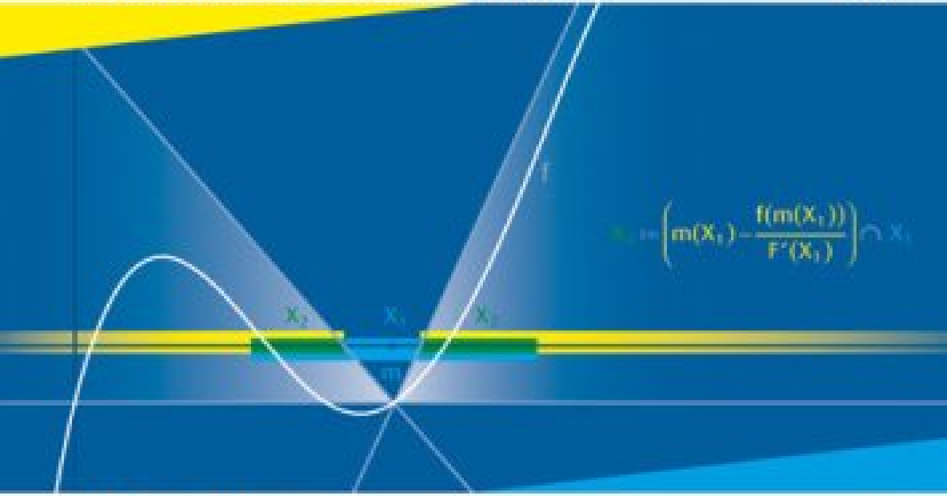
Ulrich Kulisch

Computer Arithmetic and Validity

Theory, Implementation, and Applications

Studies in Mathematics 33


f
$X_2 := \left(m(X_1) - \frac{f(m(X_1))}{F'(X_1)}\right) \cap X_1$
X_2
X_1
X_2
m

de Gruyter

de Gruyter Studies in Mathematics 33

Editors: Carsten Carstensen · Niels Jacob

de Gruyter Studies in Mathematics

1 Riemannian Geometry, 2nd rev. ed., *Wilhelm P. A. Klingenberg*
2 Semimartingales, *Michel Métivier*
3 Holomorphic Functions of Several Variables, *Ludger Kaup and Burchard Kaup*
4 Spaces of Measures, *Corneliu Constantinescu*
5 Knots, 2nd rev. and ext. ed., *Gerhard Burde and Heiner Zieschang*
6 Ergodic Theorems, *Ulrich Krengel*
7 Mathematical Theory of Statistics, *Helmut Strasser*
8 Transformation Groups, *Tammo tom Dieck*
9 Gibbs Measures and Phase Transitions, *Hans-Otto Georgii*
10 Analyticity in Infinite Dimensional Spaces, *Michel Hervé*
11 Elementary Geometry in Hyperbolic Space, *Werner Fenchel*
12 Transcendental Numbers, *Andrei B. Shidlovskii*
13 Ordinary Differential Equations, *Herbert Amann*
14 Dirichlet Forms and Analysis on Wiener Space, *Nicolas Bouleau and Francis Hirsch*
15 Nevanlinna Theory and Complex Differential Equations, *Ilpo Laine*
16 Rational Iteration, *Norbert Steinmetz*
17 Korovkin-type Approximation Theory and its Applications, *Francesco Altomare and Michele Campiti*
18 Quantum Invariants of Knots and 3-Manifolds, *Vladimir G. Turaev*
19 Dirichlet Forms and Symmetric Markov Processes, *Masatoshi Fukushima, Yoichi Oshima and Masayoshi Takeda*
20 Harmonic Analysis of Probability Measures on Hypergroups, *Walter R. Bloom and Herbert Heyer*
21 Potential Theory on Infinite-Dimensional Abelian Groups, *Alexander Bendikov*
22 Methods of Noncommutative Analysis, *Vladimir E. Nazaikinskii, Victor E. Shatalov and Boris Yu. Sternin*
23 Probability Theory, *Heinz Bauer*
24 Variational Methods for Potential Operator Equations, *Jan Chabrowski*
25 The Structure of Compact Groups, 2nd rev. and aug. ed., *Karl H. Hofmann and Sidney A. Morris*
26 Measure and Integration Theory, *Heinz Bauer*
27 Stochastic Finance, 2nd rev. and ext. ed., *Hans Föllmer and Alexander Schied*
28 Painlevé Differential Equations in the Complex Plane, *Valerii I. Gromak, Ilpo Laine and Shun Shimomura*
29 Discontinuous Groups of Isometries in the Hyperbolic Plane, *Werner Fenchel and Jakob Nielsen*
30 The Reidemeister Torsion of 3-Manifolds, *Liviu I. Nicolaescu*
31 Elliptic Curves, *Susanne Schmitt and Horst G. Zimmer*
32 Circle-valued Morse Theory, *Andrei V. Pajitnov*

