if the function is

$$f(x,y) = x^2 + y^2 - 1$$

then the fact that its zero set defines a circle of radius 1 centered at the origin is assumed to be completely unknown to the performance of the method. If such information were made available then there are many methods in the literature that will plot the circle much faster than the SCCI method.

The first example consists of two disconnected circles with a relatively low distance between them. Again, knowledge that it is two circles is not provided and would mean that any one of a number of circle generating algorithms could be used to generate the contours. In that case, the circles could touch, intersect or be disjoint and the circle generating algorithms would be able to plot the contours without any problems.

The second example consists of two straight lines crossing each other so that the contour has a double point. Similarly, only the function defining the line pair implicitly and the gradient are known. If it would be known that the objects which are plotted are straight lines any straight line generating algorithm could be used to draw the lines and the configuration of the lines (intersecting or not, parallel, etc.) would pose no problems. The cell-width in this example was chosen as 10^{-3} (in contrast to the other examples where 10^{-2} was chosen) in order to demonstrate that a small size of the cell-width has no negative influence to the plotting result as the average pixel size on a common screen is about 0.05cm. The robustness of the SCCI method is demonstrated by this parameter value and could be observed at the other examples of this section too. Several other contour tracing methods would have dissolution problems in such situations.

The third and fourth example show three straight lines crossing in a point which is once defined exactly and the other time defined within some artificially generated tolerance. As one can see the dissolution of the plotting result is very satisfactory.

The fifth example discusses a function which defines a curve with a crossing point where the last example shows a function which defines two circles which touch each other that is, they have the same tangent line in the critical point.

The reason for presenting these examples is that they clearly highlight how the SCCI method overcome the prototype difficulties encountered in contour tracing caused by various kinds of singular points, crossing points and disjoint, but close contours. We also note that the plotting domain for the input data was partially oversized in the examples in order to show that the method easily handles areas without contours.

A computer code for the SCCI method was written in PASCAL-XSC, the contours were drawn by GNU-PLOT.

In the following examples the statistical parameters are as follows:

Counter name	Counter Description
nbis	Number of bisections (excepting bisections done in Newton steps)
nfe	Number of function evaluations
ngrad	Number of gradient evaluations
nNBisect	Number of bisections done in the interval Newton algorithm
nNewton	Number of calls of the interval Newton algorithm
nNStep/nNewton	Average number of iterations at each call of the interval Newton algorithm
nplottinglist	Number of points in the plotting list

Two concentric circles with midpoint $(0,0)$ and radii 1.0 and $(1.2)^{1/2}$ are represented by the equation

$f(x,y) = (x^2 + y^2 - 1)(x^2 + y^2 - 1.2) = 0.$

The SCCI method was applied with the following input data:

Search domain: $Z = X \times Y, X = [-5,7], Y = [-6,9]$

Plotting cell width: 10^{-2}.

Statistics of the computation:

Counter name	Counter	Counter name	Counter
nbis	1027	nNewton	1988
nfe	7373	nNStep/nNewton	1.6
ngrad	5336	nplottinglist	1693
nNBisect	206		

It should be emphasized that the discretization of the circles is comparable to that which would have been achieved if the class of geometric objects, that is, circles, would have been available such that specialized circle algorithms could be used.

The contour consisting of two straight lines crossing each other in the point $(0,0)$ is represented by the equation

$f(x,y) = (x+y)(x-y) = 0.$

SCCI was applied with the following input data:

Search domain: $Z = X \times Y, X = Y = [-2,2]$

Plotting cell width: 10^{-3} (The width was made especially small in order to demonstrate the high dissolution ability of the method.)

Statistics of the computation:

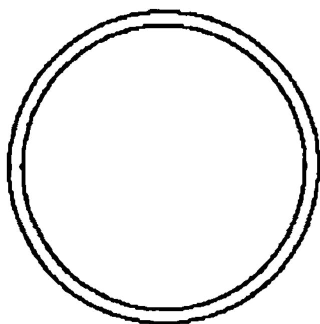


Figure 6.12: Two close co-centric circles

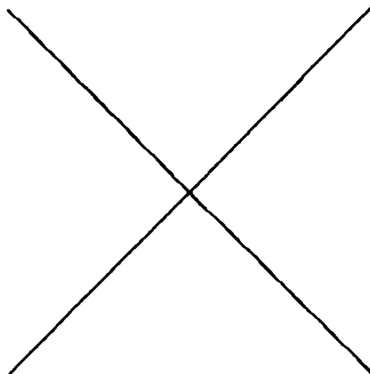


Figure 6.13: A cross with a singular point, small cell width

Counter name	Counter	Counter name	Counter
nbis	375	nNewton	11980
nfe	29445	nNStep/nNewton	2.3
ngrad	28268	nplottinglist	11848
nNBisect	0		

The result of the plotting is shown in Fig. 6.13. Again, if the complete geometric information would have been available that is, that the objects to be plotted are straight lines then the resulting plotting of appropriate methods would be comparable and certainly not significantly better.

In Figure 6.14 the contour defined implicitly by $f(x,y) = (y-x)(y+x)y = 0$ consisting of three lines crossing at a triple point $(0,0)$ is plotted.

The input for the plotting routine was a cell width of 10^{-2}. The plotting

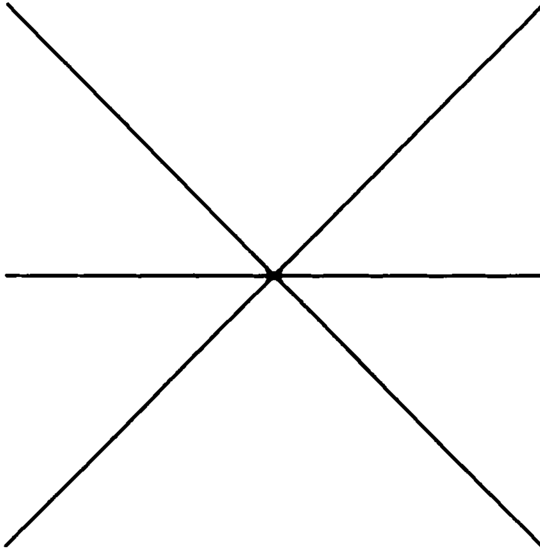


Figure 6.14: Plotting of a curve with a triple point

domain was $[-2,2] \times [-2,2]$.

The plotting statistics were

Counter name	Counter	Counter name	Counter
nbis	2555	nNewton	1616
nfe	8383	nNStep/nNewton	1.8
ngrad	6964	nplottinglist	2540
nNBisect	72		

In Figure 6.15 the contour defined implicitly by the equation $f(x,y) = (y-x)(y+x+p)(y-p) = 0$ consisting of three lines is plotted with the value of $p = 0.1$ This contour has an approximate triple point at $(0,0)$.

The input for the plotting routine was a cell width of 10^{-2}. The plotting domain was $[-2,2] \times [-2,2]$.

The plotting statistics were

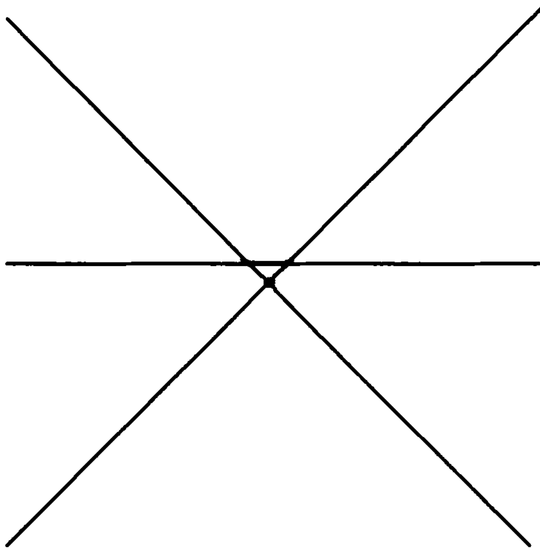


Figure 6.15: Plotting of a figure with an approximate triple point

Counter name	Counter	Counter name	Counter
nbis	1615	nNewton	1564
nfe	6798	nNStep/nNewton	1.7
ngrad	5281	nplottinglist	1984
nNBisect	120		

In Figure 6.16 the contour implicitly defined by the equation $f(x, y) = y^2 - x^2((1 + x)/(1 - x)) = 0$ is plotted. This contour has a crossing point at $(0, 0)$.

The input for the plotting routine was a cell width of 10^{-2}. The plotting domain was $[-2, 0.9] \times [-0.9, 0.9]$.

The plotting statistics were

Counter name	Counter	Counter name	Counter
nbis	349	nNewton	585
nfe	2143	nNStep/nNewton	1.8
ngrad	1692	nplottinglist	541
nNBisect	16		

In Figure 6.17 the contour defined implicitly by the equation $f(x, y) =$

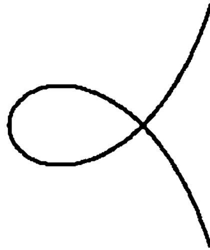


Figure 6.16: Plotting of a curve crossing point

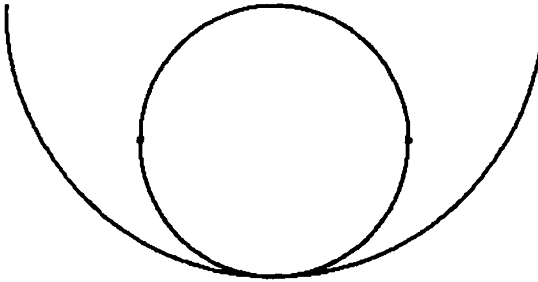


Figure 6.17: Plotting of a circle touching another circle

$(x^2 + y^2 - 1)(x^2 + (y-1)^2 - 4) = 0$ is plotted. This contour consists of a circle touching another circle at $(-1, 0)$.

The input for the plotting routine was a cell width of 10^{-2}. The plotting domain was $[-2, 2] \times [-3, 1]$.

The plotting statistics were

Counter name	Counter	Counter name	Counter
nbis	781	nNewton	1516
nfe	5075	nNStep/nNewton	1.6
ngrad	3804	nplottinglist	1522
nNBisect	66		

Chapter 7

Interval Versions of Bernstein Polynomials, Bézier Curves and the de Casteljau Algorithm

7.1 Introduction

Bernstein polynomials and Bézier curves are well known and widely used in computer-aided design (see for example [46, 15], although there are still ways in which they are being further developed (see [46, 109] for overviews). During the last years the extension to *interval Bernstein polynomials* and interval Bézier curves has also attracted the attention of researchers (cf. for example, [110, 243, 242]).

It is therefore reasonable to provide an introduction to this interesting area. A complete coverage of the area would fill another book, however, the key to the area is the basic interval arithmetic principles, tools and ideas which are explained in detail that lead from the point-view to the interval-view. If the reader has understood this introduction it is not too difficult for him to apply these principles to other geometric, algebraic or analytical situations.

In Sec. 7.2, we introduce some elementary concepts relating to the definition and manipulation of curves in the plane R^2 with emphasis on polynomial forms and Bernstein polynomials. These concepts are used in Sec. 7.3 to define sets of curves called interval curves with interval tools, i. e. interval Bernstein polynomials. The techniques for manipulating curves are similarly generalized to interval curves. Bézier curves and their extension to interval Bézier curves are considered in Sec. 7.4. The interval version of the de Casteljau algorithm

is explained in Sec. 7.5, and a new proof is provided showing that the interval version has similar properties with respect to subdividing the curve and creating new control points as the point version.

7.2 Plane Curves and Bernstein Polynomials

In this section we first introduce some elementary concepts relating to the definition and manipulation of curves in the plane R^2. These concepts are then used to define sets of curves called ***interval curves*** with interval tools. The techniques for manipulating curves are similarly generalized to interval curves.

We are particularly interested in interval curves useful in computer graphics, solid modeling as well as the other areas mentioned in the introduction to this monograph.

Further material on curves of interest for this section can be found in the excellent book by Farin [46]. Because of the aims of the monograph, only curves and interval curves in 2 dimensions, that is, plane curves will be considered. Most of the results are also valid for higher dimensions.

2-dimensional curves can be defined in a non-parametric (implicit) or in a parametric form. The ***standard parametric form*** is

$$(x(t),\ y(t)),\quad t \in T$$

where T is the parameter interval. Frequently is $T = [0,1]$. An important special case occurs when there is an explicit functional connection between $x(t)$ and $y(t)$ such as

$$y = f(x),\ x \in T.$$

which is equivalent to the form

$$(t,\ f(t)),\quad t \in T.$$

An implicit functional definition of the curve can be done via an equation

$$g(x,y) = 0,\quad (x,y) \in D$$

where D is a domain in the plane. The well-known implicit function theorem says when the representation $g(x,y) = 0$ can be transformed to an explicit connection like $y = f(x)$. Unfortunately, this theorem can only be applied locally in most of the cases. For example, the implicit form for the unit circle curve,

$$x^2 + y^2 - 1 = 0$$

is only locally representable in the forms

$$y = \pm\sqrt{1-x^2} \text{ or } x = \pm\sqrt{1-y^2}$$

with $0 \leq x, y \leq 1$. A more convenient and probably most frequently used form is the parametrization

$$(\cos t,\ \sin t),\ t \in [0, 2\pi].$$

When considering curves in the sequel we think of the standard parametrized form.

Curves are often defined by polynomials or rational (quotient of polynomials) functions and can thus be evaluated using only the four fundamental arithmetic operations. That is, the coordinate functions of the curve, $x(t)$ and $y(t)$ are then either a priori polynomials or rational functions or are at least approximated by polynomials or rational functions. Especially for these reasons polynomials have become an attractive standard vehicle for numerically computing with curves.

A given polynomial may be written down in a number of explicit expressions or forms. (We prefer to say *form* instead of *expression* because of its common use in the interval arithmetic literature.) As an example $p(x) = x^2 - 4$ can also be written as $p(x) = (x - 2)(x + 2)$ and $p(x) = x^2 - 8 + 4$. These three forms are all identical as functions. However, when they are evaluated using fixed length floating point arithmetic or when evaluated as interval expressions (see also Sec. 2.8) they will in general be different (where by *evaluation* we mean the execution of operations as given by the form).

This leads us to consider forms which allow, for example, a numerically stable or robust computation of polynomial values. Another reason for studying various kinds of forms is that polynomials are representable as linear combinations of basis polynomials. Depending on the mathematical or computational purpose for using polynomials the one or other set of basis polynomials is advantageous for the purpose in mind. The set of basis polynomials is frequently chosen so, that special theories or properties of the polynomials can be derived from properties or theories of the basis polynomials. One might for example think of Lagrange, Newton and Hermite polynomials for interpolation problems, Chebyschev and Bernstein polynomials in approximation theory, and the many other forms, supplemented by weights, for numerical integration. In the area of computational geometry and solid modeling, we need to consider the power (or standard) form, Bernstein polynomials, and Bézier curves.

We will deal with the power form and the Bernstein polynomials as far as needed in the remainder of this section, and with Bézier curves and their interval variants in the next sections.

The *power form* (also called standard or normal form) for a real polynomial in one variable of degree n is

$$p(x) = \sum_{i=0}^{n} a_i x^i \tag{7.1}$$

where the basis polynomials are the powers $1, x, \ldots, x^n$ and where $a_i \in R, i = 0, 1, \ldots, n$ with $a_n \neq 0$, are the coefficients. This representation is usually preferred because of its simplicity.

Another representation of p is the *Bernstein form*. Here p is expressed as linear combination of the so-called Bernstein basis polynomials of a given degree $k \geq n$. Then this form of p is defined by

$$p(x) = \sum_{i=0}^{k} b_i^{(k)} B_i^{(k)}(x) \tag{7.2}$$

where the k functions

$$B_i^{(k)}(x) = \binom{k}{i} x^i (1-x)^{k-i}, \; i = 0, 1, \ldots, k \tag{7.3}$$

are the *Bernstein basis polynomials* of *degree* k and where b_i, $i = 0, 1, \ldots, k$ are the coefficients. Since the *real degree* of p (that is the degree of the polynomial in the power form) is almost never needed when Bernstein forms are considered and since misunderstandings occur rather seldom the Bernstein form (7.2) is said to be of *degree* k as the degree of the basis polynomials is of degree k even if the real degree of p is n.

A polynomial written down in the form (7.2) is also called a *Bernstein polynomial of degree* k. Many important results of approximation theory originate from dealing with Bernstein polynomials. An extensive discussion of how to compute with them is found in [48]. The use of Bernstein polynomials in analysis was discussed in the monograph by Lorenz [151].

It became clear early that it was sometimes quite difficult to evaluate polynomials in the power form in fixed length floating point arithmetic within a certain accuracy [271]. A theory of conditioning of polynomials therefore developed and it turned out that polynomials that are ill-conditioned in the power form could be better conditioned in other bases. This is particularly true for Bernstein polynomials [47].

It was also realized early that the Bernstein polynomials had some interesting properties useful in designing curves for CAGD (computer-aided geometric design). For example, they form a partition of the unity, cf. (7.8), they have only one local maximum, and they form a possible basis for Bézier curves [46] as will be seen in the next section. This property in particular explains the close relationship between these polynomials and the curves.

Let $p(x) = \sum_{i=0}^{n} a_i x^i$ again be a polynomial of degree n in the power form. A short calculation shows that $p(x)$ can be converted to the Bernstein form of degree $k \geq n$ by setting

$$p(x) = \sum_{i=0}^{k} b_i^{(k)} B_i^{(k)}(x) \tag{7.4}$$

with

$$b_i^{(k)} = \sum_{s=0}^{i} \frac{\binom{i}{s}}{\binom{k}{s}} a_i, \quad i = 0, 1, \ldots, k. \tag{7.5}$$

Conversely, if a polynomial of degree n is represented in the Bernstein form (7.2) as $p(x) = \sum_{i=0}^{k} b_i^{(k)} B_i^{(k)}(x)$ of degree $k \geq n$ then it can be converted to the power form by computing the power form coefficients by means of the formulas

$$a_i = \sum_{j=0}^{i} (-1)^{i-j} \binom{k}{j} \binom{j}{i} b_j^{(k)}. \tag{7.6}$$

Note that $a_n \neq 0$ and $a_i = 0$, $i = n+1, \ldots, k$.

It is obvious that the real degree of a polynomial can be recognized immediately if it is written down in the power form. The situation is different if a Bernstein polynomial such as (7.2) is given. In this case one can only say that the real degree n is not larger than the degree of the basis polynomials, that is k. In order to determine the degree n the computation of the power coefficients (7.2) seems to be unavoidable. If n is known it is possible to represent the polynomial as a Bernstein polynomial of degree n, cf. [47]. This process is called *degree reduction*, since the degree of the basis polynomials is reduced.

Degree reduction and also its inverse process, *degree elevation* can be executed step by step. The formulas required for this procedure are obtained if a polynomial p of real degree n is expressed with Bernstein basis polynomials of degree $k \geq n$ and of degree $k+1$,

$$p(x) = \sum_{i=0}^{k} b_i^{(k)} B_i^{(k)}(x) = \sum_{i=0}^{k+1} b_i^{(k+1)} B_i^{(k+1)}(x)$$

where $B_i^{(k)}(x)$ and $B_i^{(k+1)}(x)$ are the basis polynomials of order k and $k+1$ respectively and where the $b_i^{(k)}$ and the $b_i^{(k+1)}$ are the corresponding coefficients. In order to determine the relationship between the coefficients the basis polynomials of degree $k+1$ are expanded for $0 < i < k+1$ as follows:

$$\begin{aligned} B_i^{(k+1)}(x) &= \binom{k+1}{i} x^i (1-x)^{k+1-i} \\ &= \binom{k+1}{i} x^i (1-x)^{k-i} - \binom{k+1}{i} x^{i+1} (1-x)^{k+1-(i+1)} \\ &= \frac{k+1}{k+1-i} \binom{k}{i} x^i (1-x)^{k-i} \end{aligned}$$

$$- \quad \frac{i+1}{k+1-i}\binom{k+1}{i+1}x^{i+1}(1-x)^{k+1-(i+1)}$$
$$= \quad \frac{k+1}{k+1-i}B_i^{(k)}(x) - \frac{i+1}{k+1}B_i^{(k)}(x) - \frac{i+1}{k+1}B_{i+1}^{(k+1)}(x).$$

After some smaller manipulations we get

$$b_i^{(k+1)} = \frac{i}{k+1}b_{i-1}^{(k)} + (1 - \frac{i}{k+1})b_i^{(k)}, \ i = 1, \dots, k \tag{7.7}$$

and $b_0^{(k+1)} = b_0^{(k)}$ and $b_{k+1}^{(k+1)} = b_k^{(k)}$.

This is one stage of degree elevation. Similar formulas can be derived for degree reduction.

Degree elevation and reduction are extensively applied in computer graphics and CAGD. Degree elevation is, for example, used to enable a finer control and modeling of the shape of Bézier curves, which are introduced in Sec. 7.4 and have Bernstein polynomials as coordinate functions. For example, typographical fonts can be described and modeled with Bézier curves, cf. [136], p. 116. If the Bézier curve cannot describe the design of a font satisfactory, two steps are executed: First, the number k of control points of the curve is increased which means degree elevation for the coordinate functions, that is, the Bernstein polynomials. Second, the new control points are adjusted by moving them around until the shape of the font is more satisfactory than before. Although the degree elevation does not increase the real degree n of the polynomials the adjustment of the control points will, cf. formula (7.6). Executing these two steps is rather involved but is generally accepted because of the extreme robustness of this procedure.

An extensive collection of formulas and properties for Bernstein polynomials can be found in [75]. The following two formulas are emphasized because of their importance and because they can be used to estimate the range of polynomials, see [212]: The first is

$$\sum_{i=0}^{k} B_i^{(k)}(x) = 1 \tag{7.8}$$

i.e. the basis polynomials of a certain degree, k form a partition of the unity (this follows from the expansion of $(x + (1-x))^k$), and the second is

$$B_i^{(k)}(x) \geq 0, \ \ 0 \leq x \leq 1. \tag{7.9}$$

where the basis polynomials are strictly positive on the open interval (0, 1).

Some computations and formulas involving Bernstein polynomials are simplified if the binomial coefficients $\binom{k}{i}$, cf. (7.3), and the coefficients of the Bernstein polynomial, cf. (7.2), are combined. Then $p(x)$ can be written down in a slightly different form called the *scaled Bernstein form*,

$$p(x) = \sum_{i=0}^{k} d_i^{(k)} x^i (1-x)^{k-i}. \tag{7.10}$$

Accordingly, a *scaled Bernstein polynomial* is a polynomial expressed in the scaled Bernstein form.

As an example of this concept we show how one step of degree elevation works. Let again n be the real degree of the polynomial p. We denote its scaled form with the basis polynomials $x^i(1-x)^{k-i}$ for $i = 0, ..., k$, which are of degree $k \geq n$ as described by (7.10), by p_k. Analogously, denote the scaled form with the basis polynomials of degree $k+1$ by p_{k+1}, etc. If we consider that p_k and p_{k+1} are identical as functions we obtain

$$\begin{aligned}
&p_{k+1}(x) = \\
&p_k(x)x + p_k(x)(1-x) = \\
&d_0^{(k)} x^k + \sum_{i=0}^{k} (d_i^{(k)} + d_{i+1}^{(k)}) x^{i+1} (1-x)^{k-i} + d_i^{(k)} (1-x)^{k+1}
\end{aligned} \tag{7.11}$$

(see also [48]). One recognizes that formula (7.11) is so constructed that the coefficients $d_i^{(k+1)}$, which are required for the representation of p_{k+1}, can be obtained immediately.

For a degree elevation that amounts l single steps the following calculation, which uses (7.8) leads from p_k to p_{k+l}.

$$\begin{aligned}
p_k(x) &= 1 \cdot p_k(x) = \left(\sum_{i=0}^{k} B_i^{(k)}(x) \right) \sum_{i=0}^{k} d_i^{(k)} x^i (1-x)^{k-i} \\
&= \sum_{r=0}^{k+l} \left(\sum_{s=0}^{r} d_s^{(k)} \binom{l}{s+r} \right) x^r (1-x)^{k+l-r}.
\end{aligned} \tag{7.12}$$

The expression (7.12) is of type $p_{k+l}(x)$ so that the determination of the coefficients $d_i^{(k+l)}$ is obvious.

Since Bernstein polynomials are stable with respect to evaluation and other types of manipulations (see [48]) it is recommended that any algebraic manipulation of polynomials in the Bernstein form should be done either in the original Bernstein form or in the scaled Bernstein form rather than converting them to power form, performing the manipulation and then reconverting to one of the two Bernstein forms as is often done.

7.3 Interval Polynomials and Interval Bernstein Polynomials

In the last section we considered polynomial forms and Bernstein polynomials. In this section we discuss these concepts in an interval arithmetic setting. This viewpoint is caused by the requirement of having intervals instead of real numbers as coefficients for the various kinds of polynomials. The introduction of intervals is mostly due to two requirements: The first is that rounding errors occur as side product of numerical computations and numerical manipulations. As the impact of scientific computation becomes larger and larger one can notice a trend towards developing techniques for a reasonable error analysis. Such a technique is interval arithmetic, and the interval coefficients can thus be seen as localization of the exact but unknown values combined with bounds for the errors. Second, a polynomial with interval coefficients can be interpreted as a collection of non-interval polynomials in the same way as an interval as a collection of real numbers. This interpretation offers new ways of modeling curves as the interval polynomial is a notation which on the one hand stands for a set of non-interval polynomials, cf. (7.14), which are provided for the use of modeling curves for a special purpose, and on the other hand, the manipulation of the curves is much easier if done in the context of an interval polynomial than as a set of functions.

Interval polynomials, or more precisely ***interval valued polynomials*** are functions $P : R \to I$ of the form

$$P(x) = \sum_{i=0}^{n} A_i x^i \tag{7.13}$$

where $A_i \in I$, $i = 0, 1, 2, \ldots, n$. The form in which P is written down in (7.13) is called the *power form* of P. If $A_n \neq 0$ then P is said to be of *degree* n.

The most important relationship between an interval polynomial P and real polynomials is given by

$$P(x) = \sum_{i=0}^{n} A_i x^i = \left\{ \sum_{i=0}^{n} a_i x^i : \ a_i \in A_i, \ \ i = 0, \ldots, n \right\} \tag{7.14}$$

for any $x \in R$.

Arithmetic operations for interval polynomials are defined point-wise in the same manner as the arithmetic operations for functions. That is, if Q is another interval polynomial, the arithmetic operations are

$$(P * Q)(x) = P(x) * Q(x)$$

for any $x \in R$, where the symbol $*$ represents each of the four arithmetic operations. Clearly, division is only defined if no division by an interval $Q(x)$ that

contains zero occurs. Unfortunately, the familiar formulas for the arithmetic operations for non-interval polynomials can extended to interval polynomials only in case of addition, subtraction and in special cases of the other operations. That is, if

$$Q(x) = \sum_{i=0}^{m} B_i x^i,$$

and if we assume for simplicity that $n \leq m$ and that non-defined coefficients are set equal to zero, we get

$$\begin{aligned} P(x) \pm Q(x) &= \sum_{i=0}^{m} (A_i \pm B_i) x^i, \\ P(x)Q(x) &\subseteq \sum_{i+j=0} A_i B_j + \left(\sum_{i+j=1} A_i B_j \right) x + \cdots \\ &\quad + \left(\sum_{i+j=m+n} A_i B_j \right) x^{m+n}. \end{aligned} \tag{7.15}$$

We are confronted with an inclusion instead of an equality in the product formula due to the subdistributive law in interval arithmetic, cf. [206]. Equality can only be expected in exceptional cases, which nonetheless occur in our context, for example, if the variable x ranges over a nonnegative interval only, and if additionally, the coefficients of P are all nonnegative or all nonpostive, and the coefficients of Q are all nonnegative or all nonpostive.

Let again be the interval polynomial P be defined by (7.13) and set $A_i = [u_i,\ v_i]$ for $i = 0, ..., n$, and let p and q be the lower and upper boundary functions of P, that is,

$$P(x) = [p(x),\ q(x)].$$

It is to emphasize that in general, p and q are no longer polynomials but only piecewise polynomials. For example, if

$$P(x) = [0,\ 1]$$

then

$$\begin{aligned} p(x) &= 0, \text{ if } x \geq 0, \\ &= x, \text{ if } x \leq 0, \\ q(x) &= x, \text{ if } x \geq 0, \\ &= 0, \text{ if } x \leq 0. \end{aligned}$$

As one can see, neither p nor q is a polynomial, but their restrictions to the set of nonnegative numbers and to set set of nonpositive numbers are polynomials. This holds, by the way, for all interval polynomials.

Since we are only interested in the representation of an interval polynomial in a Bernstein form and since Bernstein forms operate on the interval [0, 1], we can forget about these nonsmooth cases and have

$$P(x) = [p(x),\ q(x)] = [\sum_{i=0}^{n} u_i x^i,\ \sum_{i=0}^{n} v_i x^i] \text{ if } x \geq 0.$$

This formula is in particular true if $x \in [0,\ 1]$.