Ulrich Kulisch

Computer Arithmetic and Validity

Theory, Implementation, and Applications

Walter de Gruyter
Berlin · New York

Author

Ulrich Kulisch
Institute for Applied and Numerical Mathematics
Universität Karlsruhe
Englerstr. 2
76128 Karlsruhe
Germany
E-mail: ulrich.kulisch@math.uka.de

Mathematics Subject Classification 2000: 65-04, 65Gxx

Keywords: Computer arithmetic, interval arithmetic, floating-point arithmetic, verified computing, interval Newton method

♾ Printed on acid-free paper which falls within the guidelines of the ANSI
to ensure permanence and durability.

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

ISBN 978-3-11-020318-9

Printed in Germany.
Cover design: Martin Zech, Bremen.
Typeset using the author's LaTeX files: Kay Dimler, Müncheberg.
Printing and binding: Hubert & Co. GmbH & Co. KG, Göttingen.

This book is dedicated to my wife Ursula

and to my family,
to Brigitte and Joachim,
Johanna and Benedikt,
to Angelika and Rolf,
Florian and Niclas,

to
all former and present colleagues of my institute,
and to my students.

Lasset uns am Alten,
so es gut ist, halten
und dann auf dem alten Grund
Neues schaffen Stund um Stund.

House inscription in the Black Forest.

Contents

Preface

This book deals with computer arithmetic in a more general sense than usual, and shows how the arithmetic and mathematical capability of the digital computer can be enhanced in a quite natural way. The work is motivated by the desire and the need to improve the accuracy of numerical computing and to control the quality of the computed result.

As a first step towards achieving this goal, the accuracy requirements for the elementary floating-point operations as defined by the IEEE arithmetic standard [644], for instance, are extended to the customary product spaces of computation: the complex numbers, the real and complex intervals, the real and complex vectors and matrices, and the real and complex interval vectors and interval matrices. *All computer approximations of arithmetic operations in these spaces should ideally deliver a result that differs from the correct result by at most one rounding.* For all these product spaces this accuracy requirement leads to operations which are distinctly different from those traditionally available on computers. This expanded set of arithmetic operations is taken as a definition of what is called *basic computer arithmetic*.

Central to this treatise is the concept of semimorphism. It provides a mapping principle between the mathematical product spaces and their digitally representable subsets. The properties of a semimorphism are designed to preserve as many of the ordinary mathematical laws as possible. All computer operations of basic computer arithmetic are defined by semimorphism.

The book has three antecedents:

(I) Kulisch, U. W., *Grundlagen des numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik*, Bibliographisches Institut, Mannheim, Wien, Zürich, 1976, 467 pp., ISBN 3-411-015617-9.

(II) Kulisch, U. W. and Miranker W. L., *Computer Arithmetic in Theory and Practice*, Academic Press, New York, 1981, 249 pp., ISBN 0-12-428650-X.

(III) Kulisch, U. W., *Advanced Arithmetic for the Digital Computer – Design of Arithmetic Units*, Springer-Verlag, Wien, New York, 2002, 139 pp., ISBN 3-211-83870-8.