The next step is to transform the polynomials p and q to their Bernstein form with basis functions of degree k. Formulas (7.1), (7.4) and (7.5) will do it and one obtains

$$p(x) = \sum_{i=0}^{k} c_i^{(k)} B_i^{(k)}(x) \tag{7.16}$$

with

$$c_i^{(k)} = \sum_{s=0}^{i} \frac{\binom{i}{s}}{\binom{k}{s}} u_i, \quad i = 0, 1, \ldots, k. \tag{7.17}$$

and

$$q(x) = \sum_{i=0}^{k} d_i^{(k)} B_i^{(k)}(x) \tag{7.18}$$

with

$$d_i^{(k)} = \sum_{s=0}^{i} \frac{\binom{i}{s}}{\binom{k}{s}} v_i, \quad i = 0, 1, \ldots, k. \tag{7.19}$$

It remains to show that

$$P(x) = \sum_{i=0}^{k} C_i^{(k)} B_i^{(k)}(x) \text{ for } 0 \leq x \leq 1 \tag{7.20}$$

where $C_i^{(k)} = [b_i^{(k)},\ c_i^{(k)}] \in I,\ i = 0, 1, \ldots, n.$

As first part of the proof we show that the left hand side of formula (7.20) is contained in the right hand side: Let

$$r(x) = \sum_{i=0}^{n} \alpha_i x^i \text{ with } \alpha_i \in R$$

be a polynomial which is included in $P(x)$. Because of (7.14) it follows that $\alpha_i \in A_i = [u_i,\ v_i]$ for $i = 0, 1, \ldots, n$. By (7.4) and (7.5) the Bernstein form of r with basis polynomials of degree k is

$$r(x) = \sum_{i=0}^{k} \gamma_i^{(k)} B_i^{(k)}(x)$$

where

$$\gamma_i^{(k)} = \sum_{s=0}^{i} \frac{\binom{i}{s}}{\binom{k}{s}} \alpha_i, \quad \text{for } i = 0, \ldots, k.$$

Since $u_i \le \gamma_i \le v_i$ we have $b_i^{(k)} \le \gamma_i^{(k)} \le c_i^{(k)}$ and finally

$$r(x) \in \sum_{i=0}^{k} C_i^{(k)} B_i^{(k)}(x).$$

This proves the inclusion from the left hand side to the right hand side.

In order to show the converse inclusion of formula (7.20), one proceeds analogously and uses formulas (7.4), (7.6) and (7.1). We can drop the details.

The representation of the interval polynomial P in the form (7.20) is called its *Bernstein form of degree k*. Interval polynomials written down in the form of (7.20) are called *interval Bernstein polynomials of degree k*.

Conversely, not each interval Bernstein polynomial can be written in power form. For example, the interval Bernstein polynomial of degree 1 with the coefficients $C_0^{(1)} = [0,\ 1]$ and $C_1^{(1)} = 0$ is

$$P(x) = [0,\ 1],\ B_0^{(1)}(x) = [0,\ 1],\ x^0(1-x)^1 = [0,\ 1-x].$$

I. e., $P(0) = [0,\ 1]$ and $P(1) = 0$. As one can check for oneself there is no interval polynomial in power form that satisfies these two values. If one would extend the definition of interval polynomials to expressions like $P(x) = \sum_{i=0}^{n} A_i(x^i - x_0^i)$ the given interval Bernstein polynomial could still be written in power form. But there are then counter examples of higher degree, which can be found easily, even if one neglects the areas outside the interval $[0,\ 1]$.

We emphasize the fact that interval Bernstein polynomials cannot always be brought to power form since some authors define an interval Bernstein polynomial already as an interval polynomial, cf. [243].

The arithmetic operations of interval Bernstein polynomials are subsumed to the arithmetic operations of functions in general, that is, the operations are executed point-wise. Interval Bernstein polynomials can be added and subtracted just by adding and subtracting their coefficients, since the distributive law holds in this case as the polynomial values $B_i^{(k)}(x)$ are nonnegative reals. In order to compute the product of two interval Bernstein polynomials, P and Q one has first to compute the values $P(x)$ and $Q(x)$ and then multiply them. If one would multiply the Bernstein forms of the two polynomials as it, for example, was done on the right hand side of the formula (7.15), one only would get a superinterval of $P(x)Q(x)$, as it was shown in formula (7.15) for the case of the power form.

Further aspects of multiplication and division of interval Bernstein polynomials can be found in [243].

7.4 Real and Interval Bézier Curves

Bézier curves are curves that are numerically extremely stable with respect to variations of their shape by means of special parameters called the control points, cf. [47]. Bézier curves are therefore important in computer graphics and CAGD, cf. [46]. They were discovered by Bézier around 1963, while he was working for Renault and they have been given the name *Bézier curves*. He used these curves in in an early design program called UNISURF (see [18] for a historical account). Similar, but unpublished work, had been done by de Casteljau at Citröen. The formulation commonly used in the literature was developed by Forrest about ten years later, cf. [25].

Bézier curves are 2-dimensional parametric curves, their coordinate functions are Bernstein polynomials and they are constructed in the following manner:

Assume $n+1$ points $z_0, \ldots, z_n$ are given in the plane. Furthermore let

$$p(t) = (p_x(t),\ p_y(t)) = \sum_{i=0}^{n} z_i \binom{n}{i} t^i (1-t)^{n-i}, \quad 0 \leq t \leq 1. \tag{7.21}$$

Then $p(t)$ is called a *Bézier curve of degree* n with *control points* $z_0, \ldots, z_n$. Bézier curves can be expressed in terms of Bernstein polynomials,

$$p(t) = \sum_{i=0}^{n} z_i B_i^{(n)}(t)$$

where the functions $B_i^{(n)}$ are the Bernstein basis polynomials of degree n as introduced in Sec. 7.2. The coordinate functions p_x and p_y are thus Bernstein

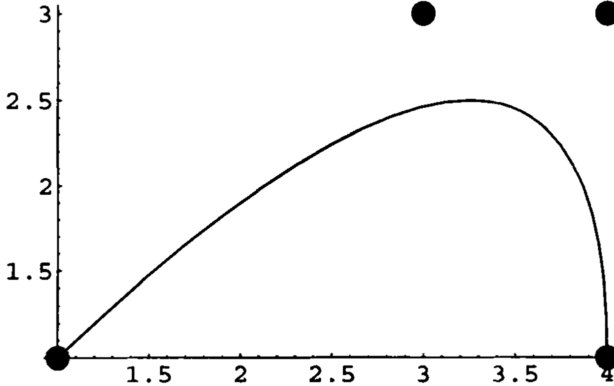


Figure 7.1: Bezier curve of degree 3

polynomials of degree n. By definition of the basis polynomials, cf. (7.3), one has $p(0) = z_0$ and $p(1) = z_n$. Hence z_0 and z_n always lie on the curve.

The control points can be viewed as tools for changing the shape of the curve in a controlled manner. In [71] this is called ***pseudolocal control*** meaning that if a Bézier curve $p(t)$ is specified by a sequence of control points $z_0, ..., z_m$ and if some control point z_i is moved a little bit, the curve is most affected around the points with parameter value close to $t = i/n$. Important is that the change of the curve by moving the control points is a numerically stable process. This is one of the main reasons for working with Bézier curves. If the coordinate functions p_x and p_y of the curve would have a form different from the Bernstein form, for example, if they were Lagrangian interpolation polynomials, and if the modeling of the curve would be done via the Lagrangean knots (also known as Lagrangean interpolation points) this would also be possible but would lead to numerically unstable results. One has, however, to pay a price for the stability of the Bézier curves, that is, the control points will in general, not lie on the curve although they clearly effect the shape of the curve. The disadvantage is not serious, however, as shown by the use of these curves in several well-known curve designing software programs like ADOBE or TGIF.

Since any polynomial can be represented as a Bernstein polynomial and any Bernstein polynomial converted to the power form, cf. Sec. 7.2, any parametric polynomial curve can be represented as a Bézier curve and any Bézier curve can be reformulated as a polynomial curve with the coordinate functions in power form.

In Figure 7.1 an example of a Bézier curve in the plane with 4 control points is given. They are

$$z_0 = (1,\ 1),\ z_1 = (3,\ 3),\ z_2 = (4,\ 3),\ z_3 = (4,\ 1).$$

The control points z_0 and z_3 are on the curve, z_1 and z_2 are not. The Bézier

curve is given by

$$\begin{aligned} p(t) &= 1(1,\ 1)t^3 + 3(3,\ 3)t^2(1-t) \\ &+ 3(4,\ 3)t(1-t)^2 + 1(4,\ 1)(1-t)^3. \end{aligned}$$

The two coordinate functions of p are

$$\begin{aligned} p_x(t) &= t^3 + 9t^2(1-t) + 12t(1-t)^2 + 4(1-t)^3, \\ p_y(t) &= t^3 + 9t^2(1-t) + 9t(1-t)^2 + (1-t)^3. \end{aligned}$$

The control points shape the curve globally even if they are not on the curve. By that we mean that the overall shape of the curve can be modeled by moving the control points around. If we for example move the point z_2 to the other side of the curve it should pull towards that point. This is shown in Figure 7.2.

If finer control of the shape of the curve with n control points is required then the process of ***degree elevation*** can be employed. That is, one uses a larger number of control points for designing the curve, say $n+1$ or more. These points then determine a Bézier curve of degree $n+1$ or higher according to eq. (7.7).

Although it is easy to guide the overall behavior of the curve via the control points it is much more difficult to ensure that the curve passes through one or more given points (interpolation). To achieve this two or more Bézier curves are usually pieced together at the given points and some smoothness conditions are added. The resulting curves are called Bézier splines.

A further important property of Bézier curves is that the curve is contained in the convex hull of its control points as shown in the example in Figure 7.3. This implies that a bounding polygon for the curve can be constructed by connecting control points. Testing if two Bézier curves intersect is therefore best done by first testing for intersection between the bounding polygons.

The length of a Bézier curve is discussed in [79] and a quick way to draw Bézier curves is given in [162].

Interval Bézier curves are obtained if the control points are rectangles, that is, two-dimensional interval vectors, which are commonly called interval control points.

Thus, if $Z_i \in I^2,\ i = 0,\dots,n$ then

$$P(t) = (P_x(t),\ P_y(t)) = \sum_{i=0}^{n} Z_i \binom{n}{i} t^i(1-t)^{n-i},\ \ 0 \le t \le 1 \qquad (7.22)$$

is called an *interval Bézier curve of degree* n with *interval control points* $Z_i,\ i = 0,\dots,n$.

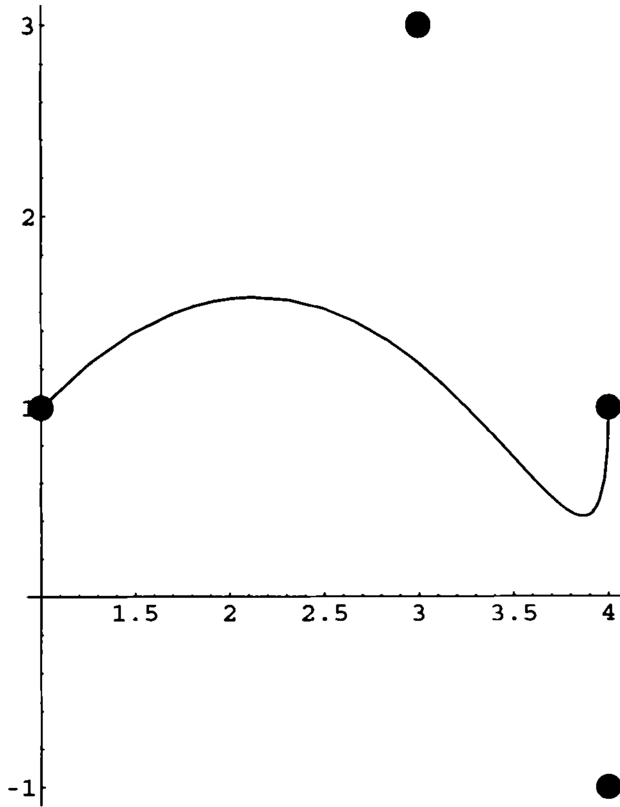


Figure 7.2: Changed Bezier curve of degree 3

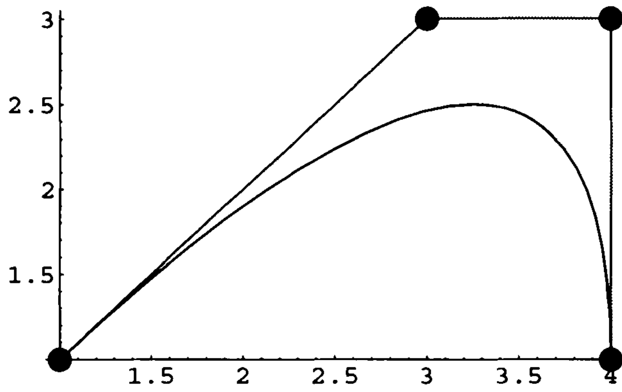


Figure 7.3: Convex hull of control points

7.5 Interval Version of the de Casteljau Algorithm

The de Casteljau algorithm is a recursive scheme to determine the value of a Bézier curve at a given point and to subdivide the curve at this point. There is no need for giving explanations or developing its theory since it has already been satisfactorily done by other authors, for example, [46, 15, 71]. Instead we discuss the basic features of extending the de Casteljau algorithm to interval form in this section, partially following [243]. The main result will be that the conclusions of the algorithm is valid even when the control points are replaced by rectangles, that is, two-dimensional interval vectors.

We first describe the non-interval form of the algorithm. The crucial step of the algorithm is to determine a point between two given points, say z' and z'' with a given ratio, cf. [46]:

Let z' and z'' be points in the plane and let a parameter value $t \in [0,\ 1]$. be given. Then the point

$$z = \zeta(z', z'', t) = tz'' + (1-t)z' \tag{7.23}$$

lies on the straight line connecting z' and z''. Formula (7.23) says that the distance from z' to z related to the distance from z' to z'' has the *ratio* t. Formula (7.23) means further that z is a *convex* or *barycentric combination* of the points z' and z'', cf. Fig. 7.4. One also finds the statement that the assignment $\zeta(z', z'', t)$ is an affine mapping. This is only correct if t is a variable of the mapping, but not z' and z'' and the assignment maps the unit interval $[0,\ 1]$ onto the straight connection line between z' and z'' by the function prescription $t \mapsto \zeta(z', z'', t)$ with constant points z' and z''. Or, if one extends the domain of the variable t to R, the assignment maps R onto that straight line which goes through z' and z'', provided the two points are different.

Let a Bézier curve p be given by $n+1$ control points $z_0, \ldots, z_n \in R^2$. Then the de Casteljau algorithm computes the numerical value of the curve point $p(t)$ and subdivides the curve at $p(t)$ into two parts. For each of the parts $n+1$ control points are determined. The outline of the algorithm is as follows:

ALGORITHM 20 *(De Casteljau Algorithm)*

Step 1. *Set* $z_k^{(0)} = z_k$ *for* $k = 0, \ldots, n$,

Step 2. *Set* $z_k^{(r+1)} = \zeta(z_{k-1}^{(r)},\ z_k^{(r)},\ t)$ *for* $r = 0, \ldots, n-1;\ k = r+1, \ldots, n$.

The final result $z_n^{(n)}$ after the termination of the algorithm is the value $p(t)$. The algorithm also splits the curve into two parts, the first being $p(s)$ for $s \in [0,\ t]$, the second $p(s)$ for $s \in [t,\ 1]$. Further, the points $z_0^{(0)}$, $z_1^{(1)}$,..., $z_n^{(n)}$

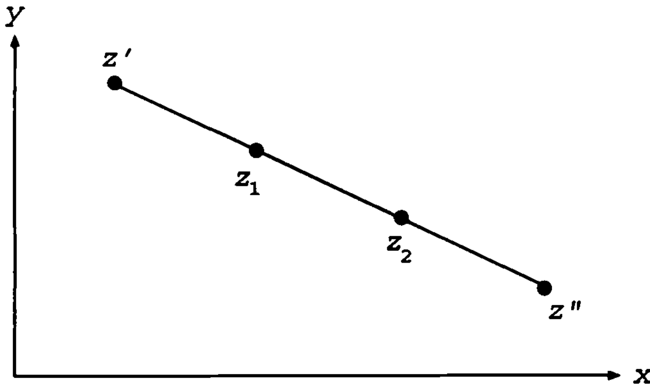


Figure 7.4: Convex combinations

are control points for the first-mentioned curve segment, and the points $z_n^{(n)}$, $z_n^{(n-1)}$,..., $z_n^{(0)}$ for the second curve segment, cf. for example, [46].

The ***interval version*** of the de Casteljau algorithm is formally the same as above. The only difference is that the control points are replaced by rectangles (two-dimensional interval vectors) and are called interval control points. For the interval version let first Z' and Z'' be such rectangles and let the parameter value $t \in [0,\ 1]$ be given. Then the interval vector

$$Z = \zeta(Z', Z'', t) = tZ'' + (1-t)Z' \tag{7.24}$$

is the *convex* or *barycentric combination* of Z' and Z'' with respect to the ratio t. There is no need to distinguish between the interval and non-interval convex combination and thus both are denoted by ζ. In the same manner as an interval polynomial can be seen as a collection of real polynomials as specified in (7.14), the interval point Z can be seen as the collection of all convex combinations $z = \zeta(z', z'', t) = tz'' + (1-t)z'$ with $z' \in Z'$ and $z'' \in Z''$, cf. Fig. 7.5.

One can also read in the literature that the assignment $\zeta(Z', Z'', t)$ is an affine mapping. In this case Z' and Z'' are not variables of the mapping, but constants, and the function prescription is $t \mapsto \zeta(Z', Z'', t)$.

Let an interval Bézier curve P now be given by $n+1$ control interval vectors $Z_0, \ldots, Z_n \in I^2$. Then the interval de Casteljau algorithm determines the interval curve value $P(t)$ and subdivides the curve at $P(t)$ into two parts. For each of the parts $n+1$ interval control points are determined. The outline of the algorithm is as follows:

ALGORITHM 21 *(Interval de Casteljau Algorithm.)*

Step 2. *Set* $Z_k^{(0)} = Z_k$ *for* $k = 0, \ldots, n$,

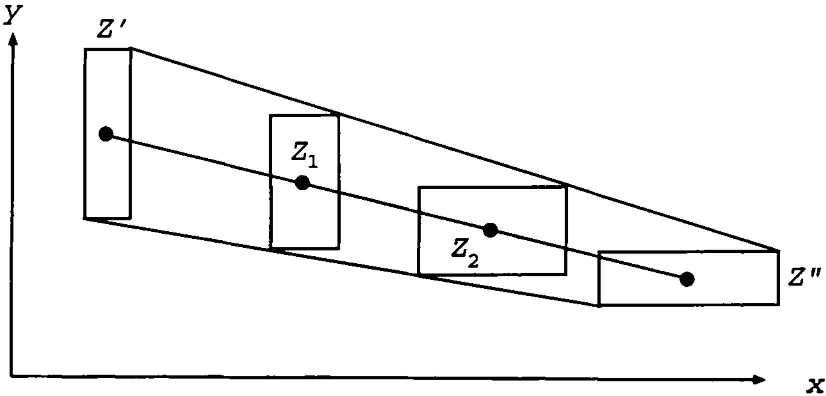


Figure 7.5: Interval convex combinations

Step 2. *Set* $Z_k^{(r+1)} = \zeta(Z_{k-1}^{(r)},\ Z_k^{(r)},\ t)$ *for* $r = 0, \ldots, n-1$;
$k = r+1, \ldots, n$.

The final result $Z_n^{(n)}$ after the termination of the algorithm is the interval value $P(t)$. The interval curve is also split into two parts, the first being $P(s)$ for $s \in [0,\ t]$, the second $P(s)$ for $s \in [t,\ 1]$. Further, the rectangles $Z_0^{(0)}$, $Z_1^{(1)}$,..., $Z_n^{(n)}$ are interval control points for the first-mentioned curve segment, and the rectangles $Z_n^{(n)}$, $Z_n^{(n-1)}$,..., $Z_n^{(0)}$ for the other curve segment, cf. for example, [243]. We now provide a proof for this proposition since we could not find one in the literature. We will not repeat a proof of the validity of the point version of the de Casteljau algorithm, since there are many descriptions of such a proof in the literature, see for example [71]. The output of the algorithm is, however, that:

(i) $p(t) = z_n^{(n)}$,
(ii) the points $z_0^{(0)}$,..., $z_n^{(n)}$ are control points
for the curve segment $p(s)$ for $s \in [0,\ t]$,
(iii) the points $z_n^{(n)}$, $z_n^{(n-1)}$,..., $z_n^{(0)}$ are control
points for the segment piece $p(s)$ for $s \in [t,\ 1]$

The proof for the interval version can be reduced to the point version. One only has to choose two point curves that characterize the interval curve. This can be the midpoint and width curve of P, as well as the two boundary curves of P. We prefer the latter for reasons we which we will explain at the end of this section.

For this purpose it is reasonable to extend the simple interval notation $[a,\ b]$

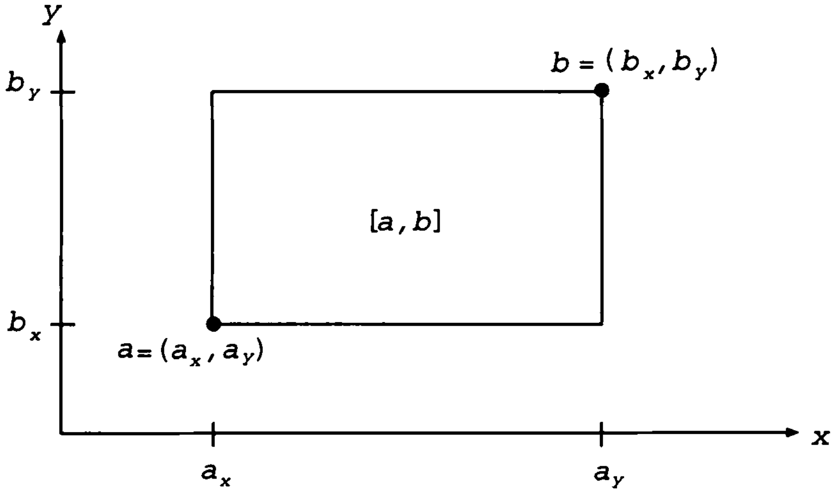


Figure 7.6: Two-dimensional interval

to interval vectors in order to avoid overly complex formulas. Let

$$a = (a_x,\ a_y) \in R^2,\ b = (b_x,\ b_y) \in R^2,\ a_x \leq b_x, \text{ and } a_y \leq b_y.$$

Then we introduce the quite natural notation

$$[a,\ b] = [a_x,\ b_x] \times [a_y,\ b_y] = ([a_x,\ b_x],\ [a_y,\ b_y]).$$

It says that $[a,\ b]$ is a rectangle where the projections to the two coordinate axes are $[a_x,\ b_x]$ and $[a_y,\ b_y]$, or equivalently, $[a,\ b]$ is a two-dimensional interval vector having the components $[a_x,\ b_x]$ and $[a_y,\ b_y]$, cf. Fig. 7.6. We sometimes will say that a and b are the ***boundary points*** or the ***generating points*** of $[a,\ b]$.

This notation of a two-dimensional interval is the set theoretic analogue to the one-dimensional notation because of

$$[a,\ b] = \{c = (c_x,\ c_y) :\ a \leq c \leq b\}$$

where the inequalities between the vectors are understood component-wise. The interval curve is then described by

$$P(t) = [p(t),\ q(t)],$$

where $p(t) = (p_x(t),\ p_y(t))$, $q(t) = (q_x(t),\ q_y(t))$ and $t \in [0,\ 1]$, cf. Fig. 7.7.

Since the main steps of the algorithm are convex combinations we first show that the convex combination of the two-dimensional interval vectors

$$Z' = [a',\ b'],\ Z'' = [a'',\ b''] \text{ with ratio } t \in [0,\ 1]$$

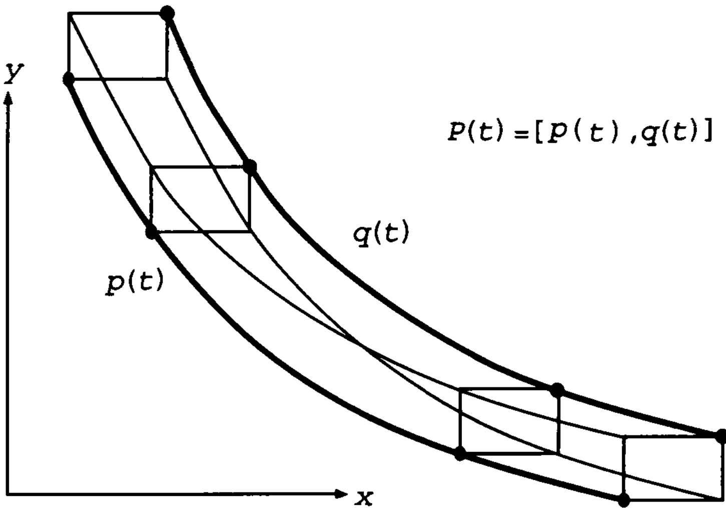


Figure 7.7: An interval curve

can be determined just by computing the convex combinations of the boundary points of Z' and Z'', that is, the convex combinations of a', a'' with ratio t and of b', b'' with ratio t.

We need the following formulas for the calculation which are direct consequences of the interval arithmetic operations and vector operations: If $\alpha \geq 0$ and $a = (a_x,\ a_y)$, $b = (b_x,\ b_y)$, then

$$\begin{aligned} \alpha[a',\ b'] &= \left(\alpha[a'_x,\ b'_x],\ \alpha[a'_y,\ b'_y]\right) \\ &= \left([\alpha a'_x,\ \alpha b'_x],\ [\alpha a'_y,\ \alpha b'_y]\right) = [\alpha a',\ \alpha b']. \end{aligned}$$

We now are ready for considering the convex combination of Z' and Z'' with respect to the ratio t,

$$\begin{aligned} Z &= \zeta(Z',\ Z'',\ t) \\ &= tZ'' + (1-t)Z' \\ &= t[a'',\ b''] + (1-t)[a',\ b'] \\ &= [ta'',\ tb''] + [(1-t)a',\ (1-t)b'] \\ &= [ta'' + (1-t)a',\ tb'' + (1-t)b'] \\ &= [\zeta(a',\ a'',\ t),\ \zeta(b',\ b'',\ t)]. \end{aligned}$$

The first and the last line of this chain of equations give the proposed result, that is, the convex combination of rectangles can be performed via the convex combinations of their boundary points.

We return to the interval Bézier curve $P = [p,\ q]$ and its $n+1$ control interval vectors $Z_0, \ldots, Z_n \in I^2$. Let $Z_k = [a_k,\ b_k]$, $a_k, b_k \in R^2$ for $k = 0, \ldots, n$. We are ready to prove that the application of the interval de Casteljau algorithm to P and the Z_k is equivalent to the application of the de Casteljau algorithm to p with the control points a_k and to q with the control points b_k. It is sufficient to prove that the steps of the interval and the non-interval versions of the algorithm are equivalent.

Parts 1 of the interval and of the non-interval de Casteljau algorithm are just the initialization or what is the same, the renaming of the input data by adding the superscript 0. Thus, part 1 of the interval version, that is

$$Z_k^{(0)} = Z_k \quad \text{for} \quad k = 0, \ldots, n$$

corresponds to part 1 of the point versions, that is,

$$a_k^{(0)} = a_k,\ b_k^{(0)} = b_k \text{ for } k = 0, \ldots, n$$

of the point version. Since $Z_k = [a_k,\ b_k]$, we obtain

$$Z_k^{(0)} = [a_k^{(0)},\ b_k^{(0)}] \text{ for } k = 0, \ldots, n$$

which proves the equivalence of parts 1.

Parts 2 of the algorithms are a sequence of convex combinations. Each maintains the proposed equivalence, as shown before, such that the relation

$$Z_k^{(r)} = [a_k^{(r)},\ b_k^{(r)}] \text{ for } k = 0, \ldots, n$$

is valid for any stage r of the computation, where $Z_k^{(r)}$ comes from the interval algorithm and the points $a_k^{(r)}$ and $b_k^{(r)}$ from the non-interval algorithm. As a by-product we get

$$Z_n^{(n)} = [a_n^{(n)},\ b_n^{(n)}] = [p(t),\ q(t)],$$

which proves the formula $Z_n^{(n)} = P(t)$.

It remains to verify that $Z_0^{(0)}, \ldots, Z_n^{(n)}$ are interval control points for the curve segment $P(s)$ for $s \in [0,\ t]$, and $Z_n^{(n)}, \ldots, Z_n^{(0)}$ for the curve segment $P(s)$ for $s \in [t,\ 1]$.

Putting this together: It follows from the properties of the non-interval algorithm that $a_k^{(k)}$ and $b_k^{(k)}$ for $k = 0, \ldots, n$ are control points for the curves $p(s)$ and $q(s)$ for $s \in [0,\ t]$, resp. and that $a_n^{(r)}$ and $b_n^{(r)}$ for $r = n, \ldots, 0$ are control points for $p(s)$ and $q(s)$ for $s \in [t,\ 1]$, respectively. Then, for $s \in [0,\ t]$ one obtains

$$\begin{aligned} P(s) &= [p(s),\ q(s)] = [\sum_{i=0}^{n} a_i^{(i)} B_i^{(n)}(s),\ \sum_{i=0}^{n} b_i^{(i)} B_i^{(n)}(s)] \\ &= \sum_{i=0}^{n} [a_i^{(i)} B_i^{(n)}(s),\ b_i^{(i)} B_i^{(n)}(s)] \text{ by definition of interval addition} \\ &= \sum_{i=0}^{n} [a_i^{(i)},\ b_i^{(i)}] B_i^{(n)}(s)] \text{ by definition of interval multiplication} \\ &= \sum_{i=0}^{n} Z_i^{(i)} B_i^{(n)}(s). \end{aligned}$$

The first and the last expression in this chain of equations proves that $Z_i^{(i)}$ for $i = 0, \ldots, n$ are interval control points for the curve piece $P(s)$ for $s \in [0,\ t]$. The proof for the second curve piece is analogously.

Remark. We were using the characterization of an interval Bézier curve by its two generating boundary curves. Some readers might prefer a characterization of an interval Bézier curve or of other kinds of interval curves by their midpoint and width curves. This representation is based on the midpoint - width form of an interval $A \in I$,

$$A = \text{mid }(A) + w(A)[-1,\ 1]/2,$$

cf. Ch. 2. We do not see too many advantages of such a representation. First, already involved formulas become even more involved with this representation, second, the intervals have to be reformulated from the boundary to the midpoint - width representation and converse. Third, the execution of products is certainly possible but gives complex formulas. Fourth, the midpoint - width representation is more sensitive with respect to rounding errors since operations with the midpoint necessitates an upwards and downwards rounding and the operations with the width an upwards rounding. This results in three rounding operations. In contrast, the standard representation only needs two roundings, that is the downward rounding of the left endpoint and the upward rounding of the right endpoint.

There exist also some investigations about the error analysis of Bézier curve computations, see for example, [243, 242].

Chapter 8

Robust Computations of Selected Discrete Problems

8.1 Introduction

In this final chapter we select three different areas in order to show the wide range of applications of interval arithmetic and ESSA. These are plane convex-hull computations, Delaunay triangulations, and line simplifications where the first two areas are in computational geometry and the last area in GIS. It is typical for each of these areas that the rounding errors and the falsification of the results by these errors are not easy to control. Many ideas and several theories have been invented to make the results reliable and independent of the numerical computations or at least to provide the results with safe error bounds.

Each of these areas need a certain depth of knowledge which is necessary to get an overview of the theory and to make essential improvements for the numerical performance of the related algorithms. Therefore we will discuss the areas in more details in order to show how interval analysis and ESSA lead to robust results for computations in these areas.

In Sec. 8.1 we present an algorithm for computing the convex hull of a finite set of points. The algorithm is based on a version of the Graham scan algorithm and shows the following features:

- If the points are already (single precision) machine numbers, the computation is rounding-error free, that is, the computed hull is the hull that would have been computed if real arithmetic were available.
- If the points are arbitrary numbers, the algorithm renders the smallest possible machine representable convex hull that includes the exact convex hull.

- The worst case computation time is still $O(n \log_2 n)$.
- Only floating point arithmetic with double mantissa length is required. No mantissa splitting or other mantissa manipulations are needed; one only has to know the exponent parts of the numbers. Also, no fixed point accumulator is needed.
- Single precision interval arithmetic is recommended for accelerating the computation, but is not necessary.

In Sec. 8.3 we present an exact and hence robust algorithm for the computation of Delaunay and power triangulations in two dimensions, which has been developed in [66] The algorithm avoids numerical errors and degeneracies caused by the accumulation of rounding errors in fixed length floating point arithmetic when constructing these triangulations.

Most methods for computing Delaunay and power triangulations involve the calculation of two basic primitives: the INCIRCLE test and the CCW orientation test. Both primitives require the computation of the sign of a determinant. We present an algorithm which computes the sign of the determinant exactly based on ESSA. The exact computation of the primitives allows the construction of the correct Delaunay and power triangulations. The method has been implemented and tested for the incremental construction of Delaunay and power triangulations. Tests have been conducted for different distributions of sites for which non-exact algorithms may encounter difficulties, for example, slightly perturbed points on a grid or on a circle. Experimental results show that the performance of our algorithm is comparable with that of a simple implementation of the incremental algorithm in single precision floating point arithmetic. For random distribution of points the exact algorithm is only 4 times slower than the inexact implementation. The algorithm is easy to implement, robust and portable as long as the input data to the algorithm remains exact.

The sensitivity of computed results in GIS (Geographical Information Systems) is considered in Sec. 8.3 on the example of the Ramer-Douglas-Peucker line simplification algorithm. Using ESSA and interval arithmetic we describe a robust version of this algorithm, where the determination of the simplification is rounding error free if the data are already machine numbers. Under these assumptions the results are reproducible which is not the case with other versions of the algorithm.

8.2 Convex-Hull Computations in 2D

8.2.1 Introduction

The computation of the convex hull of a finite set of points in the plane is a fundamental problem in computational geometry. It has been considered by

a number of researchers, see for example [198] or the survey [244]. Most of the work has been concerned with devising algorithms meeting the worst case complexity of $O(n \log_2 n)$.

Problems regarding the accuracy of a convex hull arise when implementing convex hull algorithms on finite precision floating point computing devices with fixed mantissa length. The computed hull may for example be non-convex, it may be larger than the exact hull or it may exclude some points that should be part of the hull. These problems were considered by [147] within the framework of strongly convex hulls. The thesis of Salesin [237] considered the same problems within the notion of epsilon-geometry. Other authors proposed further solutions ([41, 115, 116, 122]).

This section, which is based on [222] and ESSA presents an algorithm that avoids most of these problems. As we see later, *double precision* execution of ESSA suffices for our purpose as far as ESSA is applied to the left-turn test for single precision points. We keep in mind that ESSA needs no further expansion of the mantissa length nor any other mantissa manipulation. Only the exponent part of the numbers must be known, which is, however, a simple command in C or C^{++}. This means that if the given points are already single precision numbers then the left-turn test executed with ESSA with double precision gives the exact result, even if the distance between two adjacent points is smaller than 1 ulp.

There are many convex hull methods that depend mainly on the left-turn test. Since the left-turn test can be executed exactly with ESSA it follows that *convex hulls constructed with such methods and ESSA are exact convex hulls.* Even then it is only necessary to perform the left-turn test with ESSA in extreme cases. If the constellation of the 3 points is "normal," the usual left-turn test computation with single precision would suffice. The result is, however, not guaranteed, and further, one never knows à priori, when the standard single precision computation is sufficient to guarantee a correct result. Therefore, we first execute the left-turn test with single precision interval arithmetic, which either gives a guaranteed result or provides the message that the result is uncertain. Our algorithm repeats this left-turn test with ESSA if the second situation occurs.

Hence, if the input data for which the convex hull is required, already consists of *machine representable numbers* it follows that the application of appropriate convex hull codes together with the exact left-turn test implementation as described in Subsec. 8.2.3 renders the *exact convex hull.* If, however, the points of the original input data are *not machine representable*, a conversion error is unavoidable when the data is entered into the machine. One could imagine that ESSA now has no effect, since for what does one need an exact left-turn test for uncertain points? Clearly, it is no longer possible to compute the exact convex hull of the original data. Instead one can construct the *smallest possible machine representable convex hull* which includes the original data. This could not be achieved without exact computation devices, cf. Li-

Milenkovic [147]. For this reason we again use interval arithmetic and replace each point of the original data which is not machine representable with the smallest machine representable rectangle which is a two-dimensional interval that includes this point. Then the convex hull algorithm is applied to the totality of all corners of these rectangles. Since the corners are machine representable, the exact convex hull of the rectangles is computed rounding error free such that this is the smallest constructible convex hull of the original point set.

The choice of convex hull method for computing the hull is of less importance than ESSA since ESSA provides the essential foundation for exact computation. We therefore use a rather simple, transparent and known convex hull method in order to demonstrate clearly how the method and ESSA fit together. I.e., we chose the version of Graham scan that is described in O'Rourke [190], p. 85 and drop the improvements that have been published within the last few years, cf. [1, 76, 82]. Certainly, ESSA can also be used with other convex hull methods, no matter how sophisticated they are. In the same way, ESSA can also turn 3D-convex hull algorithms into exact and optimal ones.

The numerical costs of $O(n \log_2 n)$ are not changed if ESSA is used in Graham scan.

In Subsec. 8.2.2 we give a short description of the Graham scan version used. In Subsec. 8.2.3 ESSA and Graham scan are merged, and a few interval devices are included. Subsec. 8.2.4 contains well- and ill-conditioned numerical examples. In Subsec. 8.2.5 it is shown that ESSA can be also be combined with convex hull algorithms that are more sophisticated than Graham scan. As an example and to point out the principles which have to be considered for such a merging, we briefly discuss how an exact and optimal hull algorithm has to be designed which is based on the ideas of Kao-Knott[122].

8.2.2 A Prototype Graham Scan Version

We briefly describe a simple and transparent prototype Graham scan algorithm [77] for the computation of a convex hull as it is found, for example, in O'Rourke [190], p. 85 (version B), omitting refinements and improvements, in order to have it available for discussion in the sequel. The combination of the proper convex hull idea with ESSA and the interval devices can thus best be demonstrated. The prototype also forms the basis for the numerical examples, cf. Subsec. 8.2.4. For a more sophisticated combination see Subsec. 8.2.5.

The algorithm consists of a preprocessing sorting step (Steps 1 and 2) followed by the convex hull construction (Step 3) as the main part.

Let a plane point set $A = \{(x_i, y_i) : i = 1, \ldots, n\}$ be given. The following algorithm constructs the convex hull of this point set, assuming exact computation. More precisely, the algorithm sifts out all points of A which are vertices of the convex hull of A.

ALGORITHM 22 *(Prototype Graham scan)*

Step 1. *Among all the points of A having minimum x-coordinate, determine the one with maximum y-coordinate. Denote this point by p_0. It is already a convex hull point.*

Step 2. *Sort all other points of A by the angle counter-clockwise about p_0 where the basic sign of the angle is the positive x-direction. Use any sorting procedure, for example, Heapsort, cf. [199], but use the left-turn test (cf. Subsec. 8.2.3) for comparing any 2 angles.*

If two points have the same angle, drop the one which is nearer to p_0 (simple coordinate comparison). Denote the sorted list by $p_0, p_1, \ldots, p_{n-1}$. Also p_{n-1} is already a convex hull point.

Step 3. *Create a stack $S = (p_{n-1}, p_0)$. For $i = 1, \ldots, n-1$ do:*

(i) *Denote the last two elements of S by p_{ll}, p_l (l indicates the last, and ll the last before the last).*

(ii) *If p_i is strictly left to the directed line from p_{ll} to p_l then put p_i after p_l on S else remove p_l from S and return to (i).*

The known improvements of Graham scan mostly relate to the preprocessing step where "interior" points that can never become hull points are already removed during the sorting. Other versions of sorting are also known in which cases Step 3 sometimes has to be modified. Such changes do not influence the worst case costs of $O(n \log_2 n)$, but the average costs can be considerably reduced.

8.2.3 The Exact and Optimal Convex Hull Algorithm

The Graham scan version of Subsec. 8.2.2 is taken as the prototype skeleton for this algorithm, and nothing needs to be changed. The only items left are two details showing how to combine ESSA and the interval arithmetic devices with Graham scan in order to meet our claims that the resulting method is exact and optimal.

These two items are the input of the data set and the exact execution of the left-turn test, which is the backbone of the angular sorting as well as the proper convex hull construction, cf. Subsec. 8.2.2.

If the input data already consists of machine points, one can skip part A since Graham scan combined with the exact left-turn test implementation, cf. part B, leads to the exact hull.

A. The input of the point set

If the point set for which we want to determine the convex hull has been generated by some preceding computer program on the same machine, the

points are certainly machine representable, and hence, an *exact* hull of the previously computed points can be expected when our procedure is applied. This does not mean, however, that the convex hull is a convex inclusion of the exact point set that is expected theoretically or geometrically. The reason is that only computed approximations of these points are known. If one wishes to obtain at least a convex polygon that contains the exact points then it has to be planned in advance. It would mean that error bounds would have to be provided from the previous computation phase, for instance, with interval arithmetic. In this latter case, the interval boundaries are machine representable, and our exact convex hull procedure will render the exact convex hull of all the boundary points. This hull will, at the same time, include the originally required hull.

If the input data has not yet been entered into the machine then it is unlikely that the points are machine numbers unless they are integers and the exact convex hull cannot be expected. Nevertheless, a best ***possible including convex hull*** can be determined if one proceeds as follows:

In order to handle the input data properly one should avoid the standard input facilities of the machine since the input mode is, generally, unknown or not very reliable. The best one could do is to use an interval software package (such as C-XSC [132]), to represent the numbers, which are mostly decimal numbers, as a quotient of two integers (such as $0.3 = 3/10$) then enter the integers into the machine (they will be represented exactly) and then let the division be executed by the interval package. Good packages such as C-XSC will return the smallest machine representable interval that includes the value of the decimal quantity.

If one has access to a reliable directed rounded arithmetic one can also obtain such a smallest inclusion. In this case a left rounding accompanied with a right rounding in connection with the division will provide the desired inclusion.

Since we deal with 2D points each of two components will be surrounded by an interval so that the points will finally be included in rectangles. If the Graham scan code with the exact left-turn test implementation of part B is applied to all these rectangle corners then we obtain the ***smallest possible*** machine representable ***convex hull*** of the given data.

B. The left-turn test

The current literature on convex hull algorithms seems to be united in the opinion that the left turn test is the achilles heel of any 2D convex hull program. The reason is that there has been no easy way of avoiding the inaccuracies caused by rounding errors in this test. It is also well known that these inaccuracies are responsible for all kinds of subsequent errors, mainly of a topological, logical and numerical nature. Many sophisticated strategies have been developed so far in order to avoid this chaotic influence of rounding errors.

No sophisticated considerations are necessary in this implementation, if the

left-turn test is executed with ESSA in double precision, as already explained in Chapter 4. In this case no rounding errors can occur (provided the machine numbers are single precision). We also show a way to avoid a double precision ESSA, if desired, at the end of this section.

Let the different points a, b, c in the plane be given with single precision components a_i, b_i, c_i $(i = 1, 2)$. The *left-turn test* says that c is strictly left to the directed line leading from a to b iff

$$D = \begin{vmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{vmatrix} > 0,$$

cf. Ch. VI. There are various ways to evaluate the determinant D, for example,

$$D = (b_1 - a_1)(c_2 - a_2) - (b_2 - a_2)(c_1 - a_1), \tag{8.1}$$

which is certainly an optimum way to evaluate D in terms of the number of arithmetic operations. We do not need D, however, we only need the exact sign of D. ESSA cannot be applied to the computation of D by (8.1) since (8.1) is not presented as a sum of numbers. Hence we have to rearrange D so that a sum of numbers is formed, that is,

$$D = a_1 b_2 + a_2 c_1 + b_1 c_2 - b_2 c_1 - a_2 b_1 - a_1 c_2 .$$

ESSA computes the sign of this sum exactly if the summands are exact. Since the components a_i, b_i, c_i are assumed to be single precision numbers, we have to present the products $a_i b_j, \ldots$ in double precision to avoid the loss of being exact. This can be coded in C or C^{++} without additional efforts.

Note that the left-turn test has to be applied at two different phases in the Graham scan version of Subsec. 8.2.2. First of all at the angular sorting and secondly, when determining the convex hull points in Step 3 of the prototype version in Subsec. 8.2.2.

The *most economical* way to execute the left-turn test is probably the following procedure which is also used in the sequel: The determinant D is computed explicitly using single precision interval arithmetic with formula (8.1). Because of the obligatory outward rounding, which is incorporated in every interval package, the computational result will be an interval $\overline{D} = [D_1, D_2]$ with $D \in \overline{D}$, cf. [165]. (See also [236] for a simple implementation of interval arithmetic and [21] for a discussion of the use of interval filters in geometric computations). Hence, the possible results

$$\begin{aligned} \overline{D} &> 0 \quad (\text{i.e., } D_1 > 0)\,, \\ \overline{D} &\le 0 \quad (\text{i.e., } D_2 \le 0) \end{aligned}$$

give the guarantee that $D > 0$ or $D \le 0$ is, respectively. Only the third possible result, that is

$$D_1 \le 0 < D_2$$

does not permit a decision about the sign of D. In this case it is necessary to switch to ESSA to obtain the guaranteed information about the sign of D.

It is maybe superfluous to say that each occurring sign determination can be executed with ESSA if one wants to avoid interval arithmetic. Similarly, if one does not want to write a code for ESSA in double precision, it can be avoided by the following extra step:

Since the point components a_i, b_i, c_i are single precision, the products a_ib_j, etc., are double precision and are the summands of the sum in question in ESSA. These summands can be split into two parts which are then single precision quantities. The splitting can be executed in C or C^{++} without much ado. Now, a single precision ESSA is applicable to the sum of split summands. The single precision version will have 12 summands, a doubling of the number of summands from the double precision version.

8.2.4 Numerical Examples

A large number of numerical tests were made in order to make sure that there were no flaws in the method described above. The tests were made with randomly generated points, with stable as well as unstable configurations, machine representable (Fig. 8.1–8.7) as well as non-machine representable (Fig. 8.8–8.10) points. The number of points were varied from 10 to 1000 (which gives 4000 in case the 1000 points were not machine representable) and the example described in O'Rourke [190], p. 94, was repeated to show that the algorithm handles collinearities and other difficult situations safely. The calculations were done on a SUN20 workstation in C and C^{++}.

The following abbreviations are used in the statistics:

- n — number of points *after* the input of data (i.e., n is 4 times the number of points one had before the input of the data in Fig. 8.8-8.10)
- nHull — number of convex hull points
- nGr — number of points after the preprocessing (that is, after having removed collinear points discovered by the angular sorting and after having removed equal points arisen by the replacement of non-machine representable points with up to 4 machine representable points). Hence, nGr is the number of those points, Step 3 of Graham scan, cf. Subsec. 8.2.2, is working with.
- nESSA — counts the number of ESSA applications during the angular sorting and during the main computation (Step 3)
- tESSA — time spent with ESSA
- t — overall time computed (in microseconds)

In the graphics, the points of the input data are marked by x-like crosses. If the points are too dense w.r.t. the plotting solvability, the related crosses are united. Those points that are vertices of the convex hull are additionally marked with squares, which print over the cross.

Fig. 8.1 shows the example of O'Rourke [190], p. 94, that contains several collinearities, also on the convex hull polygon. Note that they have been discovered precisely and that they are not counted as convex hull points as long they are not vertices.

Fig. 8.2, 8.3 and 8.4 show well-conditioned harmless examples with 20, 100 and 1000 randomly generated machine representable points in the area of [10, 11] × [10, 11].

Fig. 8.5, 8.6 and 8.7 show ill-conditioned examples with 20, 100 and 1000 randomly generated machine representable points in the area of [10, 11] × [10 000, 10 001]. The ill-conditioning arises since the y-coordinates of the points are all identical at five leading digits.

Fig. 8.8, 8.9 and 8.10 show ill-conditioned examples with 20, 100 and 1000 randomly generated non-machine representable points in the area [10, 11] × [10, 11], so that after the input all corners of the including rectangles, that is, 80, 400 and 4000 points, have to be counted. One notes a rapid increase of the number of ESSA invocations. The ill-conditioning comes from the edge length of the rectangles being just 1 ulp.

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
19	8	14	10	0	3	0	2

Table 8.1: Data from computing with O'Rourke's example as input

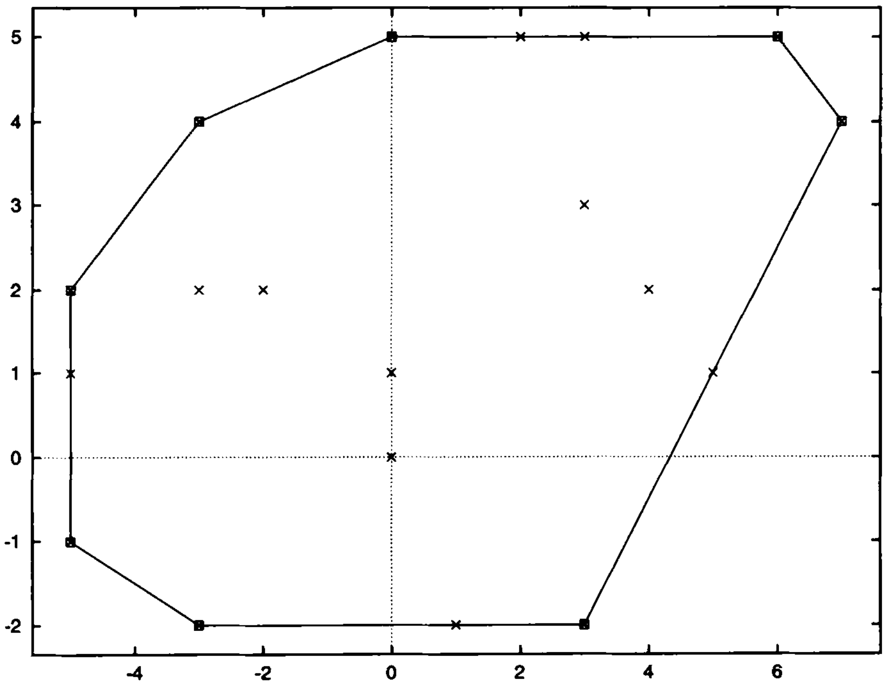


Figure 8.1: Graphics for Table 8.1 (O'Rourke's example)

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
20	9	20	0	0	0	0	2

Table 8.2: Data from well-conditioned example with 20 points

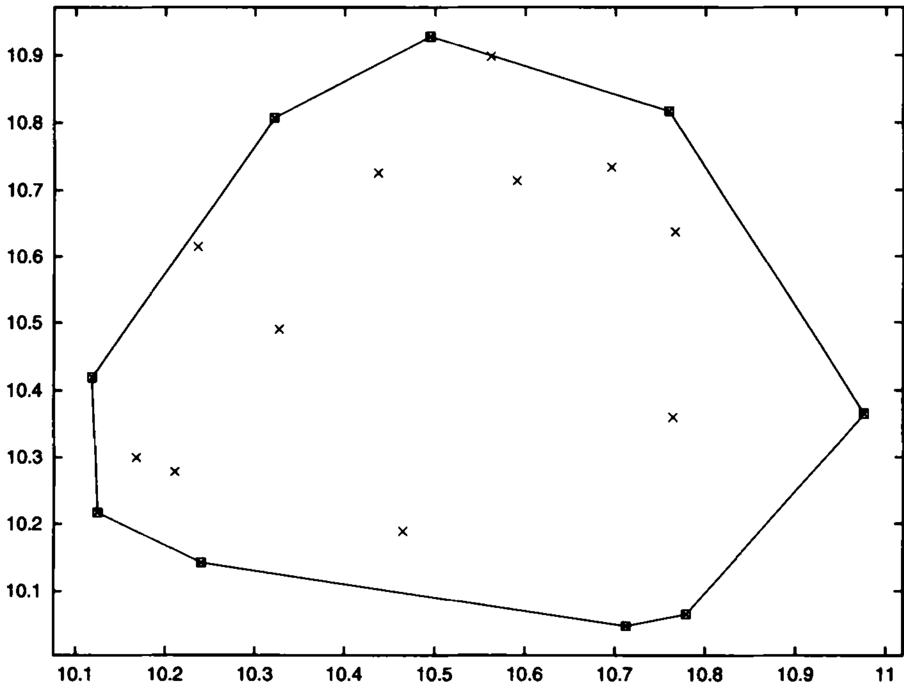


Figure 8.2: Graphics for well-conditioned example with 20 points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
100	13	100	2	0	4	0	14

Table 8.3: Data from well-conditioned example with 100 points

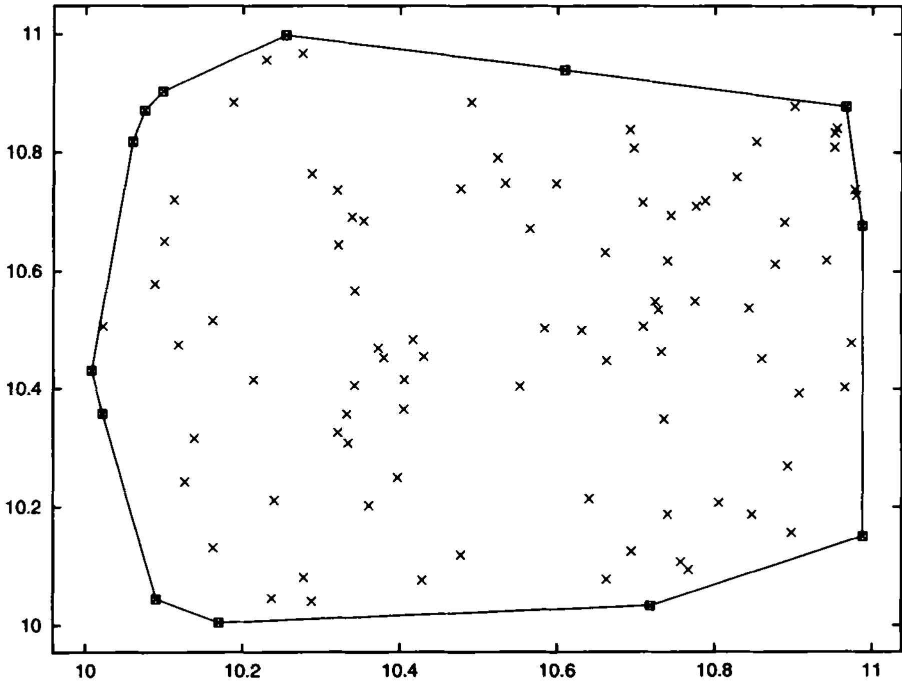


Figure 8.3: Graphics for well-conditioned example with 100 points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
1000	17	1000	176	3	92	2	208

Table 8.4: Data from well-conditioned example with 1000 points

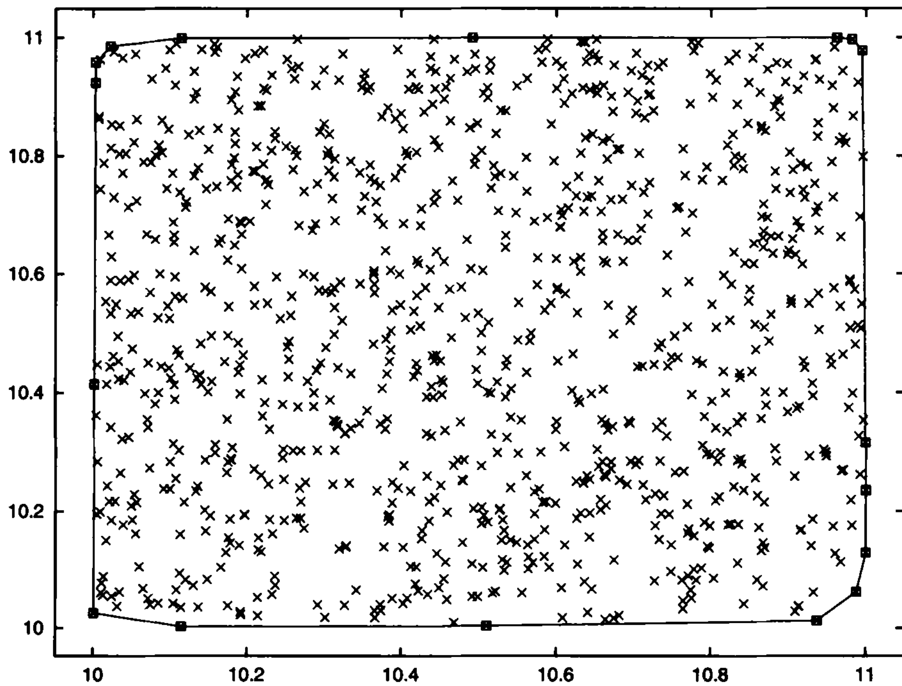


Figure 8.4: Graphics for well-conditioned example with 1000 points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
20	8	20	36	0	16	1	3

Table 8.5: Data from random ill-conditioned example with 20 points

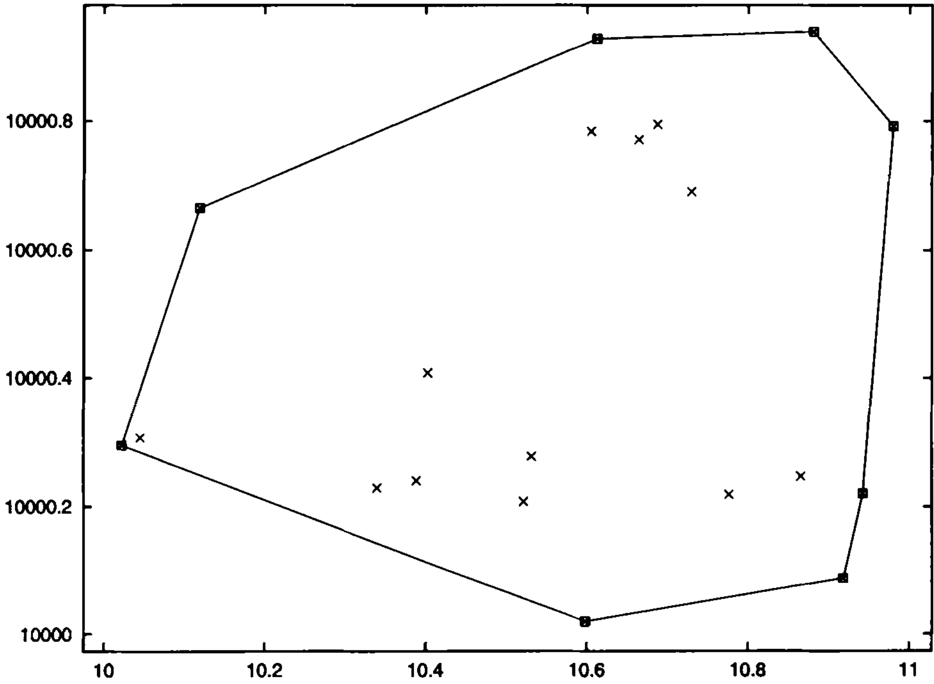


Figure 8.5: Graphics for random ill-conditioned example with 20 points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
100	13	100	471	11	174	3	25

Table 8.6: Data from random ill-conditioned example with 100 points

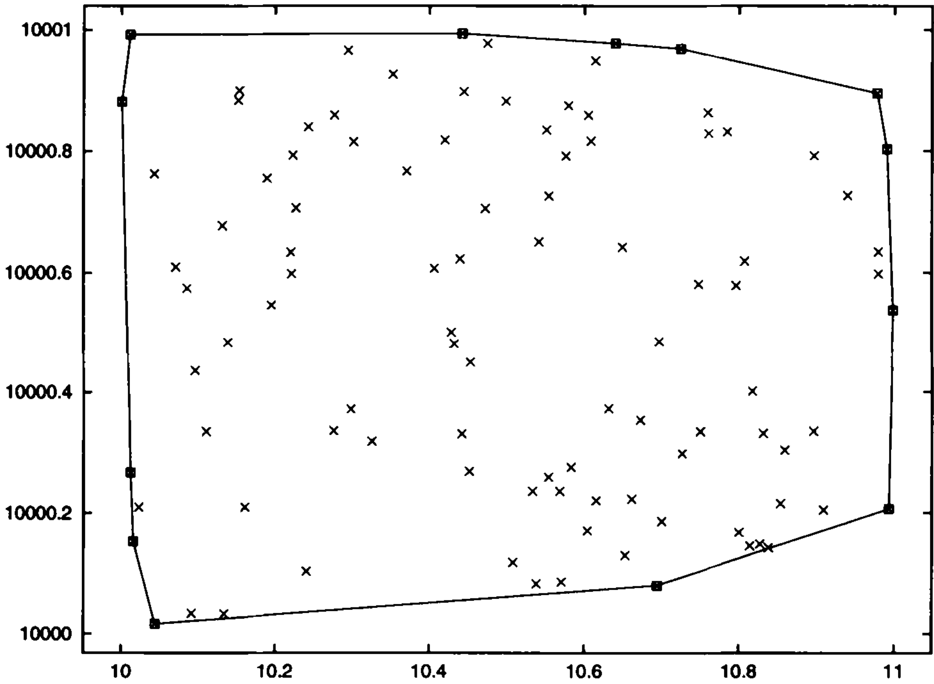


Figure 8.6: Graphics for random ill-conditioned example with 100 points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
1000	19	1000	8700	112	1959	35	427

Table 8.7: Data from random ill-conditioned example with 1000 points

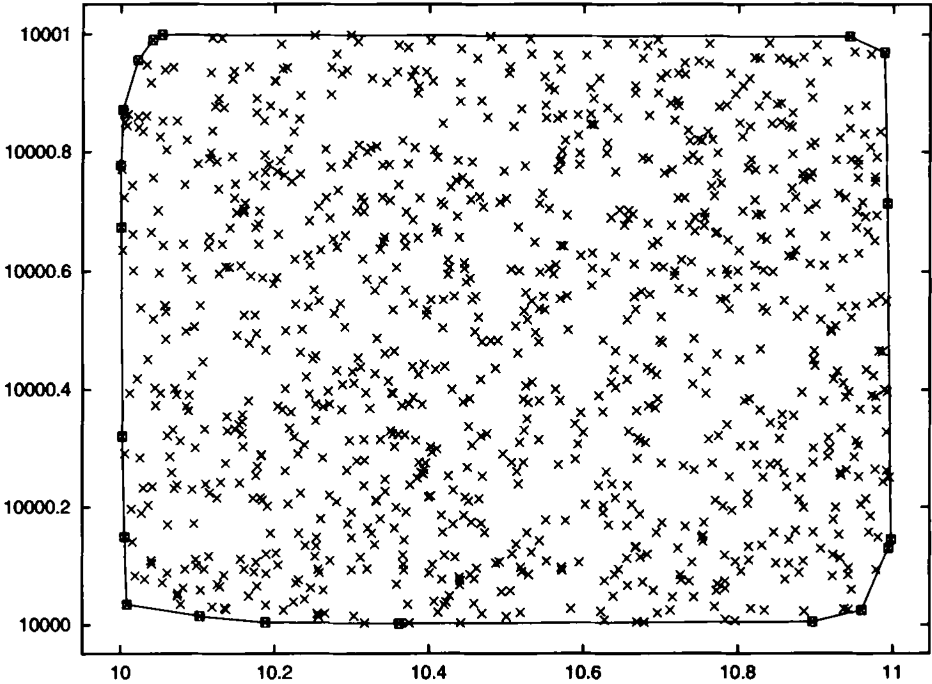


Figure 8.7: Graphics for random ill-conditioned example with 1000 points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
80	12	80	241	3	110	1	14

Table 8.8: Data from random ill-conditioned example with 20 not machine representable points

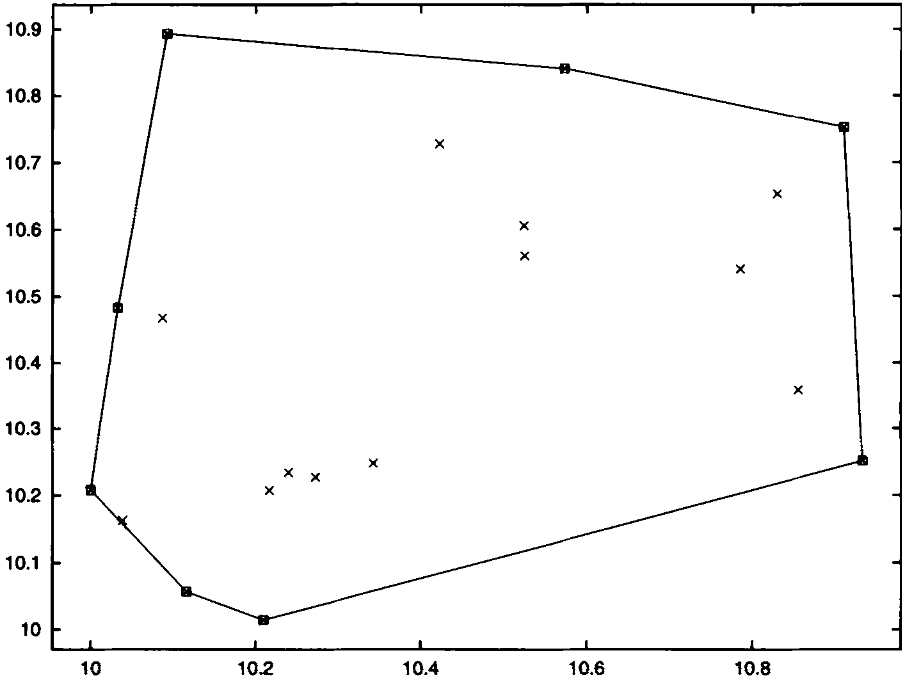


Figure 8.8: Graphics for random ill-conditioned example with 20 not machine representable points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
400	12	400	1012	7	509	3	85

Table 8.9: Data from random ill-conditioned example with 100 not machine representable points

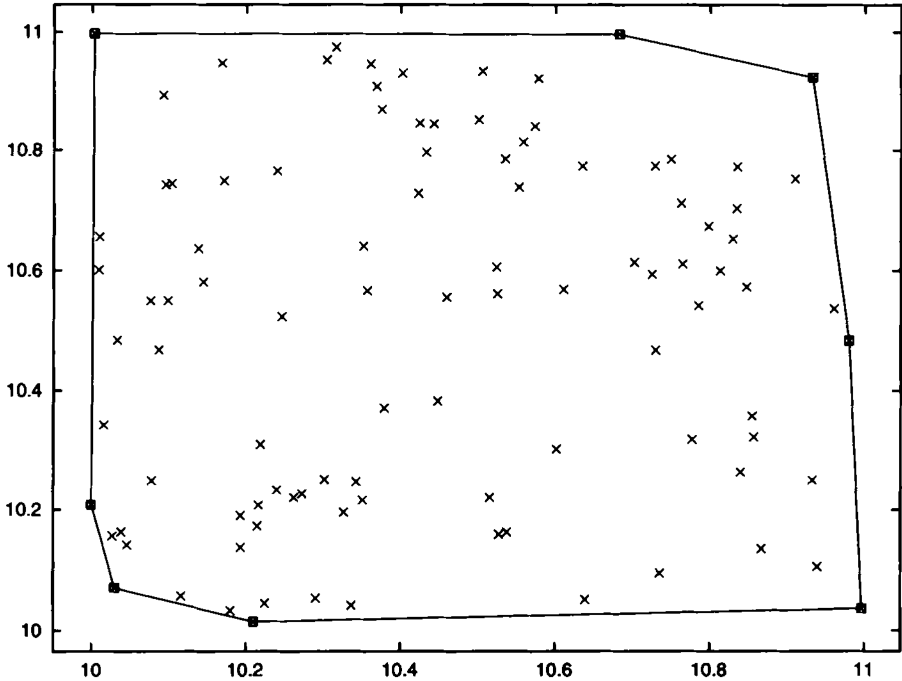


Figure 8.9: Graphics for random ill-conditioned example with 100 not machine representable points

n	nHull	nGR	Sorting		Graham		time
			nESSA	tESSA	nESSA	tESSA	
4000	17	4000	9980	101	4141	49	1178

Table 8.10: Data from random ill-conditioned example with 1000 not machine representable points

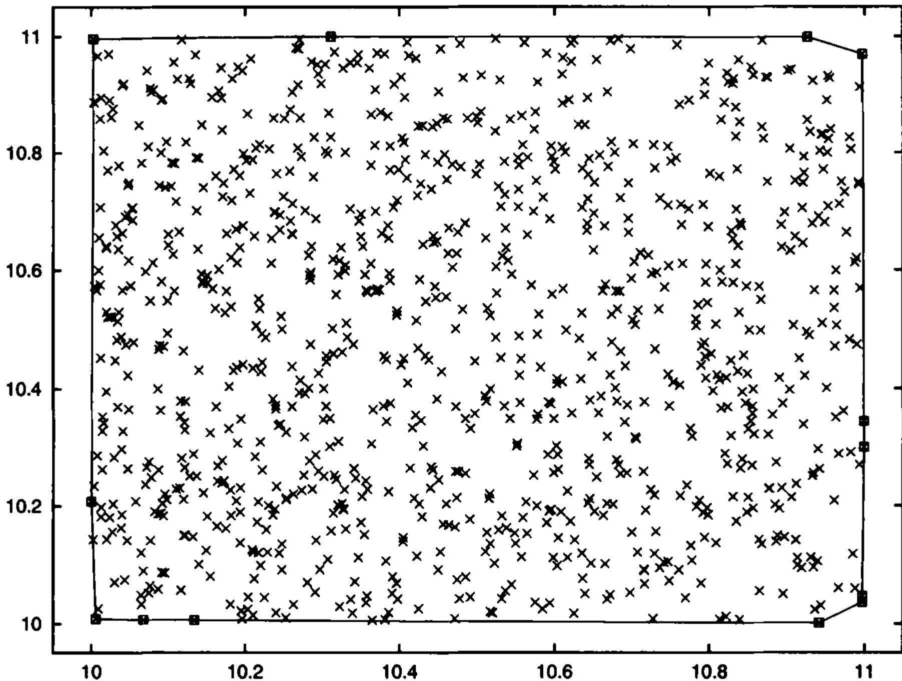


Figure 8.10: Graphics for random ill-conditioned example with 1000 not machine representable points

8.2.5 A More Practical Version of the Algorithm

It was the aim of the previous sections to develop the exact and optimal convex hull construction with a simple accessible version of Graham scan as it was found in O'Rourke's textbook [190]. Since this version furthermore requires calculations with extreme accuracy (high sensitivity of the angular sorting in the preprocessing stage, occurrence of collinearities or almost-collinearities, occurrence of points with 1 ulp distance), an opportunity was given to demonstrate the powerful combination of ESSA and a convex hull method. Hence it is shown completely that also worst cases and extremely ill-conditioned problems cannot crash the methods presented here.

There are of course algorithms for the convex hull computations that are more sophisticated than Graham scan. As examples we mention the algorithms by Kao-Knott[122], Andrew[9], Jaromczyk-Wasilkowski[116] and Chan[24]. In this section we show, using the version of Kao-Knott[122] as an example, that ESSA can be applied as well to more sophisticated convex hull algorithms such that improved, useful and applicable methods results.

The main feature of our version of the improved algorithms is:

1 Instead of an angular sorting a stair-like arrangement is executed, which only needs coordinate comparisons instead of left-turn tests. A further advantage of the stair-like sorting is that "many" points are discovered and dropped that never can become convex hull points. This means that ESSA only has to be applied to the main phase.

2 If the input data is not machine representable and if the points are internally to be represented by machine representable rectangles, one does not need to process all the corners of the rectangles any more, but only one corner of each rectangle such that the computational costs are reduced by 75% compared with the version of Subsec. 8.2.3.

Outline of the Algorithm

First, the data points have to be entered into the program and be replaced by including rectangles of minimum size, say

$$S^{(i)} = [x_L^{(i)}, x_R^{(i)}] \times [y_L^{(i)}, y_R^{(i)}], \quad i = 1, \ldots, n,$$

where the corners are machine representable. These rectangles can be degenerate. The procedure consists of 2 parts:

- **Part 1.** (Preprocessing) generates up to 4 monotone stairs of points containing the final convex hull points, where only coordinate comparisons are used,

- **Part 2.** (Main part) eliminates those points on the stairs that are no convex hull points using the left-turn test with interval arithmetic and ESSA.

Note that the algorithm works on exact machine numbers, since only comparisons, interval arithmetic and ESSA is needed as was the case in Subsec. 8.2.3. Let $S = \{S^{(i)} : i = 1, \ldots, n\}$. Then the algorithm will determine the exact convex hull of S.

The steps of **Part 1** are:

Step 1. Construct the rectangle hull of all the including rectangles, that is, the smallest axes parallel rectangle, RH_S that covers S. The coordinates of the corners are

$$\begin{aligned}
x^{\min} &= \min\left\{x_L^{(i)} : \; i = 1, \ldots, n\right\}, \\
x^{\max} &= \max\left\{x_R^{(i)} : \; i = 1, \ldots, n\right\}, \\
y^{\min} &= \min\left\{y_L^{(i)} : \; i = 1, \ldots, n\right\}, \\
y^{\max} &= \max\left\{y_R^{(i)} : \; i = 1, \ldots, n\right\},
\end{aligned}$$

cf. Fig. 8.11.

Step 2. Determine 4 sub-rectangles of RH_S that already contain the final convex hull of S. For that reason, we need the following anchor points lying on the edges of RH_S:

$$\begin{aligned}
x_B' &= \max\left\{x_H^{(i)} : y_L^{(i)} = y^{\min}, \; i = 1, \ldots, n\right\}, \\
x_B &= \min\left\{x_L^{(i)} : y_L^{(i)} = y^{\min}, \; i = 1, \ldots, n\right\}, \\
x_T' &= \max\left\{x_H^{(i)} : y_R^{(i)} = y^{\max}, \; i = 1, \ldots, n\right\}, \\
x_T &= \min\left\{x_L^{(i)} : y_R^{(i)} = y^{\max}, \; i = 1, \ldots, n\right\}, \\
y_R' &= \max\left\{y_H^{(i)} : x_R^{(i)} = x^{\max}, \; i = 1, \ldots, n\right\}, \\
y_R &= \min\left\{y_L^{(i)} : x_R^{(i)} = x^{\max}, \; i = 1, \ldots, n\right\}, \\
y_L' &= \max\left\{y_H^{(i)} : x_L^{(i)} = x^{\min}, \; i = 1, \ldots, n\right\}, \\
y_L &= \min\left\{y_L^{(i)} : x_L^{(i)} = x^{\min}, \; i = 1, \ldots, n\right\}.
\end{aligned}$$

Hence, x_B' and x_B are the rightmost and leftmost corners, respectively, of the input rectangles that lie on the basis edge of RH_S, etc. Now, let

$$\begin{aligned}
P_B &= \left(x_B, y^{\min}\right), & P_B' &= \left(x_B', y^{\min}\right), \\
P_T &= \left(x_T, y^{\max}\right), & P_T' &= \left(x_T', y^{\max}\right), \\
P_L &= \left(x^{\min}, y_L\right), & P_L' &= \left(x^{\min}, y_L'\right), \\
P_R &= \left(x^{\max}, y_R\right), & P_R' &= \left(x^{\max}, y_R'\right).
\end{aligned}$$

Then the four subrectangles denoted by $R_1, \ldots, R_4$ are defined as follows:

R_1 shall be spanned by P_R and P_B',
R_2 by P_R' and P_T',
R_3 by P_T and P_L', and
R_4 by P_L and P_B,

cf. Fig. 8.12. From the geometry of this figure it is obvious that all convex hull points of S lie in R_1 to R_4. Note that R_1 to R_4 can be degenerate.

Step 3 is applied to each of the 4 subrectangles. It deletes further points that cannot be convex hull points, and it arranges the remaining points of each R_i in a stair-like shape connecting the 2 points which we used to span R_i.

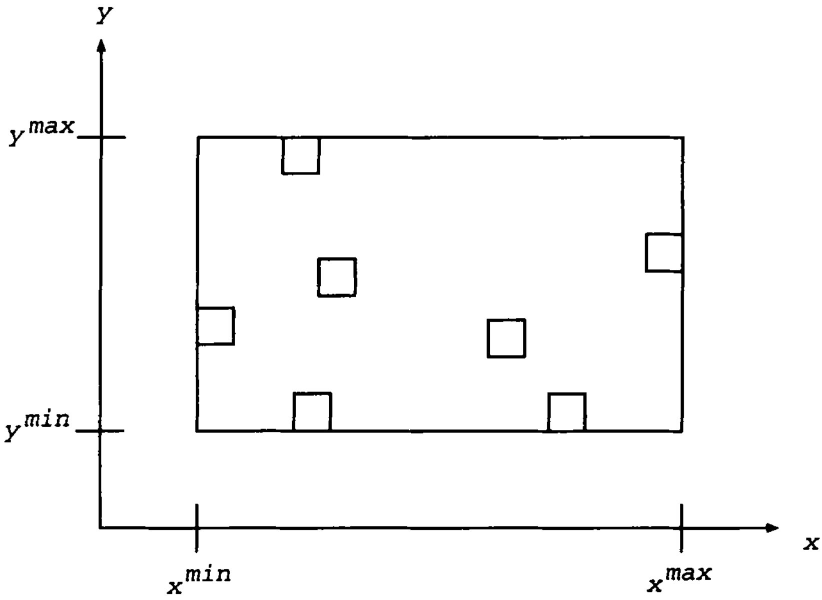


Figure 8.11: Rectangle hull of S

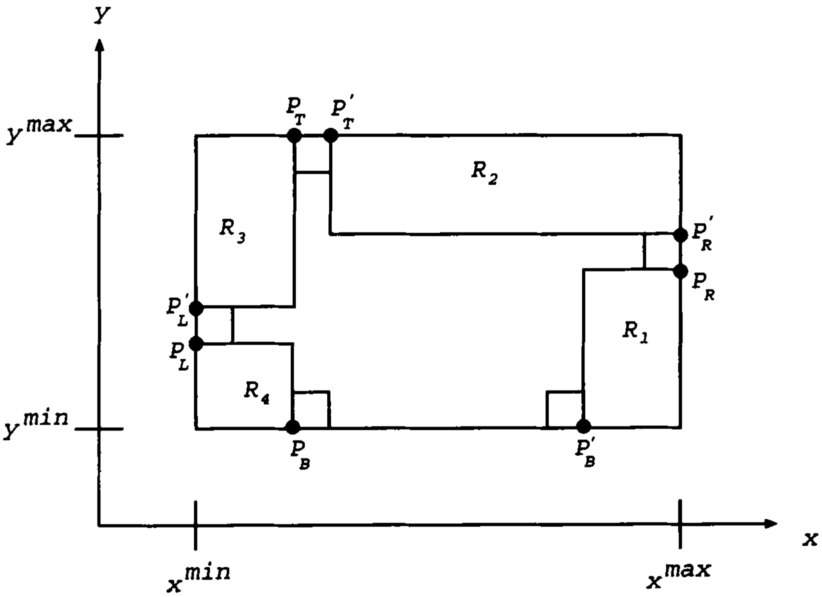


Figure 8.12: The four subrectangles

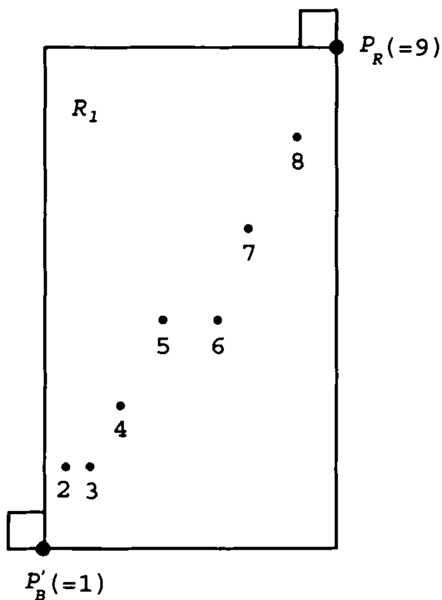


Figure 8.13: Stair in R_1

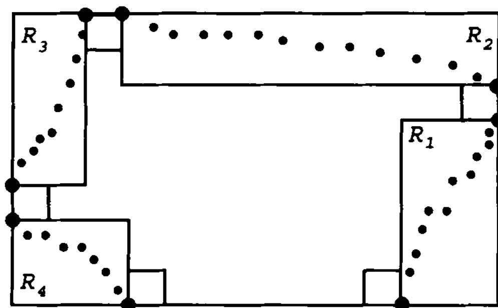


Figure 8.14: The four stairs

The parts of Step 3 focused on R_1 are:

(i) Initialize a list $\mathcal{L}_\infty = \emptyset$.

(ii) For $i = 1, \ldots, n$: If $x_R^{(i)} > x'_B$ and $y_L^{(i)} < y_R$ then enter $(x_R^{(i)}, y_L^{(i)})$ onto $\mathcal{L}_1$.

(iii) Sort $\mathcal{L}_\infty$ with respect to increasing values of $y_L^{(i)}$ obtaining an ordered list $\mathcal{L}_1^* = \{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\}$ of length $m \leq n$ where $y^{(i)} \leq y^{(i+1)}$ for $i = 1, \ldots, m-1$.

(iv) Set $x_{curr} = x'_B$ and initialize a list $\mathcal{M}_1 = \left\{P'_B\right\}$.

(v) For $i = 1, \ldots, m$: If $x^{(i)} > x_{curr}$ then set $x_{curr} = x^{(i)}$ and enter $(x^{(i)}, y^{(i)})$ onto list $\mathcal{M}_1$ after the last element.

(vi) Enter P_R onto list $\mathcal{M}_1$ after the last element.

The list $\mathcal{L}_\infty$ created in (ii) consists of all south-east corners of the rectangles of S which lie in the (topological) interior of R_1. The other 3 corners (in the case of a non-degenerate rectangle) cannot be convex-hull points, and hence they are dropped. The stair-shape finally comes out by (v) since only points with strictly increasing x-values are admitted (cf. Fig. 8.13). The treatment of R_2, R_3 and R_4 is analogous and results in stair shaped lists $\mathcal{M}_2$, $\mathcal{M}_3$ and $\mathcal{M}_4$, respectively (cf. Fig. 8.14).

Part II. (Main part) is nothing more than the proper convex hull construction acting on the ordered lists $\mathcal{M}_1$ to $\mathcal{M}_4$, in exactly the same way as it is shown in Subsec. 8.2.2 with the list $p_0, \ldots, p_{n-1}$. The hull construction can be applied to each of the lists $\mathcal{M}_i$ separately, or to that whole list which $\mathcal{M}_1, \ldots, \mathcal{M}_4$ as sublists (in this order).

Summing up, this algorithm provides on the average a reasonable compromise between coding effort (overhead), computational cost and effective computation. If the sorting of the lists in Part I is executed with a $O(n \log_2 n)$ method (such as Quicksort), the whole algorithm also has worst case cost of $O(n \log_2 n)$.

8.3 Exact Computation of Delaunay and Power Triangulations

8.3.1 Introduction

The requirement for creating a triangulation of a point set or other sets of objects (usually called sites) in the plane is common to scientific fields such as numerical analysis, computer graphics and geographical information systems, to name a few. Roughly spoken, a triangulation of a set S is a subdivision of S into subsets whose bounded faces are triangles. Some introductory references

are [198], [30], [42].

There are many possible triangulations of the given set S which depend on the request which of the sites should be considered as a unit and be put together in one subset. Among these the Delaunay and the power triangulation have special interesting properties. For example it has been shown that the Delaunay triangulation of a terrain map is the triangulation that minimizes the roughness of the resulting terrain, no matter what the actual height data is, cf. [228]. It has also been shown that various graphs (such as the Euclidean minimum spanning tree) defined on a set of points, say S, are subgraphs of the Delaunay triangulation of S. More specifically, the Delaunay triangulation, which is concerned with point sets) is used in numerical analysis, CAGD, etc., and the power triangulation, which subdivides set of sites which are objects in the plane, for example, geometric figures, has found practical applications in crystallography, metallurgy, economics, etc.

The Delaunay triangulation is closely related to the Voronoi diagram in the sense that the Delaunay triangulation is the dual of the Voronoi diagram in the sense of graph theory. Similarly, the power diagram is the dual of the power triangulation. If the Voronoi diagram for a set of n points has been computed then the Delaunay triangulation can be found with complexity $O(n)$.

The problems which are caused by the numerical computation of Delaunay and power triangulations are nearly the same as for other geometrical computations. Thus, there is often a large gap between theoretically correct geometric algorithms and practically valid computer implementations [107]. For example, a recent paper states that rapid progress of computational geometry in the past two decades has resulted in many geometric algorithms, some of which are optimal in the worst-case sense [187]. However, most of them are designed under the assumption that numerical computation can be done precisely. In actual computation errors are inevitable. These errors often generate inconsistencies in the topological structure creating degenerate situations, which can sometimes be worse than numerical errors. This has led to a number of approaches for dealing with the problems caused by the errors. Below they are classified into three groups.

Group A consists of approaches that investigate the bounds of possible errors in the construction. One of the general approaches from Group A is to obtain symbolic bounds on numerical errors so that the stability of the computation can be guaranteed. As shown in [49], it can be difficult to obtain bounds that are tight enough to be useful. Another approach is topologically-oriented and it ensures the consistency of the system topology, i.e. the topology of the triangulation, during the process of computation, rather than controlling the numeric precision [187, 260]. The result is a system free of inconsistency, constructed for any imprecise data set, which is only an approximation of the correct result. Moreover, this result is not free from all the consequences of numerical errors described above.

Group B in our classification includes degeneracy-oriented methods, backward error-analysis methods and epsilon- tolerance methods. The degeneracy-oriented methods employ the idea of avoiding degenerate special cases (which require extensive and correct computation) instead of dealing with them. This can be achieved by conceptual perturbation of the input data [41, 274]. This simplifies the algorithm, but might change the topology of the system, which violates an essential feature of a Voronoi diagram.

One of the methods popular among programmers is the epsilon tolerance approach, where two geometric elements are considered to be at the same location if the distance between them is less than a tolerance ϵ [108, 147]. The shift of any of the points within the ϵ-distance would not change the system topology. The originality of the input set is also lost in this approach, and it is not free from inconsistencies (when, for example, three or more elements come close to each other). Other applications of ϵ-geometry and ϵ-arithmetic have also been considered by some authors [59, 61, 237]. Although the algorithms are numerically stable, they compute approximate solutions.

In the error-analysis approach computational results are classified according to the fuzzy logic as true, false or inconclusive (unreliable) [158, 237]. Only conclusive results are used, and the number of inconclusive ones can be limited, but not completely eliminated, by applying backward error analysis. The computation of the most appropriate error bound is also quite complicated.

Group C, which includes the algorithm proposed in this section, represents a different approach to the problem [120, 123]. Instead of trying to avoid or to deal with the numerical error, the problems are simply eliminated by performing exact operations on the data. This should be done under the reasonable assumption that data items under consideration are already machine numbers. Then, in each step of the algorithm, the exact values of all the components are calculated, which eventually will lead to the correct result. The only question is how expensive this exact computation can be. As stated in one paper devoted to exact computation:

> Exact computation provides simplicity and assured robustness at the expense of the computational efficiency. It provides simplicity in the sense that algorithms map directly to implementations, without need to treat numerical error. Moreover, the handling of geometric degeneracies is vastly simplified by the absence of complex interactions between numerical errors and tests for degeneracies. [123]

In our approach we suggest using fixed-precision floating point arithmetic to solve the efficiency dilemma. We apply the approach to the exact computation of Delaunay and power triangulations. First, we apply the economic variation of ESSA, cf. Ch. IV for the exact computation of the necessary primitives. Then we apply a floating-point filter based on interval analysis to improve the performance of the algorithm. Our method is then tested on the algorithm for

incremental construction of the Delaunay and power triangulations [86]. We show that the worst-case performance of the exact algorithm is the same as for the inexact algorithm, that is $O(n^2)$, since the exact computation of each single primitive takes only $O(1)$. The average time complexity is, therefore $O(n)$ [86]. However, the time required for the exact algorithm presented here to perform the operations is on average 4 times longer than that of the inexact algorithm (as shown below).

Independent of the statistical analysis of the numerical examples, there arises a very surprising insight to triangulations as a side effect. The examples show drastically how large the number of wrong edges is if no error control is implemented. This number ranges from 1% in stable constellations up to 50% at close to degenerate constellations. The conclusion is that *error control is unavoidable if one wants to obtain reliable triangulations.*

8.3.2 Definitions and Methods for Computing Voronoi Diagrams

We consider d-dimensional Euclidean space R^d and a set of points, $S \subseteq R^d$. The Voronoi diagram of S which is one of the most important geometrical data structures in computational geometry, stores proximity information for the set S by dividing the space into *Voronoi regions* $V(p)$ for points $p \in S$ according to the nearest-neighbor rule. They are defined as

$$V(p) = \{x \in R^d | \rho(x,p) < \rho(x,q),\ \forall q \in S \setminus \{p\}\}.$$

where ρ denotes the Euclidean distance [12]. Voronoi regions for different points of S are disjoint. In two dimensions, the Voronoi regions are open polygonal regions. In order to define the Voronoi diagram we collect first all points of the Voronoi regions in the open set

$$Vor(S) = \bigcup \{V(p) | p \in S\}.$$

Each point of the space R^d is then either a point of $Vor(S)$ or a boundary point of $Vor(S)$, but not both. If $d = 2$, that is, the space is the plane, the *Voronoi diagram* of S is defined as the set of boundary points of $Vor(S)$,

$$VD(S) = R^2 \setminus Vor(S).$$

The Voronoi diagram is a regular planar graph of degree three under the assumption that no four points of S are co-circular [188]. The vertices of the graph are called *Voronoi vertices.* They are boundary points of exactly 3 Voronoi regions.

A very important property for the algorithms in the sequel is that each Voronoi vertex is the center of a circle defined by three points of the set S and

that no other points of S lie in the interior of the circle, cf. Sec. 8.3.5. This is known as the *empty circle condition*.

Many interesting properties of Voronoi diagrams are known. A listing of selected properties can be found in [188]. The Voronoi diagram also has numerous applications in different mathematical and industrial fields [188].

Numerical errors in finite-precision arithmetic are inevitable in the construction of the Voronoi diagram, as stated in the paper by Sugihara and Iri [260]. For example, if any coordinate of the vertex of the $VD(S)$ is a rational number, infinite precision floating point numbers may be required to represent the coordinate. This cannot be implemented in finite precision arithmetic. Therefore, the authors introduce a robust topology-oriented incremental algorithm for Voronoi diagram construction, where a higher priority is placed on the topological structure of the diagram rather than on the numerical values. The numerical stability of the algorithm is guaranteed in the sense that no matter how poor the precision may be, the algorithm will always produce a topologically consistent output. The diagram becomes "closer" to the correct Voronoi diagram as the precision becomes higher. The same idea has been applied to the construction by the divide-and-conquer method. However, as was mentioned above, this method of Sugihara and Iris while topologically correct only produces an approximation of the real diagram. Moreover, degenerate input can increase the time required to perform the task as well as degrading the accuracy of the result.

We overcome the hurdle of the rounding errors in the following manner: Instead of computing the Voronoi diagram $VD(S)$ as precisely as possible, we will calculate the exact *Delaunay triangulation* (abbreviated $Del(S)$), which is defined as the straight line dual of the Voronoi diagram. $Del(S)$ is a graph that can be obtained by connecting each two points S whose Voronoi regions share an edge point which is not a vertex. Under the assumption that the points of S are already machine numbers we construct the Delaunay triangulation in average $O(n^2)$ time, no matter how degenerate the input data, S, is, since $VD(S)$ and $Del(S)$ are planar graphs embedded in the plane, so that their complexity is $O(n)$, and once we get $Del(S)$, it can be transformed into $VD(S)$ in $O(n)$ time. We always obtain the exact diagram as a result, which in this connection means, that the resulting $VD(S)$ will be topologically correct. Numerically it may, however, be only an approximation of the real $VD(S)$.

Up to now the geometric sites we were dealing with were points. We turn now to sites which already are geometrical objects, i. e. we focus on spheres in the space R^d . Let S therefore be a set of such spheres, then a diagram which corresponds to the Voronoi diagram is the power diagram. (We use the notation S for sets of spheres too since the steps of the algorithm we will establish are the same for both kinds of sets.)

First we need some measure for the distance of a point x of the space to a given sphere p. This is done by the *power function* which is defined to be

$$pow(x,p) = (x - c(p))^T(x - c(p)) - r^2(p)$$

(see [12]) where $x \in R^d$, but x does not lie in the interior of the sphere, and where $c(p)$ and $r(p)$ are the center and radius of the sphere.

The power function can be interpreted in the following manner: Draw a tangent line from x to the sphere p. Let the tangent touch p, say at the point y. Then, by the law of Pythagoras, $\sqrt{pow(x,p)}$ is the Euclidean distance from x to y. Or, with other words, $\sqrt{pow(x,p)}$ represents the distance from the sphere p to the point x outside the sphere measured along a tangent line through the point x.

Corresponding to the Voronoi regions in the case of point sites, one considers in the case, where the sites p are spheres, the *power cells* of $p \in S$. They are defined by

$$cell(p) = \{x \in R^d | pow(x,p) < pow(x,t),\ \forall t \in S \setminus \{p\}\}.$$

In order to define the power diagram of S we collect first all points of the power cells in the open set

$$Pcl(S) = \bigcup \{V(p) | p \in S\}.$$

Each point of the space R^d is then either a point of $Pcl(S)$ or a boundary point of $Pcl(S)$, but not both. If $d = 2$, that is, the space is the plane and the spheres are circle (lines), the *power diagram* $PD(S)$ of S is defined as the set of the boundary points of $Pcl(S)$,

$PD(S) = R^2 \setminus Pcl(S)$.

In the planar case the sites of S are circles. In this case the power diagram of S is a regular planar graph of degree three under the assumption that no four sites of S are *co-circular* [12], that is, there is no solid circle (the boundary of which need not be a site!) that is outside of the four circles of S and touches each of the four circles.

When the straight-line dual graph of power diagram $PD(S)$ is drawn between the centers of the circles, it yields a planar triangulation of set of sites, which we will refer to as a *power triangulation* (abbreviated $Pow(S)$ in the sequel).

8.3.3 Methods for Constructing Delaunay and Power Triangulations

The correctness of many computational geometry algorithms depends on the exact computation of one or more simple algebraic expression. The algorithm

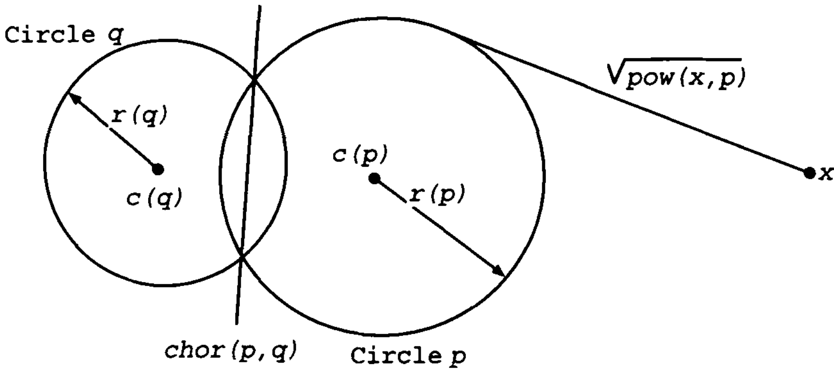


Figure 8.15: Power diagram

for the exact computation of the Delaunay and power triangulation is a good illustration of this fact. We will focus our attention on two methods for Delaunay triangulation construction, namely, the *divide-and-conquer* and the *incremental* method. The well known divide-and-conquer algorithm, introduced by Guibas and Stolfi [85], runs in optimal $O(n \log n)$ time. The simple modification of this algorithm suggested by Dwyer [39] runs in $O(n \log \log n)$ expected time, improved to $O(n)$ by Katajainen and Koppinen [124]. Both algorithms use only two geometric primitives: the CCW (Counter Clock Wise) orientation test and the INCIRCLE test. The CCW orientation test is used to find for a current point of the input set that triangle of the actual state of the triangulation which contains the point. The INCIRCLE test enables the determination of whether a triangle which occurs during the computation belongs to $Del(S)$. The algorithms for constructing the power triangulation are similar to those for the Delaunay triangulation. Only the INCIRCLE test has to be modified for the power triangulation while the CCW test remains the same.

Among the incremental algorithms we can distinguish algorithms based on incremental construction and on incremental search. The *incremental construction algorithms* (one of which we have chosen for the tests) starts with a triangle the area of which covers S. This triangle is seen as a Delaunay triangulation of itself which commences the initialization of a recursive process. The recursion consists of adding one point of S after the other and subdividing the triangle at the same time maintaining the Delaunay triangulation property till all points of S are subsumed. Then the triangulation of S is terminated after deleting the initializing triangle, cf. for example, [86, 237].

The *incremental search algorithms* [40, 147] start already with one triangle of the final triangulation, and grow the diagram by one valid triangle after the other, till all points of S are subsumed.

We note that only two primitives, the CCW and the INCIRCLE test, are used in the described approaches. The result of each test depends on the

computation of the sign of a determinant. If we want to construct a correct Delaunay triangulation, the value of the determinant has to be computed exactly. The approaches used to solve this problem in the literature are based on an arbitrary-precision arithmetic or integer arithmetic or both. Jünger et. al. [120] represent the input data as integers in the range from 0 to M and prove that the result of the determinant computation lies in the range from 0 to $6M^4$. Karasick et. al. [123] use adaptive-precision rational arithmetic. They represent the data as integers and use an interval filter to improve algorithm performance. The algorithm is four times slower than the floating-point implementation. Fortune and Wyk [62] also implement a number of Delaunay triangulation algorithms in adaptive-precision arithmetic with interval analysis as filter. They test their model on some incremental and divide-and-conquer algorithms and show that the performance of the algorithm has a cost close to that of floating-point arithmetic.

There are, however, some disadvantages to using integer and adaptive-precision arithmetic. One of these comes from the fact that not many processors can perform operations on the arbitrary precision numbers. Therefore, these operations have to be implemented at the software level and, consequently, they will be slower than those performed at the hardware level. Another problem with integer arithmetic is when the precision of the input data increases, the amount of space and time required for the algorithm grows exponentially, in proportion to the complexity of the expression. For example, the space required to store the result of calculation of a 4×4 determinant is 4 times larger than that for its operands. In fact, if the operands are of single precision, the result is of the maximum possible precision in most computers - quadruple.

In our case we are interested in developing robust and reliable algorithms for the computation of *Del*(S) and *Pow*(S). We therefore chose the incremental algorithm of McLain [152] for the exact construction of the Delaunay and the power triangulation, based on standard floating-point arithmetic. The input data, S is represented by machine numbers. This is a reasonable assumption since a triangulation is usually computed as an intermediate part of a larger computation such that the input data for the triangulation is the output of a previous computation.

The incremental algorithm has complexity $O(n^2)$, it is simple to implement and it is based on only two primitives, the CCW orientation test and the INCIRCLE test. Both of the two primitives require the exact computation of the sign of determinants which is done by ESSA. As already done at other geometrical computations we improve the performance of the algorithm by using an interval filter before executing ESSA. The reason for doing this is that if the computation of the CCW or the INCIRCLE tests is well-conditioned, i.e., the configurations are not close to being degenerate, then the less expensive interval evaluation of the determinants will provide the exact sign.

The approach has been tested on the algorithm for incremental construction of the Delaunay triangulation described in [86]. The performance of the

algorithm is verified by computational experiments described below. The fixed mantissa length is never exceeded during the computation and can be single or double precision, etc., depending on the purpose. For simplicity, we assume *single precision computation* with mantissa length t. The algorithm we introduce will render the exact result provided the input data consists of machine numbers.

8.3.4 Exact Computation of the CCW Orientation Test

The CCW orientation test is almost identical to the left-turn test, cf. Ch. IV and is used to locate that triangle among the current triangles during the computation which contains in its interior a given point, which is, depending of the class of sites, either the next point of S or the midpoint of the next circle of S to be processed. This test is seen as a primitive and decides whether a point lies to the left, right or on a directed line defined by two other points. The result of the test can be calculated as the sign of a 3x3 determinant. We assume the coordinates of the points from the input set are machine numbers. Let p_1, p_2, p_3 be 3 points in the plane such that $p_i = (x_i, y_i), i = 1, 2, 3$. If $\overline{p_1 p_2}$ denotes the directed straight line segment from p_1 to p_2, then p_3 is to the left, on or to the right of $\overline{p_1 p_2}$ iff

$$D = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix}$$

is positive, zero or negative, cf. the description of the left-turn test in Ch. IV. We know that the determinant is already a sum,

$$D = x_1 y_2 + x_2 y_3 + x_3 y_1 - x_1 y_3 - x_2 y_1 - x_3 y_2. \tag{8.2}$$

Since $p_i = (x_i, y_i), i = 1, 2, 3$ are represented in single precision arithmetic, the products appearing in (8.2) can be computed exactly using double precision arithmetic. Thus, we can apply a double precision version of ESSA to the sum (8.2) and get the correct sign. In order to achieve better performance, we use again the interval filter. That is, we first apply interval arithmetic to the expression (8.2). Thus, if the (single precision) interval arithmetic computation of D gives the interval result D^I, then,

$$\begin{array}{lll} \text{if} & D^I > 0 & \text{then } D > 0, \\ \text{if} & D^I < 0 & \text{then } D < 0, \\ \text{if} & D^I = 0 & \text{then } D = 0. \end{array}$$

When $0 \in D^I$ it is not possible to decide the sign of D with the chosen accuracy of the representation of the intervals and we apply ESSA to compute the sign of D.

8.3.5 Exact Computation of the INCIRCLE Test

We first consider point sites S. When the circle defined by three given points $p_1, p_2, p_3 \in S$ does not contain any other point from S, the empty circle condition is satisfied, the interior of the triangle p_1, p_2, p_3 does not contain any further point of S, and the triangle p_1, p_2, p_3 is a Delaunay triangle.

The INCIRCLE test enables to check whether the current triangles which occur during the computation, already belong to *Del*(S). It is performed on a diagram which consists of a quadruple of points generating 2 adjacent triangles. The test decides whether the inner edge of the two triangles remains or has to be flipped so that the triangulation of these 4 points satisfies the conditions of the Delaunay triangulation.

As the algorithm proceeds recursively, the INCIRCLE test is applied to the current state of the recursion so that S will then mean the current state of the point set.

In order to determine which situation occurs, the sign of a 4×4 determinant has to be calculated. Let $p_i = (x_i, y_i), i = 1, \ldots, 4$ be four points in the plane and assume p_1, p_2, p_3 (not collinear) define a circle C. Then the relationship of p_4 to C is determined by the sign of the determinant

$$D = \begin{vmatrix} x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \\ x_4 & y_4 & x_4^4 + y_4^2 & 1 \end{vmatrix}.$$

Assume that p_1, p_2, p_3 in this order lie clockwise on the circle. (This is checked with the CCW test.) Then

$$\begin{array}{lll} \text{if} & D > 0 & \text{then } p_4 \text{ is inside } C, \\ \text{if} & D = 0 & \text{then } p_4 \text{ is on } C, \\ \text{if} & D < 0 & \text{then } p_4 \text{ is outside } C. \end{array}$$

If now $D > 0$, the diagram consisting of the 2 adjacent triangles $\{p_1, p_2, p_3\}$ and $\{p_1, p_2, p_4\}$ is a Delaunay triangulation of the set $\{p_1, p_2, p_3, p_4\}$. If $D < 0$, the diagram is not a Delaunay triangulation of this set, but the diagram consisting of the 2 adjacent triangles $\{p_3, p_4, p_1\}$ and $\{p_3, p_4, p_2\}$ is (flipping operation), cf. [30].

Numerically, we have to determine the sign of D. When the determinant is multiplied through then products of the form

$$x_i y_j (x_k^2 + y_k^2) = x_i y_j x_k^2 + x_i y_j y_k^2 \tag{8.3}$$

result. Each product of the form $x_i y_j x_k^2$ requires quadruple precision for getting exact results with ESSA if the points are single precision quantities. However, those expressions could also be accomplished by 4 double precision quantities

without too much mantissa manipulations. This is done as follows: we start with $x_i y_j x_k$ in single precision, compute the products $x_i y_j$ and x_k^2 in double precision, but split each of them immediately in the sum of two single precision numbers,

$$x_i y_j = (x_i y_j)_L + ((x_i y_j)_R, \quad x_k^2 = (x_k^2)_L + (x_k^2)_R. \tag{8.4}$$

Finally, we execute the four products

$$(x_i y_j)_\nu (x_k^2)_\mu, \quad \nu, \mu = L, R \tag{8.5}$$

in double precision. Their exact sum is $x_i y_j x_k^2$. We perform the same operations to calculate $x_i y_j y_k^2$:

$$x_i y_j = (x_i y_j)_L + ((x_i y_j)_R, \quad y_k^2 = (y_k^2)_L + (y_k^2)_R. \tag{8.6}$$

Finally, we execute the four products

$$(x_i y_j)_\nu (y_k^2)_\mu, \quad \nu, \mu = L, R. \tag{8.7}$$

Hence, the determinant is the sum of 192 double precision quantities and the computation of the sign of D can be done exactly by ESSA. When computing the determinant, we first apply interval arithmetic as explained in the previous section and ESSA is only executed when an inconclusive result occurs.

Let us now turn to the case where the sites are circles. that is, the power triangulation is the target of the computation. Then the form of the 4x4 determinant in the INCIRCLE test is changed [66]:

$$D = \begin{vmatrix} x_1 & y_1 & x_1^2 + y_1^2 - r_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 - r_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 - r_3^2 & 1 \\ x_4 & y_4 & x_4^4 + y_4^2 - r_4^2 & 1 \end{vmatrix}.$$

where $c_i = (x_i, y_i), \ i = 1, \ldots, 4$ are the centers of circles, and $r_i, \ i = 1, \ldots, 4$ are the radii of the circles. let $p = (c, r)$ be a circle (not necessarily belonging to S) which does not contain the circles p_1, p_2, and p_3 in its interior (but p can belong to the interiors of one or more of the circles p_1, p_2, and p_3) but which touches the circles p_1, p_2, and p_3. If such a circle exists it is uniquely defined.

Then the sign of the determinant reflects the following cases: Let the points c_1, c_2, and c_3 be clockwise ordered. If $D < 0$ then the interiors of the two disks which have p_4 and p as boundaries are not disjoint. If $D = 0$ then p_4 touches p, but the interiors of the related disks are disjoint (and p_1, p_2, p_3 and p_4 are cocircular). If $D > 0$ then the disks which are generated by p_4 and p are disjoint

We have to compute the sum of products in the form

$$x_i y_j (x_k^2 + y_k^2) = x_i y_j x_k^2 + x_i y_j y_k^2 - x_i y_j r_k^2 \tag{8.8}$$

which is done in the same way as for *Del*(S) and the determinant will be computed as a sum of 288 double precision quantities.

8.3.6 Complexity Analysis for the Primitives

We analyse and compare the expected costs of the CCW and INCIRCLE primitive calculation for 3 different implementations, the straight implementation only using single precision machine numbers, the implementation using interval arithmetic and the implementation which uses ESSA. In the analysis of the implementation with ESSA we assume that the average number of iterations performed by ESSA is one half of the initial number of summands (experiments confirm this assumption). The total sizes of lists of summands are $l = 6$ for the CCW orientation test, $l = 192$ for the INCIRCLE test in the Delaunay triangulation and $l = 288$ for the INCIRCLE test in the power triangulation. At each iteration of ESSA, at most two additions are performed.

CCW Orientation Test

In the straight implementation the CCW test is calculated by the following formula:

$$CCW(p_1, p_2, p_3) = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1).$$

This involves 2 multiplications and 5 additions.

In the interval implementation, the calculations are performed using the same formula, but the operands are represented by intervals. Since an interval addition requires 2 scalar additive operations, and an interval multiplication requires 4 scalar multiplications and comparisons, the total number of operations for interval CCW test is 10 additions and 8 multiplications. On the average, the interval implementation is approximately 4 times slower than the direct implementation, because most of the time is spent doing multiplications. For the ESSA implementation, the preparation of the list of summands (eqn. (2)) requires 6 multiplications. The ESSA itself requires approximately $2*(6/2) = 6$ additions.

INCIRCLE Test for the Delaunay Triangulation

In the straight implementation with machine numbers, the INCIRCLE test is calculated by the following formula:

$$\begin{aligned} &INCIRCLE(p_1, p_2, p_3, p_4) \\ &\quad = D(p_1) * CCW(p_2, p_3, p_4) - D(p_2) * CCW(p_1, p_3, p_4) \\ &\quad + D(p_3) * CCW(p_1, p_2, p_4) - D(p_4) * CCW(p_1, p_2, p_3) \end{aligned}$$

where $D(p_i) = x_i^2 - y_i^2, \quad i = 1, \ldots, 4$. As one can count, this requires 27 additions, 12 multiplications and 8 squares.

The interval implementation of INCIRCLE test will therefore require $54 = 2 * 27$ additions and $64 = 4 * 12 + 2 * 8$ multiplications (the interval square requires only 2 multiplications).

Primitive	Straight-forward		Interval		ESSA	
	+	*	+	*	+	*
CCW	5	2	10	8	6	6
INCIRCLE(Delaunay)	27	20	54	64	192	288
INCIRCLE(Power)(S)	31	24	62	72	288	432

Table 8.11: Total number of operations

For the implementation with ESSA one has to build 48 products of the form $x_i y_j x_k^2$. Each of the four products need 6 multiplications. Therefore, 288 multiplications are required in total to prepare the list of summands. The execution of ESSA will require $2 * (192/2) = 192$ additions on average.

INCIRCLE Test for the Power Triangulation

In the straight implementation the INCIRCLE test for the power triangulation is calculated by the following formula:

$$\begin{aligned} &INCIRCLE(p_1, p_2, p_3, p_4) \\ &\quad = D(p_1) * CCW(p_2, p_3, p_4) - D(p_2) * CCW(p_1, p_3, p_4) \\ &\quad + D(p_3) * CCW(p_1, p_2, p_4) - D(p_4) * CCW(p_1, p_2, p_3) \end{aligned}$$

where $p_i = (c_i, r_i)$, $c_i = (x_i, y_i)$, $D(p_i) = x_i^2 - y_i^2 - r_i^2$, $i = 1, \ldots, 4$. This requires 31 additions, 12 multiplications and 12 squares.

The interval implementation of the INCIRCLE test requires $62 = 2 * 31$ additions and $72 = 4 * 12 + 2 * 12$ multiplications.

For the ESSA implementation one has to build 72 products of the form $x_i y_j x_k^2$ Each of the 4 products needs 6 multiplications. Therefore, 432 multiplications are required in total to prepare the list of summands. The execution of ESSA will require $2 * (288/2) = 288$ additions on average. The performance results for all the tests are summarized in Table 8.11.

Both the straight-forward and the interval implementations use single precision operations, while the ESSA implementation uses double precision additions and single precision multiplications.

8.3.7 The Main Scheme of the Incremental Algorithm

We only want to sketch the algorithm in order to show where the primitives and hence demand for exact computation are located in the computation. I. e., we don't discuss sophisticated implementations and versions which are worked out completely.

The algorithm we consider is based on the incremental construction method of the Delaunay triangulation [86]. It works for sets S of point sites as well

as for classes of circle sites. In the latter case the algorithm is applied to a set S which consists of the midpoints of the circles, and only the INCIRCLE test has to be modified. Therefore it suffices to deal only with the Delaunay triangulation in the sequel.

The idea of the algorithm is the following: Let $S = \{p_1, ..., p_n\}$ be the set of points (or the set of midpoints in the case of the power triangulation of sets of circles). Then a triangle with vertices, say a, b, c not belonging to S has to be found the interior of which covers the set S completely. Now let $S_i = \{a, b, c, p_1, ..., p_i\}$ and τ_i the Delaunay triangulation of S_i for $i = 0, ..., n$. The essential feature of the incremental construction is that τ_i can be constructed using the two primitives if τ_{i-1} is known ($i = 1, ..., n$). Since τ_0 is the Delaunay triangulation of S_0 and consists therefore of the triangle a, b, c, the start of the recursion is settled too. The final state of the recursion is τ_n. Removing the points a, b, c and all edges with these points as endpoints from this diagram gives the Delaunay triangulation of S.

The input for the algorithm are the coordinates of the point sites of the given set S in the plane resp. the coordinates of the midpoints of the circles. The radii of the circles are hidden input parameters as they occur in the INCIRCLE test only.

ALGORITHM 23 *(Delaunay Triangulation)*

Input: *The set $S = \{p_1, ..., p_n\}$ of points in the plane where no 4 points are cocircular.*

Step 1. *Find three points a, b, c such that all points of S lie in the interior of the triangle defined by the three points.*

Step 2. *Set $i = 0$ and initialize τ_i as the triangulation consisting of a, b, c.*

Step 3. *Set $i = i + 1$.*

Step 4. *Find one triangle of τ_i, say u, v, w, containing p_i by applying CCW tests in an appropriate manner (where by triangle an atomar triangle is meant, which contains no points of τ_i in its interior).*

Step 5. *If*

p_i lies in the interior of triangle u, v, w

then

- *subdivide the triangle u, v, w by connecting p_i with each of the points u, v, w. by a straight line. Let τ_{i+1} be that triangulation which arises from τ_i by adding p_i together with the three new triangles. (This triangulation need not be a Delaunay triangulation. It has first to be*

checked whether the edges of the triangle u, v, w satisfy the neighborhood relationship between the vertices also in τ_{i+1}. If not, the edges are switched with the edges of the adjacent triangles. This is done by calling the following procedure, which is explained below.)

- *call LEGALIZEEDGE($\overline{uv}, \tau_{i+1}$)*
- *call LEGALIZEEDGE($\overline{vw}, \tau_{i+1}$)*
- *call LEGALIZEEDGE($\overline{wu}, \tau_{i+1}$)*

else (p_i lies on an edge of the triangle u, v, w, say on $\overline{uv}$; let z be the third vertex of the adjacent triangle which has p_i as an edge point)

- *subdivide the triangles u, v, w and u, v, z by connecting p_i with each of the points w and z by straight lines. Let τ_{i+1} be that triangulation which arises from τ_i by adding p_i together with the four new triangles. (As above, this triangulation need not be a Delaunay triangulation and the check for the correct neighborhood relationship has to be done for the four outer edges of the triangle pair u, v, w and u, v, z.)*
- *call LEGALIZEEDGE($\overline{uz}, \tau_{i+1}$)*
- *call LEGALIZEEDGE($\overline{zv}, \tau_{i+1}$)*
- *call LEGALIZEEDGE($\overline{vw}, \tau_{i+1}$)*
- *call LEGALIZEEDGE($\overline{wu}, \tau_{i+1}$)*

Step 6. *Set $i = i + 1$.*

Step 7. *If $i < n$ go to Step 4.*

Step 8. *Discard a, b, c and all incident edges from τ_n. The resulting diagram is the Delaunay triangulation of S.*

An essential part of the algorithm is the flipping operation which potentially has to be executed when a new point is added to the actual state of the triangulation in order to maintain the Delaunay property of the triangulation. This is done by the procedure LEGALIZEEDGE:

Let $\overline{uv}$ be an interior edge of any state of the triangulation, say τ , which might occur during the execution of the algorithm. Then there are two adjacent triangles which share $\overline{uv}$ as edge, say u, v, w and u, v, z. It is well possible that the edge $\overline{uv}$ does not satisfy the neighborhood relationship which is required for the Delaunay triangulation. Then the edge $\overline{uv}$ is replaced by the edge $\overline{wz}$, which then satisfies the neighbourhood relationship, cf. Fig. 8.16. This process is frequently called *flipping.*

The procedure consists of the following steps (its only a finite number of steps, cf. [30] so the procedure will terminate):

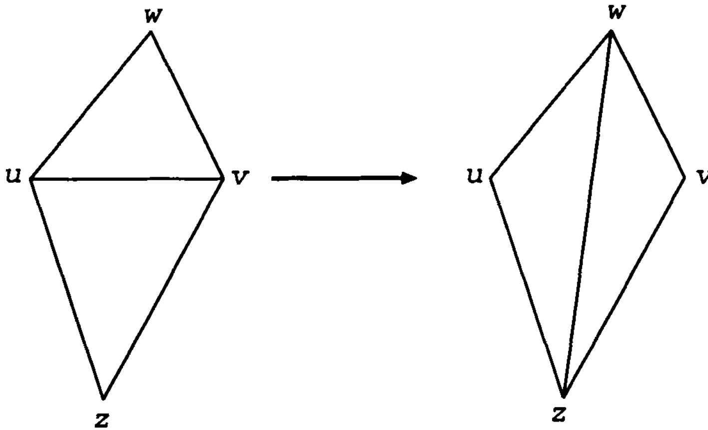


Figure 8.16: Flipping the edges

LEGALIZEEDGE *($\overline{uv}, \tau$) (Legalize an edge or flip it)*

Input: *An inner edge $\overline{uv}$ of a triangulation τ, and τ itself. Let u, v, w and u, v, z be the two triangles of τ that share the edge $\overline{uv}$, and u, v, w (in this order) be clockwise oriented.*

Step 1. *Set $D = INCIRCLE(u, v, w, z)$, cf. Sec. 8.3.5.*

Step 2. *If $D < 0$ (that is, the edge $\overline{uv}$ does not obey the neighborhood relationship and has to be flipped)*

then

- *Replace $\overline{uv}$ with $\overline{wz}$ (Flipping. Note that τ will be changed by the flipping process!)*
- *call LEGALIZEEDGE($\overline{uz}, \tau$)*
- *call LEGALIZEEDGE($\overline{zv}, \tau$)*

Output: *Either the triangulation of the input or a triangulation which arose from the input triangulation by flipping one or more edges which then satisfy the neighborhood relationship.*

There is not too much to say about the comparison of the numerical costs of the straight implementation which performs on single machine numbers, of the algorithm with the exact implementation with uses ESSA and interval arithmetic as filter. The complexity order of ESSA and of the interval arithmetic operations is independent of the number of sites, say N, of S. Since

the complexity of the exact algorithm is $O(N)$, and the worst-time complexity is $O(N^2)$, the same holds for the straight implementation, because the two implementations are distinguished only by ESSA and interval instead of real arithmetic operations.

8.3.8 Test Results

Many examples were computed in order to investigate the numerical performance of the *exact algorithm* (the one with interval filter and ESSA). For comparison, the performance of the straight algorithm (that is the plain implementation of the algorithm in machine numbers without caring about error control). The experiments were conducted on a 486DX2/66 PC.

In the first series of experiments we tested the two implementations on different distributions of points in the following input sets (Sites were points):

- Sites are randomly distributed in a rectangular area (further referenced as random distribution)
- Sites are distributed at the nodes of a rectangular grid (further referenced as grid distribution)
- Sites uniformly distributed on a circle line (further referenced as circle distribution)
- Sites uniformly distributed on the boundary of a square (further referenced as square distribution)

The number of tests was 5 for each class of sets. Each set consisted of 100 points. A perturbation parameter, P was also chosen for the tests except for the random distribution. It set the distance from the original points of the input data to perturbations in the distance of P. The perturbation was randomly applied to the input set in question before the computation commenced in order to generate input data with different rate of degeneracy. Note that grid, circle, and square distributions are degenerate. The parameter was in the range from $P = 0.1$, where the points were significantly perturbed from their original positions, to $P = 1E - 9$, where the points were only slightly shifted. For $P = 0.1$ the distribution practically becomes random.

When P decreases the number of degenerate situations (when four points are cocircular) increases. We varied the value of perturbation parameter from $1E - 9$ (when coordinates of points are perturbed only in the last digit of mantissa) to 0.1 (when points perturbed in almost all digits of the mantissa). With small perturbations the number of degenerate situations in the input set is significant, since the original data in the three perturbation cases is already degenerate, and with large perturbations the distribution of points is close to random distribution. If a perturbation had been applied to the random

P	Grid (261 edges)				Grid (197 edges)			
	ESSA	TE	TD	WE	ESSA	TE	TD	WE
1E-1	0	5.71	2.92	0	1	4.67	2.81	0
1E-2	0	5.88	2.92	0	1	4.67	2.80	0
1E-3	8	6.27	2.96	0	17	4.89	2.75	0
5E-4	19	6.32	2.91	1	27	5.66	2.69	0
1E-4	60	7.64	2.91	1	81	6.97	2.64	0
5E-5	66	8.07	2.80	2	114	10.16	2.69	1
1E-5	80	8.46	2.86	9	314	11.54	2.69	3
5E-6	80	8.62	2.86	13	371	13.18	2.75	11
1E-6	83	8.68	3.24	32	362	17.75	2.80	41
5E-7	81	8.52	2.91	30	345	16.85	2.69	54
1E-7	81	8.02	2.98	32	357	17.35	2.69	73
5E-8	81	7.91	2.86	28	354	17.84	2.80	75
1E-8	81	7.91	2.97	27	410	18.89	2.90	80
1E-9	81	7.53	2.92	26	337	18.18	2.74	87

Table 8.12: Algorithm performance for perturbation values on grid

distribution it would have been still a random distribution equally, how the size of P would have been. Therefore the input data with random distribution remains unperturbed.

We compare the performance of the exact with that of the straight implementation of the incremental method. The results of the tests are presented in Tables 8.12 and 8.13. In the table P stands for the perturbation parameter, ESSA for the number of ESSA calls, TE for the computation time of the exact implementation (in seconds), TD for the time of the straight implementation, and WE for the number of wrong edges produced by the straight implementation.

Although it would be sufficient for an exact computation if the input date would be single precision machine numbers, the tests were executed with double precision machine numbers. The reason is that the statistics for the computational performance in dependency of the perturbation grants more insight as the perturbation is applied to each digit of the mantissa, and the more digits are available the more obvious the dependency can be demonstrated.

The number of ESSA calls grows when the perturbation is decreased. Since ESSA is the most time consuming test, the growth of TE depends entirely on this number. The first phenomenon that we can note is that the number of

	Square (225 edges)			
P	ESSA	TE	TD	WE
1E-1	0	5.15	2.97	0
1E-2	0	5.16	3.18	0
1E-3	0	6.35	3.07	0
5E-4	4	6.09	3.08	0
1E-4	8	6.19	3.07	0
5E-5	23	6.01	3.07	0
1E-6	46	6.09	3.02	1
5E-6	46	6.08	3.08	2
1E-6	52	6.09	3.08	6
5E-7	64	6.18	3.07	9
1E-7	63	6.17	3.07	18
5E-8	61	6.09	2.97	24
1E-8	60	6.09	3.07	25
1E-9	58	5.55	2.97	27

Table 8.13: Algorithm performance for perturbation values on square

ESSA calls remains almost constant when the perturbation parameter reaches some critical value $P_{\max}$. The related statistics is printed out for grid distributions in Tables 8.12 and 8.13 At this value of the perturbation the straight implementation is no longer able to produce reliable results (because it considers many cases as degenerate or close to degenerate), and therefore the number of wrong edges increases. As it can be seen from Tables 8.12 and 8.13 $P_{\max}$ lies between $5E-7$ till $E-5$ depending on the actual constellation.

Figure 4 illustrates the time ratio of the exact and the straight implementation in dependence on the value of the perturbation parameter. The time ratio is defined as

$Tratio = \mathrm{TE}/\mathrm{TD}$.

As we can see, the ratio remains constant after the perturbation reaches the value $P_{\max}$, since the ratio also depends on the number of ESSA calls. Another phenomenon is that for the circle distribution the ratio is much higher than that of the grid and square distributions. This can be explained by the fact, that any four points in the circle distribution are cocircular, hence the interval filter gives the inconclusive result for almost all INCIRCLE tests. We have included the ratio for the random distribution. Since the number of ESSA calls performed for the random distribution is very low (the random distribution does not depend on the perturbation parameter), the ratio is entirely determined by the efficiency of interval filter.

The number of wrong edges increases significantly as the perturbation parameter decreases. The straight algorithm encounters more pseudo-degenerate

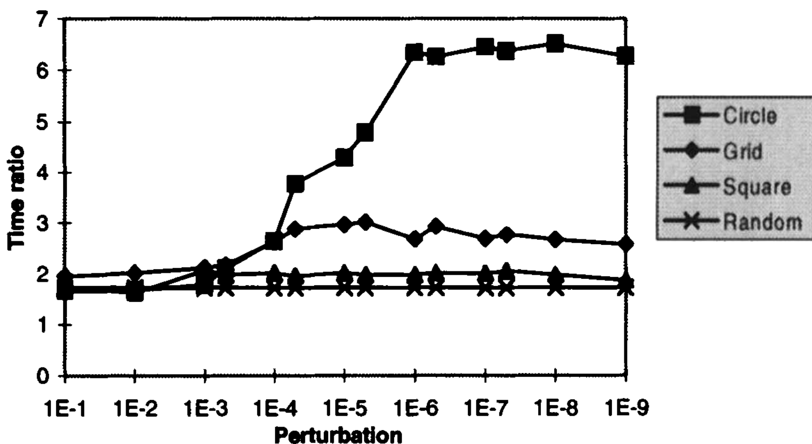


Figure 8.17: Time ratio vs. perturbation

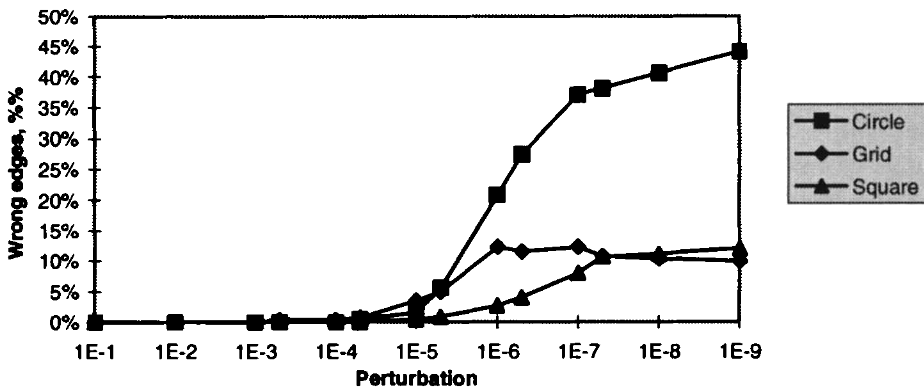


Figure 8.18: Percentage of wrong edges vs. perturbation

situations. Also sometimes it recognizes actual degeneracy and performs incorrect CCW tests, which results in wrong edges inserted into the triangulation. This in turn leads to the increase of wrong edges (see Figure 8.17 and Tables 8.12 and 8.13). The number of wrong edges reaches almost 50% for small perturbations in the circle distribution. The number of wrong edges in the random distribution is usually very low (less than 1%).

In the next series of experiments we investigated how the performance of the exact implementation is affected by the number of points in the input set. Figure 8.19 illustrates the results for different point distributions. The time ratio grows slowly and tends to a constant. For example, for the grid distribution this constant is approximately 10. The ratio is lower for smaller sets of points because more time is spent in straight and exact algorithms for

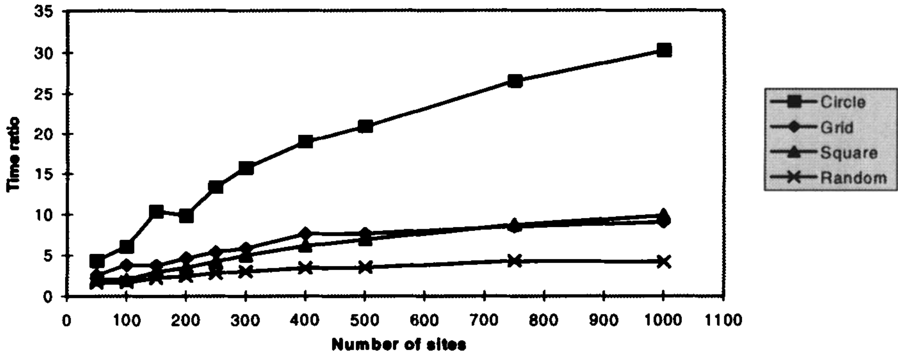


Figure 8.19: Time ratio vs. number of points

miscellaneous tasks. The time ratio for the circle distribution is much higher than for the other distributions because the circle distribution is a worst-case example for the algorithm, i. e., the interval filter almost never gives the conclusive answer for the INCIRCLE tests. The random distribution has the lowest increase since it generally has only a few close-to-degenerate cases, which can not be analysed by the interval filter, it is the situation where the straight implementation produces a wrong result.

The percentage of wrong edges as a function of number of points also tends to a constant (see Figure 8.18). One can note that for the worst-case example (the circle distribution) the percentage of wrong edges is much higher than for other distributions. For the circle distribution the straight algorithm simply fails to produce a reliable result, generating 30% of wrong edges. For certain values of perturbation this percentage reaches 50%. Hence, the computational expense of the exact algorithm pays off by correcting more edges in the Delaunay triangulation. The percentage of wrong edges for the random distribution does not exceed 1%.

Independent of the statistical analysis of the numerical examples, one gets a very surprising insight to triangulations as side effect. The examples have shown drastically how large the number of wrong vertices is if no error control is implemented. This number ranges from 1% in stable constellations up to 50% at close to degenerate constellations. One learns that *error control is unavoidable if one wants to obtain reliable triangulations.*

8.4 Exact and Robust Line Simplification

8.4.1 Introduction

Line simplification is a data reduction process that occurs for example in cartography when the scale of a map is decreased. Given a polygon $\mathcal{P}$ on a map a

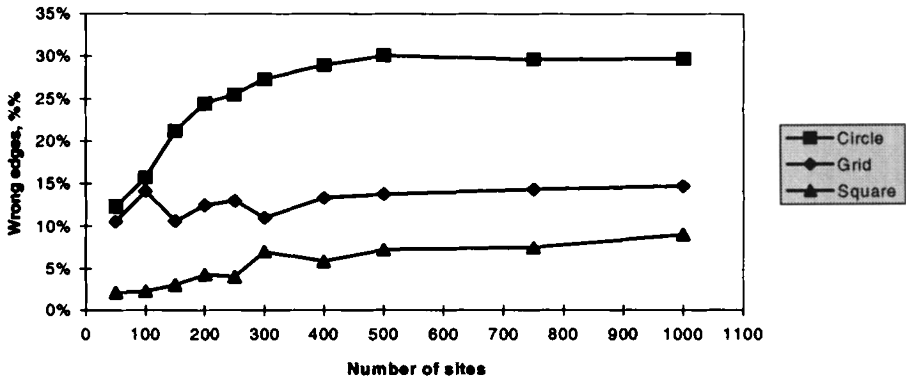


Figure 8.20: Percentage of wrong edges vs. number of points

line simplification algorithm generates a simplified polygon with fewer number of vertices which lies in some neighborhood of $\mathcal{P}$.

It is also used in image processing and pattern recognition as part of the vectorization process [268, 114] and for general approximation of planar curves [38, 203].

It is best to develop the concept for line simplification and to explain the need for it from the point of view of GIS (geographical information systems). Digital cartography and GIS are developments following the invention of the electronic computer. These developments have necessitated automating tasks that were previously manual such as map simplification and other transformations between the various representations used for geographical information (see [19, 117, 148, 150, 173, 175, 194, 272]). Initially it was thought that the implementation of these tasks on a computer would lead to representations of geographical data which would be reproduced without error on any device. This was found to be optimistic for example by Visvalingam-Whyatt [268] who discussed the influence of rounding errors on a version of the Ramer-Douglas-Peucker [34, 203] algorithm for (straight) line simplification (abbreviated the *R-D-P algorithm* in the sequel).

The reason for simplification is that details easily visible in a cartographic map at a given scale can often not be displayed effectively at a smaller scale due to cluttering. When a scale was decreased in manual cartography, line simplification was done by a cartographer who would simplify using a pen, an erasing tool or both. These tools were employed to bring out important features of a map eliminating unwanted detail. The implementation of this process on a computer resulted in the development of a number of algorithms dealing with various aspects of simplification. Each of the algorithms delivers different results. The choice of algorithm depends on the user's experience and preference as well as factors such as purpose, database issues versus map simplification, and even the geomorphology of the data being simplified. An

optimal approach might be to develop an artificial intelligence based algorithm using knowledge elicited from skilled cartographers.

A reasonable requirement could be that the results should be reproducible for a given algorithm, It was shown by [268] that even this reasonable goal was not easily achieved due to numerical errors. We quote [273] discussing GIS data bases:

> A major problem with spatial data is the control of error propagation under spatial operations. Further research is needed on finite precision geometry and multiple resolution techniques.

This problem might taken on greater importance in web environments when GIS is distributed and issues of reproducibility and correctness of algorithms are becoming essential.

As a hypothetical example from another area suppose that in a military campaign the troops rely on a cartographic database that is distributed to the troops in the field over a communications network. Unfortunately, the information can only be displayed on a low resolution device so that line simplification has to be performed. If the simplification parameters for the local displays are all the same then the GIS information should be identical. It is easy to envision situations where conflicting information would lead to disastrous results.

Almost all of the problems which are due to rounding errors can be avoided in the R-D-P algorithm if variable precision arithmetic is used, cf.[273]. However, the computational costs are then high since the length of the intermediate results can increase exponentially. In this paper, based on [223], we propose an approach to the rounding error problem in GIS algorithms that achieves the same result as variable multiple precision arithmetic at a much lower computational cost.

The R-D-P algorithm is not uniquely defined since there are a number of versions that differ in some of the details of the algorithm and since some of the implementation details are left open in all of the published versions. In this paper we therefore first define a prototype algorithm that specifies the details of implementation in order to be precise and able to guarantee reproducibility of the exactly computed result.

The R-D-P algorithm was chosen since it is widely used and hence a suitable vehicle for a discussion of stability, robustness and reproducibility in the field of GIS. The choice of the R-D-P algorithm does, however, not imply that we maintain that it is the most suitable algorithm for line simplification now and in the future. If there is another algorithm that can be shown to be more suitable from the GIS point of view then we would maintain that it also should be subject to the same demands for stability, robustness and reproducibility as we have established for the R-D-P algorithm.

To deliver the exactly computed result we proceed as follows: The computations of the prototype algorithm are first executed using interval arithmetic.

In this manner rounding errors are kept under control and the computations produce intervals as results which are guaranteed to include the results of the underlying exact, that is, error free computations, cf. [165], [132], [212]. (Any other software or hardware which is able to produce guaranteed bounds for the numerical errors can also be used as an alternative to interval arithmetic.) During the interval arithmetic computation each interval result will be checked to see if it is small enough to make a guaranteed decision with respect to the flow of the computation. If a decision is not possible then the part computation which leads to such an interval is reformulated so that ESSA can be applied. The decisions which are obtained with ESSA are then completely correct. The logical flow as well as the mathematical underpinnings of the R-D-P algorithm are not changed by this reformulation.

This particular implementation of the R-D-P algorithm is therefore robust, rounding error free and produces reproducible results, provided the input data is represented exactly.

A recent paper by Franklin [65] calls for such algorithms. He states that *a fragile implementation may process small test cases, while failing on large, realistic examples, perhaps because a neglected round-off error deep inside the code caused a topological inconsistency that, much later was fatal.* The robust version of the R-D-P algorithm presented in this paper avoids these inconsistencies.

The remainder of this section consists of two parts. That is, Subsec. 8.4.2 describes a prototype R-D-P algorithm and Subsec. 8.4.3 shows how to reformulate certain parts of the prototype R-D-P algorithm so that it can be handled with ESSA. An interval filter is incorporated which accelerates the computation.

8.4.2 The Ramer-Douglas-Peucker Line Simplification Algorithm for Polygons

Cartographic line simplification is a surprisingly delicate task which has turned out to be rather difficult to quantify. A review of some of the algorithms can be found in [160] or in [161]. Most of the algorithms compute simple geometric primitives for the line (which is generally a polygon, also called a broken line), which are then used to decide whether the line should be simplified or not. For our purposes we selected the R-D-P algorithm as the vehicle for our discussions since it is probably the most popular algorithm due to its simplicity and early publication.

The original R-D-P algorithm was described rather informally [34] where the first vertex of a polygon was called an *anchor point* and where a second vertex of the polygon selected according to some rule was called a *floater*. Several interpretations of the algorithm are possible depending on how the floater is selected. For example, consider simplifying the polygon in Figure 8.21.

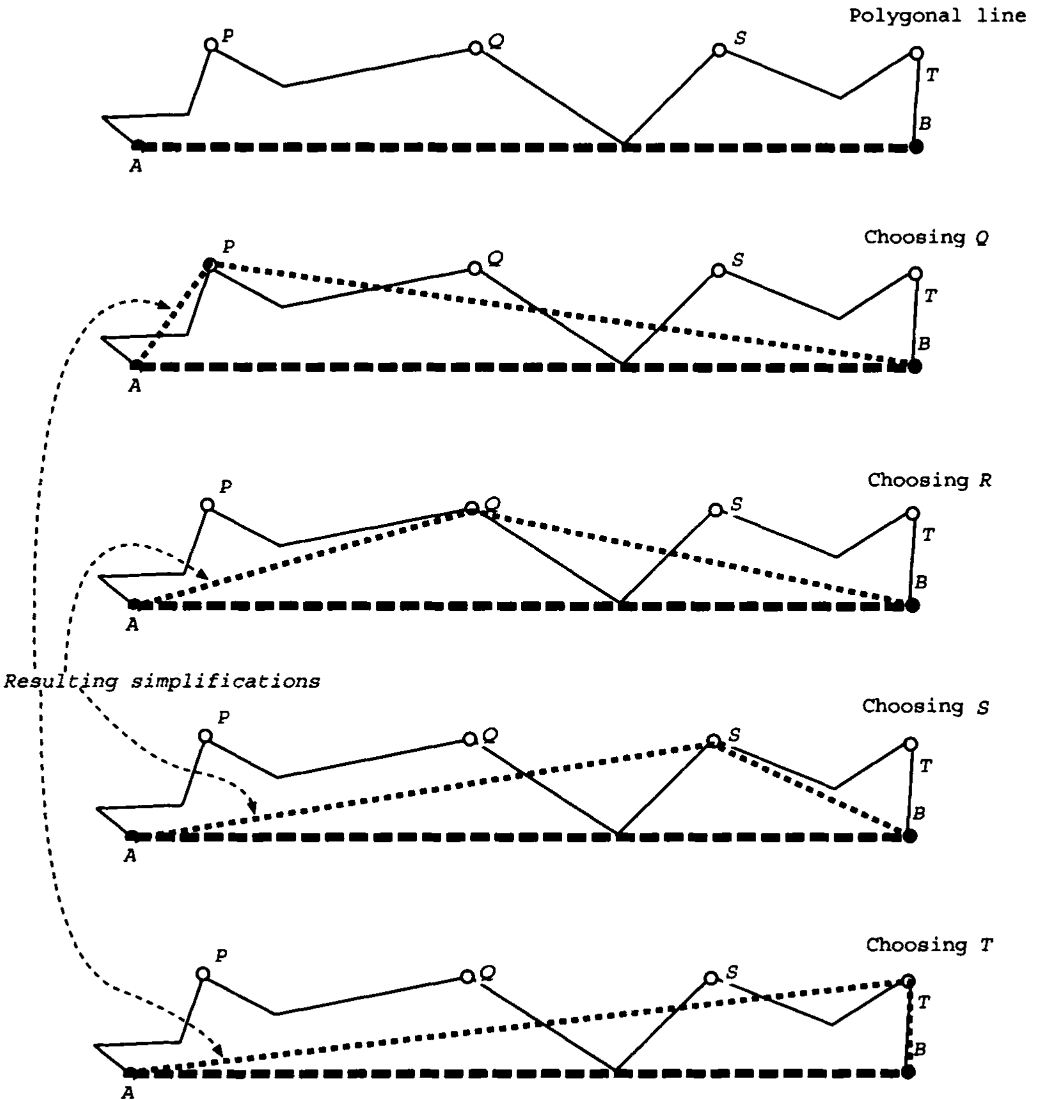


Figure 8.21: A non-unique R-D-P simplification

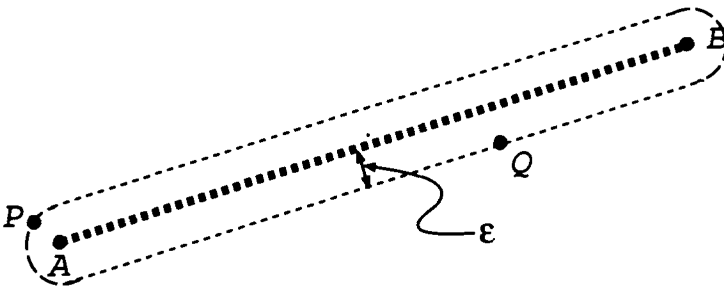


Figure 8.22: An ϵ-strip

Points P, Q, S and T are equidistant from the line connecting A and B. The anchor point is chosen as A and the remaining points have been chosen in turn to be the floater. The procedure procedure described in [34] is then followed obtaining a simplified line from A to the floater (using a tolerance slightly less than the distance from the line to the floater). The polygon from the floater to B is also simplified in the figure. There are four choices of floater and hence there are four different simplified polygons as shown in 8.21.

Because of the ambiguity in selecting the floater and because of the problems that can occur in numerical computations a precise definition of a version of the R-D-P was needed.

We focused on a precise recursive prototype version since one of the aims of the paper is to present an algorithm that delivers reproducible results. This is clearly not possible unless the computational steps are completely determined in the algorithm.

Let A and B be two points in the plane. Then a straight line segment between A and B is denoted by $\overline{AB}$. If $X_1, X_2, \ldots, X_n$ is a sequence of points and $\mathcal{L}_i = \overline{X_i X_{i+1}}$, then $\mathcal{P} = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_{n-1}$ is a polygon.

If P is a point and $\mathcal{L}$ a straight line segment then by $d(P, \mathcal{L})$ we mean the Hausdorff distance between P and $\mathcal{L}$ defined as min $\|P - X\|$ w.r.t. all $X \in \mathcal{L}$ where the norm is the Euclidean standard norm. Furthermore the region formed by points X satisfying $d(X, \mathcal{L}) \leq \epsilon$ is called an ***epsilon-strip around*** $\mathcal{L}$ or, better, an ***epsilon-neighborhood of*** $\mathcal{L}$, cf. Figure 8.22 following Perkal [193].

The typical computations in cartography are executed so that questions such as "is point P closer to line $\mathcal{L}$ than point Q in Figure 8.22"? can be answered. The subsequent flow of the algorithm depends strongly on the answer to this question. If P and Q are approximately equidistant to $\mathcal{L}$ then small perturbation in P, Q or $\mathcal{L}$ can have a large effect on the final simplification. Examples of this are found in [268] where the implementation of the R-D-P algorithm for different computer systems is explored.

Let the polygon $\mathcal{P}$ to be simplified be given by the vertices A, $X_1, \ldots, X_n$, B, in this order. Then A and B are the two endpoints of $\mathcal{P}$, and $\mathcal{P} = \mathcal{L}_1 \cup \mathcal{L}_2 \cup$

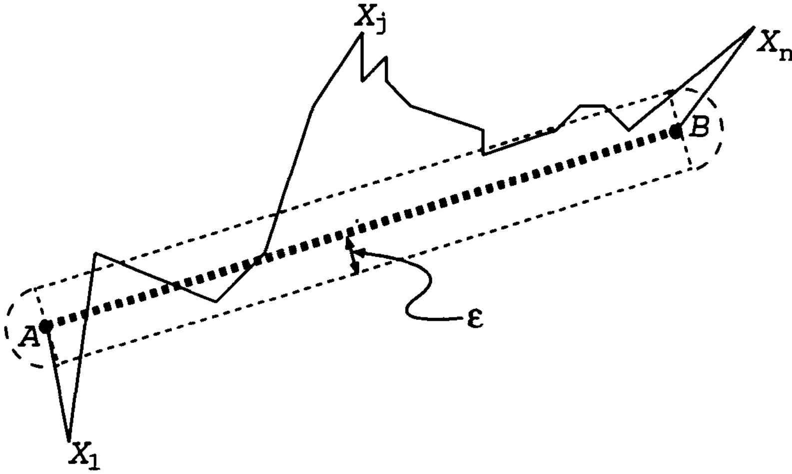


Figure 8.23: Line to be simplified

$\ldots \cup \mathcal{L}_{n+1}$ where $\mathcal{L}_1 = \overline{AX_1}$, $\mathcal{L}_i = \overline{X_{i-1}X_i}, i = 2, \ldots n$ and $\mathcal{L}_{n+1} = \overline{X_nB}$, The line segment $\mathcal{B} = \overline{AB}$ is known as the ***base line*** or the ***anchor-float line*** of the polygon [268]. If all points $X_1, \ldots, X_n$ lie in an ϵ-strip around $\mathcal{B}$, we say the ***epsilon criterion*** w.r.t. $\mathcal{B}$ is satisfied.

The essence of the R-D-P algorithm is that the polygon $\mathcal{P} = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_{n+1}$ is deleted and replaced by $\mathcal{B}$ as simplification if the points $X_1, \ldots, X_n$ satisfy the epsilon criterion w.r.t. $\mathcal{B}$. If the epsilon criterion is not satisfied, the polygon is divided into two smaller polygons, $\mathcal{P}_l = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_j$ and $\mathcal{P}_r = \mathcal{L}_{j+1} \cup \mathcal{L}_{j+2} \cup \ldots \cup \mathcal{L}_{n+1}$. The dividing point X_j is selected as a point with maximum distance to the anchor line, $\mathcal{B}$, see Figure 8.23. If there is more than one with maximum distance to the line then the point with minimum index is chosen. The algorithm is now recursively executed for each of the two smaller polygons.

The prototype version of the R-D-P algorithm we present differs slightly from the version given in [102] or in [161] when choosing the unique division point. The execution of the computational steps of the algorithm are, however, completely different in our case, since the computation of distances is avoided (real valued) and replaced by comparisons of distances (Boolean valued), which can be done rounding error free, cf. Subsec. 8.4.3. The cycles dealt with in [268] are not considered since we wanted a precise, but not overly complex prototype algorithm which is transparent and easy to describe and discuss.

ALGORITHM 24 *(Ramer-Douglas-Peucker)*

Input: *Points* $A, X_1, \ldots, X_n, B$ *and tolerance* ϵ.

Step 1. *Set $\mathcal{B} = \overline{AB}$ and $\mathcal{P} = \overline{AX_1} \cup \overline{X_1X_2} \cup \cdots \cup \overline{X_{n-1}X_nX_nB}$.*

Step 2. *If $n = 0$ then goto 8.*

Step 3. *Compute j s.t. $d(X_j, \mathcal{B}) \geq d(X_i, \mathcal{B}), i = 1, \ldots, n$ and $d(X_j, \mathcal{B}) > d(X_i, \mathcal{B}), i = 1, \ldots, j-1$.*

Step 4. *If $d(X_j, \mathcal{B}) \leq \epsilon$ then replace the polygon $\mathcal{P}$ with $\mathcal{B}$ and go to 8.*

Step 5. *Let $\mathcal{B}_L = \overline{AX_j}$, $\mathcal{P}_L = \overline{AX_1} \cup \overline{X_1X_2} \cup \cdots \overline{X_{j-1}X_j}$, $\mathcal{B}_R = \overline{X_jB}$, $\mathcal{P}_R = \overline{X_jX_{j+1}} \cup \cdots \cup \overline{X_{n-1}X_n} \cup \overline{X_nB}$.*

Step 6. *Call Ramer-Douglas-Peucker with input A, $X_1, \ldots, X_{j-1}$, X_j and epsilon resulting in straight line segments $\mathcal{L}_1, \ldots, \mathcal{L}_k$ as output such that $\mathcal{P}_L = \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_k$ is a polygon with endpoints A and X_j.*

Step 7. *Call Ramer-Douglas-Peucker with input X_j, $X_{j+1}, \ldots, X_n$, B and epsilon resulting in straight line segments $\mathcal{L}_{k+1}, \ldots, \mathcal{L}_m$ as output such that $\mathcal{P}_R = \mathcal{L}_{k+1} \cup \cdots \cup \mathcal{L}_m$ is a polygon with endpoints X_j and B.*

Step 9. Output: *Lines $\mathcal{L}_1, \ldots \mathcal{L}_m$ such that $\mathcal{P}' = \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_m$ is a polygon with endpoints A and B.*

Note that the output in Step 8 is not necessarily the result of the complete simplification process since it might just be an intermediate result from Step 2 or from Step 4 in some branch of the recursive process. When this branch is terminated then this intermediate result is dealt with by another branch.

If one reflects for a moment on the flow of the algorithm it is clear that the only floating point operations in the algorithm which are subject to rounding errors are the evaluations of distances. An unstable calculation of these distances will therefore cause an instability in the determination of the index j in Step 3 and also for the Boolean statement of whether X_j lies in the ϵ-strip around $\mathcal{B}$ or not. It can in each singular case easily be decided by interval arithmetic whether the influence of the rounding errors is significant because of the guaranteed error bounds which are part of the interval arithmetic computation:

For example, let the interval results of the computations when entering Step 3 be D_i for $d(X_i, \mathcal{B})$ for $i = 1, \ldots, n$ and $D_j > D_i$ for $i \neq j$. Clearly, for this case X_j is the point with maximum distance to $\mathcal{B}$, and no further processing is necessary.

If we have the situation that $D_1 = [10, 10.001]$, $D_9 = [10.001, 10.002]$ and $D_i < D_9$ for all $i \neq 1, 9$, it cannot be decided whether $j = 1$ or $j = 9$ at the current state of the computation: Certainly, it is most likely that $j = 9$, but $j = 1$ would be the result in the rather rare case that $d(X_1, \mathcal{B})= d(X_9, \mathcal{B})= 10.001$. In order to satisfy the claim for robustness and reproducibility, it has to be decided which of 1 or 9 is the real dividing index. In this case, a rounding error free comparison of the two distances $d(X_1, \mathcal{B})$ and $d(X_9, \mathcal{B})$ is necessary.

Note, however, it is only one comparison, independent of the number of vertices of the polygon.

We sum up and proceed as follows in order to determine the point X_j of Step 3 of Alg. 24 where the polygon is divided:

1. Determine an interval, say D^*, which has the largest lower boundary among the inclusions D_i, $i = 1, \ldots, n$. (This is the same work as determining the largest number out of a set of numbers.)

2. All the intervals where the upper boundary is (strictly) smaller than the lower boundary of D^* are removed since they cannot contain X_j.

3. For the remaining intervals it is suspected that they contain $d(X_j, \mathcal{B})$. Generally, only one interval is remaining which then must contain $d(X_j, \mathcal{B})$, and X_j and the splitting index j has been found. If several intervals remain, simple interval arithmetic is no longer helpful, and the distances which correspond to these intervals are compared rounding error free with ESSA as will be explained below.

The execution of the ϵ-criterion in Step 4 of Alg. 24 is very simple. Note that Step 4 always has to start with D_j even if ESSA had to be used to determine j:

1. If the upper boundary of D_j is smaller or equal to ϵ, the ϵ-criterion is guaranteed to be satisfied.

2. If the lower boundary of D_j is (strictly) larger than ϵ, the ϵ-criterion is guaranteed not to be satisfied.

3. Hence, if $\epsilon \in D_j$, no decision is possible so far, and the comparison has to be made rounding error free with ESSA.

How this can be incorporated into the algorithm is explained in the next section.

8.4.3 Exact Computations of the Comparisons

The comparisons which are listed at the end of the previous section are now translated into algorithmic steps that can be processed using ESSA. We start with some general facts about the orientation and the area of a triangle and relate these to the computational steps required for the algorithm.

Let points $A = (a_x, a_y)$, $B = (b_x, b_y)$, $C = (c_x, c_y)$ and the line segment $\mathcal{L} = \overline{AB}$ be given and consider the determinant

$$s = \begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{vmatrix}. \tag{8.9}$$

This determinant can be interpreted in two ways. The first interpretation is that $s/2$ provides the oriented area of the triangle with vertices A, B, and C (in this order). The second interpretation is that the sign of s provides the orientation of the three points. More precisely, if $s > 0$ the points A, B and C are ordered counter clockwise, if $s = 0$ then they lie on a line and if $s < 0$ they are ordered clockwise. This is called the *orientation test.* Equivalently, the sign of s determines on which side of $\mathcal{L}$ the point C lies. In this case, the line segment $\mathcal{L}$ is assumed to be directed from A to B. More precisely, if $s > 0$ the point C lies to the left of the line $\mathcal{L}$, if $s < 0$ the point C lies to the right of the line $\mathcal{L}$. If, $s = 0$ the point C lies exactly on the straight line (but not necessarily on the line segment) going through A and B. These tests are well-known in computational geometry, see for example Preparata-Shamos [198]. One recognizes that we have the same formalism as at the left-turn test in Ch. 4.

We will need the orientation test to find out on which side of the base line the polygon points X_i lie which will be needed for determining the sign of s. The area of the triangle given by its corners A, B, and X_i will be needed since it is proportional to the normal distance from the segment $\overline{AB}$ to the point X_i.

As one can see, the computation of the determinant is subject to rounding errors even if the entries of the determinant are exactly representable in the machine. Hence intermediate results and also the final simplification can be falsified. We again emphasize that the purpose of ESSA is to avoid the rounding errors.

In order to use ESSA the determinant (8.9) has to be expanded as a sum. We obtain

$$s = a_x b_y + b_x c_y + c_x a_y - a_x c_y - b_x a_y - c_x b_y. \tag{8.10}$$

If the quantities $a_x, a_y, b_x, b_y, c_x, c_y$ are all stored as single precision machine numbers the products $a_x b_y$ etc. require double precision for their exact representation and the sign of (8.10) can be computed using a double precision version of ESSA. Alternately, each double precision product can be split into two single precision parts so that (8.10) is a sum of 12 single precision numbers and only a single precision version of ESSA in needed.

We now need an executable formula for the distance of a point P to the anchor line $\mathcal{B} = \overline{AB}$, which can be found for example, in [161]. The formula provides a decision as to where P lies with respect to the line $\mathcal{B}$ with reference to Figure 8.24:

- $d(P, \mathcal{B}) = \|P - A\|$, if P is *outside* A.
- $d(P, \mathcal{B}) = \|P - B\|$, if P is *outside* B.
- $d(P, \mathcal{B}) = |s|/\|A - B\|$, P is *inside both* A and B.

The third case of this formula is also valid in the boundary case where A or B is nearest and $P - A$ resp. $P - B$ is orthogonal to $\mathcal{B}$.

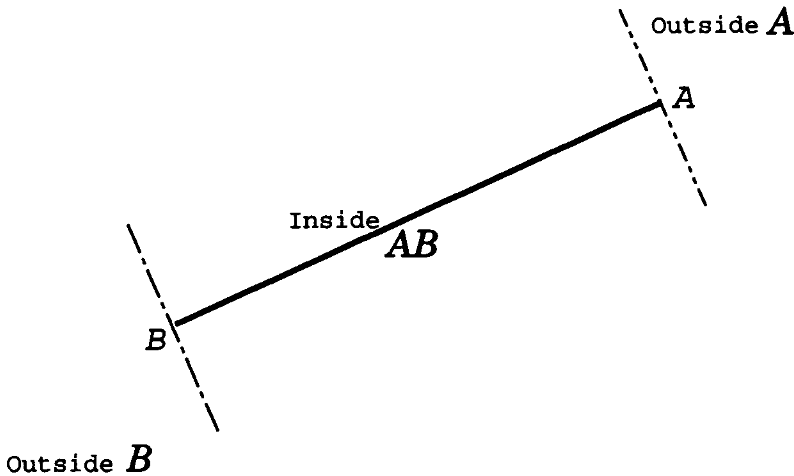


Figure 8.24: The three regions

The cosine of the angle between the vectors $B - A$ and $P - A$ as well as between the vectors $A - B$ and $P - B$ gives a criterion to decide which of the three cases for the distance formula applies, cf. [161].

P is outside B iff
$$\frac{(P-A)\cdot(B-A)}{\|P-A\|\|B-A\|} \geq 0,$$
where "$\cdot$" is the standard inner product in the plane, R^2, and the norm is the Euclidean norm. Hence, P is outside A iff
$$(p_x - a_x)(b_x - a_x) + (p_y - a_y)(b_y - a_y) < 0. \tag{8.11}$$
It is easy to see that (8.11) can be represented exactly as a sum of eight numbers in double mantissa length, so that (8.11) can be handled with ESSA.

Similarly, P is outside B iff
$$(p_x - b_x)(a_x - b_x) + (p_y - b_y)(a_y - b_y) < 0. \tag{8.12}$$
Finally, P is inside A and B iff neither (8.11) nor (8.12) holds.

With these formulas, it is possible to decide precisely which of the 3 cases of the formula for the distance $d(P, B)$ is applicable. Since the distance formula is only used for comparisons and its value is never computed, we have to know the sign in order to make the distance formula accessible to ESSA. That is, if sign $s > 0$ then $|s| = s$, and if sign $s < 0$ then $|s| = -s$. The sign can first be determined with interval arithmetic, and if not successful, with ESSA using (8.10).

If the comparison of distances $d(P, B)$ and $d(Q, B)$ of 2 points P and Q is required one proceeds as in the following steps:

1. One has to determine, which cases of the distance formulas are valid for P and Q.

2. If the comparison is involved in the cases $\|P - A\|$, $\|P - B\|$, $\|Q - A\|$ or $\|Q - B\|$ just raise the norm to the power 2 (then the square roots vanish), multiply through, and apply ESSA.

3. If the comparison is involved in the cases $|s_P|/\|A - B\|$ and $|s_Q|/\|A - B\|$ where s_P and s_Q are the determinants for P and Q, resp., cancel the denominators, determine the signs of s_P and s_Q, replace $|s_P|$ by $\pm s_P$ and $|s_Q|$ by $\pm s_Q$ in order to avoid absolute values, and apply ESSA for the comparison.

4. If one of the first two cases of the distance formula is connected with the third case, for example, $\|P - A\|$ with $|s_Q|/\|A - B\|$ the comparison is executed via $\|P - A\|^2\|A - B\|^2$ and $|s_Q|^2 = s_Q^2$. These expressions are multiplied through and ESSA is applicable. The occurring summands are of quadruple mantissa length, but can be split up into summands of double or single mantissa length as explained before.

The comparisons in Step 4 of Alg. 24 are processed analogously.

The ideas presented above have been implemented in Java and the program can be tested at

`http://www.ucalgary.ca/~rokne`

Bibliography

[1] Akl, S. G., Toussaint, G. T.: *A fast convex hull algorithm.* Info. Proc. Letters 7, pp. 219-222 (1978).

[2] Alander, J.: *On interval arithmetic range approximation methods of polynomials and rational functions.* Computers and Graphics 9, pp. 365-372 (1985).

[3] Albrycht, J., Wisniewski, H. (eds.): *Proc. Polish Symp. Interval and Fuzzy Math.* Inst. Math., Tech. Univ. Poznan (1985).

[4] Alefeld, G.: *Intervallrechnung Über den komplexen Zahlen und einige Anwendungen.* Dissertation, Universität Karlsruhe (1968).

[5] Alefeld, G., Herzberger, J.: *Einführung in die Intervallrechnung.* Bibliographisches Institut, Mannheim (1974).

[6] Alefeld, G., Herzberger, J.: *Introduction to Interval Computations.* Academic Press, New York (1983).

[7] Alefeld, G., Rokne, J.: *On the interval evaluation of rational functions in interval arithmetic.* SIAM Journal on Numerical Analysis 18, pp. 862-870 (1981).

[8] Allgower, E. L., Georg, K.: *Numerical Continuation Methods. An Introduction.* Springer-Verlag, Berlin (1990).

[9] Andrew, A. M.: *Another efficient algorithm for convex hulls in two dimensions.* Information Processing Letters 9, pp. 216-219 (1979).

[10] Arvo, J.: *Transforming axis-aligned bounding boxes.* In: [69], pp. 548-550 (1990).

[11] Arvo, J.: *A simple method for box-sphere intersection testing.* In: [69], pp. 335-339 (1990).

[12] Aurenhammer, F.: *Power diagrams: properties, algorithms, and applications.* SIAM Journal of Computing 16, pp. 78-96 (1987).

[13] Avnaim, F., Boissonanat, J-D., Devillers, O., Preparata, F.P., Yvinec, M.: *Evaluation of a new method to compute signs of determinants.* Proc. of the Eleventh Symposium on Computational Geometry. ACM Press, pp. C16-C17 (1995).

[14] Bao, P., Rokne, J.: *Low complexity k-dimensional Taylor forms.* Applied Mathematics and Computation 27, pp. 265-280 (1988).

[15] Bartels, R., Beatty, J., Barsky, B.: *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling.* Morgan Kaufmann (1987).

[16] Bauch, H., Jahn, K.-U., Oelschlägel, D., Süsse, H., Wiebigke, V.: *Intervallmathematik - Theorie und Anwendungen.* Math.-Nat. Bibl., Bd. 72, BSB B.G. Teubner Verlagsges. Leipzig (1987).

[17] Baumann, E.: *Optimal centered forms.* Freiburger Intervall-Berichte 87/3, Institut für Angewandte Mathematik, Universität Freiburg, pp. 5-21 (1987).

[18] Bézier, P.: *The first years of CAD/CAM and the UNISURF CAD system.* In: [195], pp. 13-26 (1993).

[19] Blakemore, M.: *Generalization and error in spatial data bases.* Cartographica 21, pp. 131-139 (1987).

[20] Böhm, W., Farin, G., Kahmann, J.: *A survey of curve and surface methods in CAGD.* Computer Aided Geometric Design 1, pp. 1-60 (1984).

[21] Brönnimann, H., Burnickel, C. and Pion, S.: *Interval arithmetic yields efficient dynamic filters for computational geometry.* Proc. of the fourteenth annual symposium on computational geometry. ACM Press, pp. 165-174 (1998).

[22] Bühler, K.: *Fast and reliable plotting of implicit curves.* Proc. of the Workshop of Uncertainty in Geometric Computations, The University of Sheffield, 5-6 July 2001. Kluwer, forthcoming.

[23] Burnickel, C., Funke, S., Seel, M.: *Exact geometric predicates using cascaded computation.* Proc. of the fourteenth annual symposium on computational geometry. ACM Press, pp. 175-183 (1998).

[24] Chan, T. M.: *Output-sensitive results on convex hulls, extreme points and related problems.* Proc. of the eleventh annual symposium on computational geometry. ACM Press, pp. 10-19 (1995).

[25] Chandler, R.: *Bezier Curves.* 1990 Mathematical Sciences Calendar, pp. Jan.-March, Rome Press Inc. (1989).

[26] Chazelle, B., Dobbin, D. P.: *Intersection of convex objects in two and three dimensions*. Journal of the Association for Computing Machinery 34, pp. 1-27 (1987).

[27] Conte, S. D., de Boor C.: *Elementary Numerical Analysis, An Algorithmic Approach*. Third Ed., McGraw-Hill, New York (1980).

[28] Csendes, T.: *Convergence properties of interval global optimization algorithms with a new class of interval selection criteria*. J. Global Optim. 19, pp. 307-327 (2001).

[29] Csendes, T., Ratz, D.: *Subdivision direction selection in interval methods for global optimization*. SIAM J. Numer. Anal. 34, pp. 307-327 (2001).

[30] de Berg, M., van Krefeld, M., Overmars, M., Schwarzkopf, O.: *Computational Geometry. Algorithms and Applications*. Springer-Verlag, New York (1997).

[31] De Lorenzi, M.: *The XYZ GeoServer for geometric computation*. In: IGIS '94: Geographic Information Systems, LNCS 884, Springer-Verlag, Berlin, pp. 202-213 (1994).

[32] Dobkin, D., Silver, D.: *Recipes for geometry and numerical analysis, Part 1.: An empirical study*. Proc. of the Fourth ACM Symp. on Computational geometry, pp. 93-105 (1988).

[33] Dobkin, D., Silver, D.: *Applied computational geometry: towards robust solutions of basic problem*. Journal of Computer and System Science 40, pp. 70-87 (1990).

[34] Douglas, D. H., Peucker, T. K.: *Algorithm for the reduction of the number of points required to represent a digitized line or its caricature*. The Canadian Cartographer 10, pp. 112-122 (1973).

[35] Douglas, D.: *It makes me so CROSS*. In: D.J. Peuquet and D.F. Marble (eds.): Introductory readings in Geographic Information Systems. Taylor and Francis Ltd., London, pp. 303-307 (1990).

[36] Drabek, K.: *Plane curves and constructions*. In: K. Rektorys (ed.): Survey of Applicable Mathematics, The M.I.T. Press, Mass., pp. 150-204 (1969).

[37] Duff, T.: *Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry*. Computer Graphics 26, pp. 131-138 (1992).

[38] Dunham, J. G.: *Optimum uniform piecewise linear approximation of planar curves*. IEEE Tran. Pattern Anal. and Machine Intelligence PAMI-8, pp. 67-75 (1986).

[39] Dwyer, R.: *A fast divide and conquer algorithm for constructing Delaunay triangulations.* Algorithmica 2, pp. 137-151 (1987).

[40] Dwyer, R.: *Higher-dimensional Voronoi diagrams in linear expected time.* Discrete and Computational Geometry 6, pp. 343-367 (1991).

[41] Edelsbrunner, H., Mücke, E.P.: *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms.* Proc. of the Fourth Annual Symposium on Computational Geometry. ACM Press, pp. 118-133 (1988).

[42] Edelsbrunner, H.: *Geometry and Topology for Mesh Generation.* Cambridge University Press, Cambridge (2001).

[43] Ely, J. S.:*The VPI Software package for variable precision interval arithmetic.* Interval Computations 1993, pp. 135-153 (1993).

[44] Enger, W.: *Interval ray tracing - a divide and conquer strategy for realistic computer graphics.* The Visual Computer 9, pp. 91 - 104 (1992).

[45] Espelid, T. O.: *On floating point summation.* SIAM Review 37, pp. 603-607 (1995).

[46] Farin, G.: *Curves and Surfaces for CAGD.* (Fourth edition) Academic Press, Boston, (1997).

[47] Farouki, R. T., Rajan, V. T.: *On the numerical condition of polynomials in Bernstein form.* Computer Aided Geometric Design 4, pp. 191-216 (1987).

[48] Farouki, R.T., Rajan, V.T.: *Algorithms for polynomials in Bernstein form.* Computer Aided Geometric Design 5, pp. 1-26 (1988).

[49] Farouki, R. T.: *Numerical stability in geometric algorithms and representations.* In: D. C. Hanscomb (ed.): Proceedings Mathematics of Surfaces III, Oxford University Press, New York (1989).

[50] Farouki, R. T.: *Computing with barycentric polynomials.* The Math. Intelligencer 13, pp. 61-69 (1991).

[51] Farouki, R. T.:' *Pythagorean-hodograph curves in practical use.* In: R. Barnhill (ed.): Geometry Processing for Design and Manufacture, SIAM, Philadelphia, pp. 3-33 (1992).

[52] Faux, I. D., Pratt, M. J.: *Computational Geometry for Design and Manufacture.* John Wiley, New York (1979).

[53] de Figueiredo, H., Stolfi, J.: *Adaptive enumeration of implicit surfaces with affine arithmetic.* Computer Graphics Forum 15, pp. 287-296 (1996).

[54] Foley, J. D., Van Dam, A., Feiner, S. K., Hughes, J. F.: *Computer Graphics, Principles and Practice* (Second edition). Addison-Wesley, Reading, pp. 431-445 (1990).

[55] Fomia, N.: *A robust and fast convergent interval analysis method for the calculation of internally controlled switching instants.* IEEE Trans. on Circuits and Systems 43, pp. 191-199 (1996).

[56] Forrest, A. R.: *Geometric computing environments: Computational geometry meets software engineering.* In: R. A. Earnshaw (ed.): Theoretical Foundations of Computer Graphics and CAD. NATO Advanced Study Institute Series Vol 40, Springer Verlag, New York, pp. 185-197 (1987).

[57] Forsythe, G.: *What is a satisfactory quadratic equation solver.* In: B. Dejon and P. Henrici (eds.): Constructive Aspects of the Fundamental Theorem of Algebra. Wiley-Interscience, New York, pp. 53-61 (1969).

[58] Forsythe, G.: *Pitfalls in computation, or why a math book isn't enough.* American Math. Monthly 77, pp. 931-956 (1970).

[59] Fortune, S.: *Stable maintenance of point set triangulations in two dimensions.* 30th Symposium on Foundations of Computer Science, pp. 494-499 (1989).

[60] Fortune, S.: *Numerical stability of geometric algorithms.* In: P. J. Laurent, A. Le Méhauté and L. L. Schumaker (eds.): Curves and Surfaces. Academic Press, Boston, pp. 189-192 (1991).

[61] Fortune, S.: *Numerical stability of algorithms for 2D Delaunay triangulations.* International Journal of Computational Geometry and Applications 6, pp. 193-213 (1995).

[62] Fortune, S., Wyk, C.: *Efficient exact arithmetic for computational geometry.* Proceedings of the Ninth Annual Symposium on Computational Geometry. ACM Press, pp. 163-172 (1993).

[63] Fortune, S., Wyk, C.: *Static analysis yields efficient exact integer arithmetic for computational geometry.* ACM Transactions on Graphics 15, pp. 223-248 (1996).

[64] Franklin, W. R.: *Efficient polyhedron intersection and union.* Proc. Graphics Interface 82, Canadian Information Processing Soc., Toronto, pp. 73-80 (1982).

[65] Franklin, W. R.: *Applications of analytical cartography.* Cartography and Geographic Information Science 27, pp. 225-237 (2000).

[66] Gavrilova, M., Rokne, J.: *An efficient algorithm for construction of the power diagram from the Voronoi diagram in the plane.* International Journal of Computer Mathematics 11, pp. 3-4 (1996).

[67] Gavrilova, M., Rokne, J.: *Reliable line segment intersection testing.* Computer-Aided Design 32, pp. 737-745 (2000).

[68] Gavrilova, M., Rokne, J.: *Computing line intersections.* Int. J. of Image and Graphics 1, pp. 217-230 (2001).

[69] Glassner, A.S.: *Graphics Gems.* Academic Press, New York (1990).

[70] Gagnet, M., Hervé, J.-C., Pudet, T., Van Thong, J.M.: *Incremental computation of planar maps.* Computer Graphics (SIGGRAPH' 89) 23, pp. 345-354, (1989)

[71] Gallier, J.: *Curves and Surfaces in Geometric Modeling, Theory and Algorithms.* Morgan Kaufmann, San Francisco (1999).

[72] Garloff, J.: *Interval mathematics. A bibliography.* Freiburger Intervall-Berichte 6, pp. 1-122 (1985).

[73] Garloff, J.: *Bibliography on interval mathematics. Continuation.* Freiburger Intervall-Berichte 2, pp. 1-50 (1987).

[74] Goldstein, A. J., Richman, P. L.: *A midpoint phenomenon.* Journal of the A.C.M. 20, pp. 301-304 (1973).

[75] Goldman, R. N.: *Identities for the univariate and bivariate Bernstein basis polynomials.* In: [192], pp. 149-162 (1995).

[76] Golin, M. and Sedgewick, R.: *Analysis of a simple yet efficient convex hull algorithm.* Proc. of the fourth annual symposium on computational geometry. ACM Press, pp. 153-163 (1988).

[77] Graham, R.: *An efficient algorithm for determining the convex hull of a finite planar set. Information Processing Letters*, 1, pp. 132-133 (1972).

[78] Grandine, T. A.: *Computing zeroes of spline functions.* Computer Aided Geometric Design 6, pp. 129-136 (1989).

[79] Gravesen, J.: *The length of Bézier curves.* In: [192], pp. 199-205 (1995).

[80] Greene, D., Yao, F.: *Finite-resolution computational geometry.* Proceedings of the 27th Annual Symposium on Foundations of Computer Science. ACM Press, pp. 143 - 152 (1986).

[81] Greene, N.: *Detecting intersection of a rectangular solid and a convex polyhedron.* In: [101], pp. 74-82 (1994).

[82] Gries D. and Stojmenović I.: *A note on Graham's convex hull algorithm.* Inform. Process. Lett. 25, pp. 323-327 (1987).

[83] Griewank, A., Corliss, G. (eds.): *Automatic Differentiation of Algorithms.* SIAM, Philadelphia (1991).

[84] Guddat, J., Vazquez, F. G., Jongen, H. T.: *Singularities, Pathfollowing, and Jumps.* Wiley, Chichester (1990).

[85] Guibas, L., Stolfi, J.: *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams.* ACM Transactions on Graphics 4, pp. 74-123 (1985).

[86] Guibas, L., Knuth, D., Sharir, M.: *Randomized incremental construction of Delaunay and Voronoi diagram.* Algorithmica 7, pp. 381-413 (1992).

[87] Hansen, E. R. (ed.): *Topics in Interval Analysis.* Oxford University Press, Oxford (1969).

[88] Hansen, E. R.: *The centered form.* In: [87], pp. 102-105 (1969).

[89] Hansen, E. R.: *A globally convergent interval analytic method for computing and bounding real roots.* BIT 18, pp. 153-163 (1978).

[90] Hansen, E. R.: *Global optimization using interval analysis - the multidimensional case.* Numer. Math. 34, pp. 247-270 (1980).

[91] Hansen, E. R.: *Global Optimization using Interval Arithmetic.* Marcel Dekker, New York (1992).

[92] Hansen, E. R.: *Bounding the solution of interval linear equations.* SIAM J. Numer. Anal. 29, pp. 1493-1503 (1992).

[93] Hansen, E. R.: *The hull of preconditioned interval linear equations.* Rel. Computing 6, pp. 95-103 (2000).

[94] Hansen, E. R., Greenberg, R. I.: *An interval Newton method.* Applied Math. and Comp. 12, pp. 89-98 (1983).

[95] Hansen, E. R., Sengupta, S.: *Global constrained optimization using interval analysis.* In: [184], pp. 25-47 (1980).

[96] Hansen, E. R., Sengupta, S.: *Bounding solutions of systems of equations using interval analysis.* BIT 21, pp. 203-211 (1981).

[97] Hansen, E. R., Smith, R. R.: *Interval arithmetic in matrix computations, part II.* SIAM Journal on Numerical Analysis 4, pp. 1-9 (1967).

[98] Hanson, A. J.: *Geometry for N-dimensional graphics.* In: [101], pp. 149-170 (1994).

[99] Haines, E.: *Essential ray tracing algorithms.* In: A. Glassner (ed.): An Introduction to Ray Tracing, Academic Press, New York, pp. 33-78 (1989).

[100] Hanrahan, P.: *A survey of ray-surface intersection algorithms.* In: A. Glassner (ed.): An Introduction to Ray Tracing, Academic Press, New York, pp. 79-120 (1989).

[101] Heckbert, P. S.: *Graphic Gems IV.* Academic Press, Boston (1994).

[102] Hershberger, J., Snoeyink, J.: *Speeding up the Douglas-Peucker line simplification algorithm.* In: Proceedings of the 5th International Symposium on Spatial Data Handling, International Geographical Union, Columbia, pp. 210-218 (1992).

[103] Herve, C., Morain, F., Salesin, D., Serpette, B. P., Vullemin, J., Zimmermann, P.: *BigNum: A portable and efficient package for arbitrary-precision arithmetic.* INRIA Report 1016, Rocquencourt, (1989).

[104] Higham, N. J.: *The accuracy of floating point summation.* SIAM J. Sci. Computation 14, pp. 783-799 (1993).

[105] Hill, K. J.: *Matrix-based ellipse geometry.* In: [192], pp. 73-77 (1995).

[106] Hobby, J., Baird, H.: *Degraded character image restoration.* Proceedings of the Fifth Annual Symposium on Document Analysis and Image Retrieval, pp. 233-245 (1996).

[107] Hoffmann, C.: *The problem of accuracy and robustness in geometric computation.* IEEE Computer 22, pp. 31-42 (1989).

[108] Hoffmann, C.: *Geometric and Solid Modeling. An Introduction.* Morgan Kaufmann Inc., San Mateo (1989).

[109] Hoschek, J., Lasser, D.: *Fundamentals of Computer Aided Geometric Design.* A.K.Peters, Wellesley, Mass. (1993).

[110] Hu, C.-Y., Maekawa, T., Patrikalakis, N. M., Ye, X.: *Robust interval algorithm for surface intersections.* Computer-Aided Design 29, pp. 617-627 (1997).

[111] Hyvönen, E., DePascale, S.: *InC++ library family for interval computations.* In: *Reliable Computing,* Supplement (Extended Abstracts of APIC'95), pp. 85-90 (1995).

[112] Institute of Electrical and Electronic Engineers: *IEEE standard for binary floating-point arithmetic.* IEEE Standard 754-1985, IEEE, New York (1985).

[113] Institute of Electrical and Electronic Engineers: *IEEE standard for radix-independent floating-point arithmetic.* IEEE Standard 854-1987, IEEE, New York (1987).

[114] Janssen, D. D. T., Vossepoel, A. M.: *Adaptive vectorization of line drawing images.* Computer Vision and Image Understanding 56, pp. 38-56 (1995).

[115] Jaromczyk, J. W., Wasilkowski, G. W.: *Numerical stability of a convex hull algorithm for simple polygon.* Algorithmica 10, pp. 457-472 (1993).

[116] Jaromczyk, J. W., Wasilkowski, G. W.: *Computing convex hull in a floating point arithmetic.* Computational Geometry Theory and Applications 4, pp. 283-293 (1994).

[117] Jones, C. B., Abraham, I. M.: *Line generalization in a global cartographic database.* Cartographica 24, pp. 32-45 (1987).

[118] Jones, A. K.: A review of *Christoph M. Hoffman: Geometric and Solid Modeling: An Introduction.* SIAM Reviews 34, pp. 327-329(1992).

[119] Jou, E.: *Determine whether two line segments intersect.* In: Modeling in Computer Graphics. Proceedings of the 10th Working Conference, Springer-Verlag, Berlin, Germany, pp. 265-274 (1991).

[120] Jünger, K., Reinelt, G., Zepf, D.: *Computing correct Delaunay triangulations.* Computing 47, pp. 43-49 (1991).

[121] Kalmykov, S. A., Shokin, Y. I., Yuldashev, Z. K.: *Interval Analysis Methods.* NAUKA, Novosibirsk, (1986) (in Russian).

[122] Kao, T., Knott, G.: *An efficient and numerically correct algorithm for the 2D convex hull problem.* BIT 30, pp. 311-331 (1990).

[123] Karasick, M., Lieber, D., Nackman, L.: *Efficient Delaunay triangulation using rational arithmetic.* ACM Transactions of Graphics 10, No. 1, pp. 71-91 (1991).

[124] Katajainen, J., Koppinen, M.: *Constructing Delaunay triangulations by merging buckets in quadtree order.* Annales Sociatatis Mathematicae Polonae, Series IV, Fundamenta Informaticae 11, 275- 288 (1988).

[125] Kearfott, R. B.: *Preconditioners for the interval Gauss-Seidel method.* SIAM J. Numer. Anal. 27, pp. 804-822 (1990).

[126] Kearfott, R. B.: *Rigorous computation of surface patch intersection curves.* Preprint (1993)

[127] Kearfott, R. B.: *Rigorous Global Search: Continuous Problems.* Kluwer, Dordrecht (1996).

[128] Kearfott, R. B.: *INTERVAL-ARITHMETIC: A Fortran 90 module for an interval data type.* ACM Trans. Math. Software 22, pp. 385-392 (1996).

[129] Kearfott, R. B. *On existence and uniqueness verification of non-smooth functions.* Reliable Computing 8, pp. 267-282 (2002).

[130] Kearfott, R. B. and Xing, Z.: *An interval step control for continuation methods.* SIAM J. Numerical Analysis 31, pp. 892-914 (1994).

[131] , Kearfott, R. B., Dawande, M., Du, K. S., Hu, C. Y.: *Algorithm 737 INTLIB.* ACM Trans. Math. Software 20, pp. 447-459 (1994).

[132] Klatte, R., Kulisch, U., Wiethoff, A., Lawo, Rauch, M.: *C-XSC.* Springer-Verlag, Berlin (1993).

[133] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ulrich, C.: *PASCAL-XSC.* Springer-Verlag, Berlin (1991).

[134] Knott, G., Jou, D.: *A program to determine whether two line segments intersect.* Technical Report CAR-TR-306, Dept. of Computer Science, University of Maryland, MD, USA (1987).

[135] Knüppel, O.: *PROFIL/BIAS - A fast interval library.* Computing 53, pp. 277-288 (1994).

[136] Knuth, D. E. *Metafont: The Program.* Addison Wesley Publ. Co., Reading, Mass. (1986).

[137] Knuth, D. E.: *Axioms and Hulls.* LNCS 606, Springer Verlag, New York (1992).

[138] Krawczyk, R.: *Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken.* Computing 4, pp. 187-201 (1969).

[139] Krawczyk, R.: *Zur äußeren und inneren Einschließung des Wertebereichs einer Funktion.* Freiburger Intervallberichte 80/7, pp. 1-19 (1980).

[140] Krawczyk, R.: *Intervallsteigerungen für rationale Funktionen und zugeordnete zentrische Formen.* Freiburger Intervallberichte 83/2, pp. 1-30 (1983).

[141] Krawczyk, R.: *A class of interval Newton operators.* Computing 37, pp. 179-183 (1986).

[142] Krawczyk, R., Nickel, K.: *Die zentrische Form in der Intervallarithmetik, ihre quadratische Konvergenz und ihre Inklusionsisotonie.* Computing 28, pp. 117-132 (1982).

[143] Kriegel, H.-P., Schmidt, T., Seidl, T.: *3D similarity search by shape approximation.* In: M. Scholl and A. Voisar (eds.): Advances in Spatial Databases, LNCS 1262, Springer-Verlag, Berlin, pp. 11-28 (1997).

[144] Kulisch, U.: *Numerik mit automatischer Ergebnisverification.* GAMM-Mitteilungen, pp. 39-58 (1994).

[145] Kulisch, U., Miranker, W. L. (eds.): *A new Approach to Scientific Computation.* Academic Press, Orlando (1983).

[146] Levin, J.: *A parametric algorithm for drawing pictures of solid objects composed of quadric surfaces.* CACM 19, pp. 555-563 (1976).

[147] Li, Z., Milenkovic, V.: *Constructing strongly convex hulls using exact or rounded arithmetic.* Proceedings of the Sixth Annual Symposium on Computational Geometry. ACM Press, pp. 235-243 (1990).

[148] Li, Z.: *Some observations on the issue of line generalization.* The Cartographic Journal 30, pp. 68-71 (1993).

[149] Lin, Q., Rokne, J.: *A family of centered forms for a polynomial.* BIT 32, pp. 167-176 (1992).

[150] Lodwick, W. A., Monson, W., Svoboda, L.: *Attribute error and sensitivity analysis of map operations in geographical informations systems: suitability analysis.* Int. J. Geographical Information Systems 4, pp. 413-428 (1990).

[151] Lorentz, G. C.: *Bernstein Polynomials.* University of Toronto Press, Toronto (1953).

[152] McLain, D. H.: *Two dimensional interpolation from random data.* Comp. Journal 19, pp. 178-181 (1976)

[153] Mäntylä, M.: *Boolean operations of 2-manifolds through vertex neighborhood classification.* ACM Trans. on Graphics 5, pp. 1-29 (1986).

[154] Martin, R., Shou, H., Voiculescu, I., Bowyer, A., Wang, G.: *Comparison of function range methods for solving algebraic equations in two variables.* Manuscript (2001).

[155] Martin, R. et al.: *Comparison of interval methods for plotting algebraic curves.* Computer Aided Geometric Design 19, pp. 553-587 (2002).

[156] Maruyama, K.: *A procedure to determine intersections between polyhedral objects.* International Journal of Computer and Information Sciences 1, pp. 255-266 (1972).

[157] Maus, A.: *Delaunay triangulation and the convex hull of n points in expected linear time.* BIT 24, pp. 151-163 (1984).

[158] Milenkovic, V.: *Verifiable implementations of geometric algorithms using finite-precision arithmetic.* Artificial Intelligence 37, pp. 377-401(1988).

[159] Milenkovic, V.: *Double-precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic.* Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science. ACM Press, pp. 500-505 (1989).

[160] McMaster, R. B.: *The integration of simplification and smoothing algorithms in line generalization.* Cartographica 26, pp. 101-121 (1989).

[161] Meek, D. S., Walton, D. J.: *Several methods for representing discrete data by line segments.* Cartographica 28, pp. 13-20 (1991).

[162] Miller, R. D.: *Quick and simple Bézier curve drawing.* In: [192], pp. 206-209 (1995).

[163] Markov, S. M.: *On the algebra of intervals and convex bodies.* Journal of Universal Computer Science 4, pp. 34-47 (1998).

[164] Moore, R. E.: *Interval arithmetic and automatic error analysis in digital computation.* Ph. D. Thesis, Stanford University (1962).

[165] Moore, R. E.: *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ. (1966)

[166] Moore, R. E.: *Mathematical Elements of Scientific Computing.* Holt Rinehart Winston, New York (1975).

[167] Moore, R. E.: *A test for existence of solutions to non-linear systems.* SIAM Journal on Numerical Analysis 14, pp. 611-615 (1977).

[168] Moore, R. E.: *A computational test for convergence of iterative methods for non-linear systems.* SIAM Journal on Numerical Analysis 15, pp. 1194-1196 (1978).

[169] Moore, R. E.: *Methods and Applications of Interval Analysis.* SIAM, Philadelphia (1979).

[170] Moore, R. E.: *Interval methods for nonlinear systems.* Computing Suppl. 2, pp. 113-120 (1980).

[171] Moore, R. E. (ed.): *Reliability in Computing: The Role of Interval Methods in Scientific Computation.* Academic Press, New York (1988).

[172] Moore, R. E., Jones, S. T.: *Safe starting regions for iterative methods.* SIAM Journal on Numerical Analysis 14, pp. 1051-1065 (1977).

[173] Mower, J. E.: *Developing parallel procedures for line simplification.* Int. J. Geographical Information Systems 10, pp. 699-712 (1996).

[174] Mudur, S. P., Koparkar. P. A.: *Interval methods for processing geometric objects.* IEEE Computer Graphics and Applications 4, pp. 7-17 (1984).

[175] Muller, J.-C.: *Fractal and automated line generalization.* The Cartographic Journal 24, pp. 27-34 (1987).

[176] Nataraj, P. S. V, Sheela, S. M.: *A new subdivison strategy for range computations.* Rel. Computing 8, pp. 83-92 (2002).

[177] Nazarenko, T. I., Marchenko, L. V.: *Introduction to Interval Methods of Computational Mathematics.* Izd-vo Irkutskogo Universiteta, Irkutsk, (1982) (in Russian).

[178] Neumaier, A.: *The enclosure of solutions of parameter-dependent systems of equations.* In [171], pp. 269-286 (1988).

[179] Neumaier, A.: *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge (1990).

[180] Neumaier, A.: *A simple derivation of the Hansen- Bliek -Rohn- Ning - Kearfott enclosure for linear interval equations.* Rel. Computing 5, pp. 131-136 (1999) .

[181] Nickel, K.: *On the Newton method in interval analysis.* Mathematics Research Center Report 1136, University of Wisconsin (1971).

[182] Nickel, K. (ed.): *Interval Mathematics.* Proceedings of the International Symposium, Karlsruhe 1975, Springer-Verlag, Wienna (1975).

[183] Nickel, K.: *Die Uberschätzung des Wertebereichs einer Funktion in der Intervallrechnung mit Anwendungen auf lineare Gleichungssysteme.* Computing 18, pp. 15-36 (1977).

[184] Nickel, K. (ed.): *Interval Mathematics 1980.* Proceedings of the International Symposium, Freiburg 1980, Academic Press, New York (1980).

[185] Nickel, K. (ed.): *Interval Mathematics 1985.* Proceedings of the International Symposium, Freiburg 1985, Springer Verlag, Wienna (1986).

[186] Ning, S., Kearfott, R. B.: *A comparison of some methods for solving lineaer interval equations.* SIAM J. Numer. Anal. 34, pp. 1289-1305 (1997).

[187] Oishi, Y., Sugihara, K.: *Topology-oriented divide-and-conquer algorithm for Voronoi diagrams.* Graphics Models and Image Processing 57, pp. 303-314 (1995).

[188] Okabe, A., Boots, B., Sugihara, K.: *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, Chichester, West Sussex, England, pp. 205-208 (1992).

[189] Oliveira, F.A.: *Bounding solutions of nonlinear equations with interval analysis*. Proc. 13th World Congr. Comp. Appl. Math., Dublin, pp. 246-247 (1991).

[190] O'Rourke, J.: *Computational Geometry in C*. Cambridge University Press, Cambridge (1994).

[191] Ortega, J. M., Rheinboldt, W. C.: *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, Orlando (1970).

[192] Paeth, A. W. (ed.): *Graphics Gems V*. Academic Press, Boston, 1995.

[193] Perkal, J.: *An attempt at objective line generalization*. (Trans. R. Jackowski). Michigan Inter-University Consortium of Mathematical Geographers, Discussion Paper #10, University of Michigan, Ann Arbor, MI (1966).

[194] Peucker, T. K.: *A theory of the cartographic line*. Auto-Carto II. Proceedings of the international symposium on computer-assisted cartography. U.S. Department of Commerce Bureau of the Census and the ACSM. pp. 508-518 (1975).

[195] Piegel, L. (ed.): *Fundamental Developments of Computer-Aided Geometric Modeling*. Academic Press, London (1993).

[196] Pion, S.: *De la géométrie algorithmique au calcul géométrique*. Thèse doctorat, Université de Nice Sophia-Antipolis (1999).

[197] Prautzsch, H., Boehm, W., Paluszny, M.: *Bezier and B-Spline Techniques*. Springer-Verlag, New York (2002).

[198] Preparata, F. P., Shamos, M. I.: *Computational Geometry* (Second printing). Springer-Verlag, New York (1988).

[199] Press, W. H., Flannery, B. P., Teukolsky, S.A, Wetterling, W.T.: *Numerical Recipes in C*. Cambridge U. P., Cambridge (1988).

[200] Priest, D. M.: *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. Ph.D. Thesis, University of California, Berkeley (1992).

[201] Rall, L. B.: *Automatic Differentiation: Techniques and Applications*. LNCS 120, Springer-Verlag, New York (1981).

[202] Rall, L. B.: *Mean value and Taylor forms in interval analysis*. SIAM Journal on Mathematical Analysis 14, pp. 223-238 (1983).

[203] Ramer, U.: *An iterative procedure for the polygonal approximation of plane curves*. Computer Graphics and Image Processing 1, pp. 244-256 (1972).

[204] Ramshaw, L.: *CSL notebook entry: The braiding of floating point lines.* Unpublished manuscript, Xerox PARC, Palo Alto, California (1982).

[205] Ratschan, S.: *Search heuristics for box decomposition methods.* J. Global Optim. 24, pp. 35 - 49 (2002).

[206] Ratschek, H.: *Die Subdistributivität der Intervallarithmetik.* Z. Angew. Math. Mech. 51, pp. 189-192 (1971).

[207] Ratschek, H.: *Teilbarkeitskriterien der Intervallarithmetik.* J. Reine Angew. Math. 252, pp. 128-138 (1972).

[208] Ratschek, H.: *Nichtnumerische Aspekte der Intervallarithmetik.* In: [182], pp. 48-74 (1975).

[209] Ratschek, H.: *Zentrische Formen.* Zeitschrift für Angewandte Mathematik und Mechanik 58, pp. T434-T436 (1978).

[210] Ratschek, H.: *Centered forms.* SIAM Journal on Numerical Analysis 17, pp. 656-662 (1980).

[211] Ratschek, H.: *Some recent aspects of interval algorithms for global optimization.* In: [171], pp. 325-339 (1988).

[212] Ratschek, H., Rokne, J.: *Computer Methods for the Range of Functions.* Ellis Horwood, Chichester (1984).

[213] Ratschek, H., Rokne, J.: *New Computer Methods for Global Optimization.* Ellis Horwood, Chichester (1988).

[214] Ratschek, H., Rokne, J.: *Nonuniform variable precision bisecting.* In: C. Brezinski and U. Kulisch: Computational and Applied Mathematics I, Elsevier, Amsterdam, pp. 419-428 (1992).

[215] Ratschek, H., Rokne, J.: *Test for intersection between plane and box.* Computer-Aided Design 25, pp. 249-250 (1993).

[216] Ratschek, H., Rokne, J.: *Box-sphere intersection tests.* Computer-Aided Design 26, pp. 579-584 (1994).

[217] Ratschek, H., Rokne, J.: *Formulas for the width of interval products.* Reliable Computing 1, pp. 9-14 (1995).

[218] Ratschek, H., Rokne, J.: *Determination of the exact sign of a sum.* In: G. Alefeld and J. Herzberger (eds.): Numerical Methods and Error Bounds, Akademie Verlag, Berlin, pp. 198-204 (1995).

[219] Ratschek, H., Rokne, J.: *The relationship between a rectangle and a triangle.* The Visual Computer 12, pp. 360-370 (1996).

[220] Ratschek, H., Rokne, J.: *Test for Intersection Between Box and Tetrahedron.* Intern. J. Computer Math. 65, pp. 191-204 (1997).

[221] Ratschek, H., Rokne, J.: *Exact computation of the sign of a finite sum.* Applied Mathematics and Computation, 99, pp. 99-127 (1999).

[222] Ratschek, H., Rokne, J.: *Exact and Optimal Convex Hulls in 2D.*. International Journal of Computational Geometry & Applications 10, pp. 109-129 (2000).

[223] Ratschek, H., Rokne, J., Lerigier, M.: *Robustness in GIS Algorithm Implementation with Application to Line Simplification.* Int. J. Geographical Inf. Systems 15, pp. 707-720 (2001).

[224] Ratschek, H., Rokne, J.: *A 2D Ellipse-Rectangle Intersection Test.* Journal of Mathematical Modelling and Algrithms. Forthcomming.

[225] Ratschek, H., Rokne, J.: *SCCI-hybrid methods for 2D-curve tracing.* Accepted by IJIG.

[226] Reliable Computing. *Formerly Interval Computations.* Institute of New Technologies in Education, St. Petersburg, Vol. 1 (1995).

[227] Rheinboldt, W. C.: *Numerical Analysis of Parametrized Nonlinear Equations.* John Wiley and Sons, New York (1986).

[228] Rippa, S.: *Minimal roughness property of the Delaunay triangulation.* Comp. Aided Geometric Design 7, pp. 489-497 (1986).

[229] Rodgers, D. F., Adams, J, A.: *Mathematical Elements for Computer Graphics.* McGraw Hill, New York (1990).

[230] Rohn, J.: *Systems of linear interval equations.* Lin. Alg. Appls. 126, pp. 39-78 (1989).

[231] Rohn, J.: *Solving systems of linear interval equations.* In: [171], pp. 171-182 (1988).

[232] Rokne, J.: *Optimal computation of the Bernstein algorithm for the bound of an interval polynomial.* Computing 28, pp. 239-246 (1982).

[233] Rokne, J.: *A low complexity rational centered form.* Computing 34, pp. 261-263 (1985).

[234] Rokne, J.: *Low complexity k-dimensional centered forms.* Computing 37, pp. 247-253 (1986).

[235] Rokne, J., Bao, P.: *Interval Taylor forms.* Computing 39, pp. 247-259 (1987).

[236] Rokne, J.: *Interval arithmetic.* In: D. Kirk (ed.): Graphics Gems III, Academic Press, San Diego, pp. 61-66 and pp. 454-457 (1992).

[237] Salesin, D. H.: *Epsilon geometry: building robust algorithms from imprecise computations.* Ph.D. Thesis, Stanford University (1991).

[238] Samet, H.: *Applications of Spatial Data Structures.* Addison Wesley, Reading (1990).

[239] Schramm, P.: *Intersection problems of parametric surfaces in CAGD.* Computing 53, pp. 355-364 (1994).

[240] Schwetlick, H.: *Numerische Lösung nichtlinearer Gleichungen.* Oldenburg, München-Wien (1979).

[241] Scott, N. R.: *Computer Number Systems and Arithmetic.* Prentice Hall, Englewood Cliffs, N.J. (1985).

[242] Sederberg, T. W., Buehler, D. B.: *Offsets of polynomial Bézier curves: Hermite approximation with error bounds.* In: T. Lyche and L. Schumaker (eds.): Mathematical Methods in Computer Aided geometric Design II, Academic Press, pp. 549-558 (1992).

[243] Sederberg, T. W., Farouki, R. T.: *Approximation by interval Bézier curves.* IEEE Computer Graphics and Applications 12, September, pp. 87-95 (1992).

[244] Seidel, R.: *Convex hull computations.* In: J. E. Goodman and J. O'Rourke (eds.): Handbook of Computational Geometry, CRC Press LLC, Boca Raton, pp. 361-375 (1997).

[245] Shary, P. S.: *An optimal solution of interval linear equations.* SIAM J. Numer. Anal. 32, pp. 610-630 (1995).

[246] Shary, P. S.: *A surprising approach in interval global optimization.* Reliable Computing 7, pp. 497-505 (2001).

[247] Shene, C.-K.: *Test for intersection between plane and a connected compact polyhedron.* Computer-Aided Design 26, pp. 585-588 (1994).

[248] Shokin, Yu. I.: *Interval Analysis.* NAUKA, Novosibirsk, (1981) (in Russian).

[249] Simcik, L., Linz, P.: *Boundary-based interval Newton's method.* Interval Computations 4, pp. 89-99 (1993).

[250] Skelboe, S.: *Computation of rational interval functions.* BIT 14, pp. 87-95 (1974).

[251] Smith, R. E. (ed.): *Numerical Grid Generation Techniques*. NASA Langley Research Center, (1980).

[252] Snyder, J. M., Barr, A. H.: *Ray tracing complex models containing surface tesselations*. Computer Graphics 21, pp. 119-126 (1987).

[253] Snyder, J. M.: *Interval analysis for computer graphics*. Computer Graphics 26, pp. 121-130 (1992).

[254] Snyder, J. M.: *Generative Modeling for Computer Graphics and CAD*. Academic Press, New York,(1992).

[255] Strip, D., Karasik, M.: *SIMD algorithms for intersecting three-dimensional polyhedra*. SIAM News 27(3), pp. 1, 10, 11, 13 (1994).

[256] Su, P., Drysdale, R.: *A comparison of sequential Delaunay triangulation algorithms*. In:Proceedings of the 11th Annual Symposium on Computational Geometry, pp. 61-70 (1995).

[257] Suffern, K. G.: *Quadtree algorithms for contouring functions of two variables*. The Computer Journal 33, pp. 402-407 (1990).

[258] Suffern, K. G., Fackerell, E. D.: *Interval methods in computer graphics*. Computers and Graphics 15, pp. 331-340 (1991).

[259] Sugihara, K.: *A simple method for avoiding numerical error and degeneracy in Voronoi diagram construction*. IEICE Trans. Fundamentals E75-A, pp. 468-477 (1992).

[260] Sugihara, K., Iri, M.: *A robust topology-oriented incremental algorithm for Voronoi diagrams,*. International Journal of Computational Geometry and Applications 4, pp. 179-228 (1994).

[261] Sun Microsystems: *Interval Arithmetic in High performance Technical Computing*. Version 1.0. Sun Microsystems,Santa Clara, California (2002).

[262] Sunaga, T.: *Theory of an interval algebra and its application to numerical analysis*. RAAG Memoirs 2, pp. 547-564 (1958).

[263] Sutcliffe, D. C.: *An algorithm for drawing the curve* $f(x,y) = 0$. The Computer Journal 19, pp. 246-249 (1976).

[264] Taligent: *http://hpsalo.cern.ch/TaligentDocs/TaligentOnline/ DocumentRoot/1.0/Docs/classes/TGEllipse.html* (1997).

[265] Taubin, G. L.: *Distance approximations for rasterizing implicit curves*. ACM Transactions on Graphics 13, pp. 3-42 (1994).

[266] Toth, D. L.: *On ray tracing parametric surfaces*. Computer Graphics 19, pp. 171-179 (1985).

[267] Tupper, J.: *Reliable two-dimensional graphing methods for mathematical formulae with two free variables*. Siggraph 2001, Los Angeles, CA., pp. 77-86 (2001)

[268] Visvalingam, M., Whyatt, J. D.: *Cartographic algorithms: Problems of implementation and evaluation and the impact of digitizing errors*. Computer Graphics Forum 10, pp. 225-235 (1991).

[269] Walter, W. V.: *FORTRAN-XSC: A portable Fortran 90 module library for accurate and reliable scientific computation*. Computing (Supplement) 9, pp. 265-286 (1993).

[270] Warmus, M.: *Calculus of approximations*. Bull. Acad. Polon. Sci. Cl. III, 4, pp. 253-259 (1956).

[271] Wilkinson, J. H.: *Rounding Errors in Algebraic Processes*. Her Majesty's Stationery Office, London(1963).

[272] Wood, C. H.: *Perceptual responses to line simplification in task-oriented map analysis experiment*. Cartographica 32, pp. 22-32 (1995).

[273] Worboys, M. F.: *GIS. A Computing Perspective*. Taylor & Francis, London (1995).

[274] Yap, C. K.: *A geometric consistency theorem for a symbolic perturbation theorem*. Proceedings of the Fourth Annual ACM Symposium on Computational Geometry, ACM Press, pp. 134-142 (1988).

[275] Yap, C. K., Dubé, T.: *The exact computation paradigm*. In: Z. Du and F.K. Hwang (eds.): Computing in Euclidean Geometry. second ed., World Scientific, Singapore, pp. 452-492 (1995).

[276] Yap, C. K.: *Robust geometric computations*. In: J. E. Goodman and J. O'Rourke (eds.): Handbook of Computational Geometry, CRC Press LLC, Boca Raton, pp. 653-668 (1997).

[277] Yap, C. K.: http://cs.nyu.edu/exct/inc/nonrobustness.html (2000).

[278] Young, R. C.: *The algebra of many-valued quantities*. Math. Ann. 104, pp. 260-290 (1931).

[279] Zangwill, W. I., Garcia, C. B.: *Pathways to solutions of fixed points and equilibria*. Prentice-Hall, Englewood Cliffs (1981).

[280] Zeid, I.: *CAD/CAM Theory and Practice*. McGraw-Hill, New York (1991).

Index

ϵ-strongly convex hull, 82

anchor point, 287
arbitrary precision arithmetic, 82

Bézier curve of degree n, 230
Bézier curves, 219, 230
backward error analysis, 81, 82
barycentric combination, 234, 235
Bernstein basis polynomials, 222
Bernstein form, 222
Bernstein form of degree k, 229
Bernstein polynomial of degree k, 222
Bernstein polynomials, 219
bisection recipes, 56
boundary based interval Newton variants, 68
boundary points, 237
box, 26
box and plane intersections, 123

centered forms, 47
co-circular, 269
continuation method, 197–199
control points, 230
convergence order, 32
conversion error, 14, 97
convex combination, 234, 235
convex hull, 244
coplanarity test in 3D, 98
counter clock wise orientation test, 117
crossing points, 209
curve contour, 186

degree elevation, 223, 232
degree reduction, 223
Delaunay triangulation, 265, 268
discarded points, 199
distillation principle, 83
distributive law, 17
divide-and-conquer, 270

empty circle condition, 268
epsilon geometry, 81
ESSA, 79, 83
exact algorithm, 280
exact computation, 244
exact hull, 83
excess-width, 32
exclusion test, 187
existence test, 75
extended arithmetic, 34

floater, 287
floating point arithmetic, 14
floating point numbers, 14
fundamental property of interval arithmetic, 31

Gauss-Seidel iteration, 70
generating points, 237
global optimization, 29
Graham scan, 260
Graham scan algorithm, 80, 244

implicit curve, 186
in-circle-test, 99
incircle test, 178
including convex hull, 246
inclusion function, 30, 187, 189

inclusion function for the gradient, 190
inclusion isotone, 55
inclusion isotonicity, 18, 27, 33
inclusion principle of interval arithmetic, 16
inclusion principle of machine interval arithmetic, 19
incremental construction, 270
incremental construction algorithms, 270
incremental search algorithms, 270
inner approximations, 39
inner inclusions of the range, 40
insphere test, 178
intersection of two line segments, 115
interval Bézier curve of degree n, 232
interval Bézier curves, 219
interval barycentric coordinates, 130
interval Bernstein polynomials, 219
interval Bernstein polynomials of degree k, 229
interval control points, 232
interval curves, 220
interval filter, 81
interval Gauss-Seidel step, 69
interval matrices, 27
interval matrix operations, 27
interval Newton algorithm, 65
interval Newton method, 65
interval Newton variant of Oliveira, 68
interval polynomials, 226
interval tools, 81
interval valued polynomials, 226
interval variable, 28
interval vector, 27
isothetic rectangle hull, 134

last in first out, 57
left-turn test, 45, 80, 97, 247
line segment intersection test, 116
linear interval equation, 62
Lipschitz constant, 32
Lipschitz matrices, 68

machine exact operations, 26
machine interval arithmetic, 19
machine numbers, 14
machine representable numbers, 14
meanvalue form, 48, 190
meanvalue form function, 48
midpoint, 14, 27
monotone continuous functions, 35
monotonicity, 39

natural interval extension, 31
next_floating_point_number, 21

order of 3 lines in plane, 99
oriented area, 43
outward rounding, 21

perturbation parameter, 119
pixels, 186
point intervals, 16
power cells, 269
power diagram, 269
power form, 221, 226
power function, 268
power triangulation, 269
pre-declared inclusion functions, 30
preconditioning step, 69
Prototype Graham scan, 245
pseudolocal control, 231

R-D-P algorithm, 285
range, 30
raster devices, 186
real degree, 222
rounding, 20
rounding error, 14
rounding error control, 19

scaled Bernstein form, 224
scaled Bernstein polynomial, 225
SCCI-hybrid method, 185, 188, 189, 212

sign of determinant, 81
simple arithmetic, 34
simple power evaluation, 34
single interval Gauss-Seidel step, 75
Skelboe's principle, 35
slope comparisons, 82
slope of line, 81
slopes, 50
smallest machine intervals, 19
standard exclusion test, 192
standard parametric form, 220
strictly monotonically decreasing, 193
strictly monotonically increasing, 193
subdistributive law, 17
subdividing plotting cell, 199
subdivision process, 191
symmetric, 18

Taylor form, 51
Taylor form function, 51
touching points, 209
truly convex hull, 81
truncation, 20

unbounded intervals, 18

validated, 117
variable precision arithmetic, 83
Voronoi diagram, 265, 267
Voronoi regions, 267
Voronoi vertices, 267

width, 14, 26