The need to define all computer approximations of arithmetic operations by semimorphism goes back to the first of these books. By the time the second book had been written, early microprocessors were on the market. They were made with a few thousand transistors, and ran at 1 or 2 MHz. Arithmetic was provided by an 8-bit adder. Floating-point arithmetic could only be implemented in software. In 1985 the IEEE binary floating-point arithmetic standard was internationally adopted. Floating-point arithmetic became hardware supported on microprocessors, first by coprocessors and later directly within the CPU. Of course, all operations of basic computer arithmetic can be simulated using elementary floating-point arithmetic. This, however, is rather complicated and results in unnecessarily slow performance. A consequence of this is that for large problems the high quality operations of basic computer arithmetic are hardly ever applied. Higher precision arithmetic suffers from the same problem if it is simulated by software.

Dramatic advances in speed and memory size of computers have been made since 1985. Today a computer chip holds more than one billion transistors and runs at 3 GHz or more. Results of floating-point operations can be delivered in every cycle. Arithmetic speed has gone from megaflops to gigaflops, to teraflops, and to petaflops. This is not just a gain in speed. A qualitative difference goes with it. If the numbers a petaflops computer produces in one hour were to be printed (500 on one page, 1000 on one sheet, 1000 sheets 10 cm high) they would form a pile that reaches from the earth to the sun and back. With increasing speed, problems that are dealt with become larger and larger. Extending the word size cannot keep up with the tremendous increase in computer speed. Computing that is continually and greatly speeded up calls conventional computing into question. Even with quadruple and extended precision arithmetic the computer remains an experimental tool. The capability of a computer should not just be judged by the number of operations it can perform in a certain amount of time without asking whether the computed result is correct. It should also be asked how fast a computer can compute correctly to 3, 5, 10 or 15 decimal places. If the question were asked that way, it would very soon lead to better computers. Mathematical methods that give an answer to this question are available. Computers, however, are not built in a way that allows these methods to be used effectively.

Computer arithmetic must move strongly towards more reliability in computation. Instead of the computer being merely a fast calculating tool it must be developed into a scientific instrument of mathematics. Two simple steps in this direction would have great effect. They are both simple and practical:

I. fast hardware support for (extended[1]) interval arithmetic and

II. a fast and exact multiply and accumulate operation or, what is equivalent to it, an exact scalar product.

These two steps together with *basic computer arithmetic* comprise what is here called *advanced computer arithmetic*. Fast hardware circutries for I. and II. are developed in Chapters 7 and 8, respectively. This additional computational capability is gained at very modest hardware cost. Besides being more accurate the new computer operations greatly speed up computation. I. and II., of course, can be used to execute and speed up the operations of basic computer arithmetic. This would boost both the speed of a computation and the accuracy of its result.

[1] including division by an interval that includes zero

Advanced computer arithmetic opens the door to very many additional applications. All these applications are extremely fast. I. and II. in particular are basic ingredients of what is called *validated numerics* or *verified computing*.

This book has three parts. Part 1, of four chapters, deals with the theory of computer arithmetic, while Part 2, also of four chapters, treats the implementation of arithmetic on computers. Part 3, of one chapter, illustrates by a few sample applications how advanced computer arithmetic can be used to compute highly accurate and mathematically verified results.

Part 1: The implementation of semimorphic operations on computers requires the establishment of various isomorphisms between different definitions of arithmetic operations on the computer. These isomorphisms are to be established in the mathematical spaces in which the actual computer operations operate. This requires a careful study of the structure of these spaces. Their properties are defined as invariants with respect to semimorphisms. These concepts are developed in Part 1 of the book. Part 1 is organized along the lines of its second antecedent. However it differs in many details from the earlier one, details that spring from advances in computer technology, and many derivations and proofs have been reorganized and simplified.

Part 2: In Part 2 of the book, basic ideas for the implementation of advanced computer arithmetic are discussed under the assumption that the data are floating-point numbers. Algorithms and circuits are developed which realize the semimorphic operations in the various spaces mentioned above. The result is an arithmetic with many desirable properties, such as high speed, optimal accuracy, theoretical describability, closedness of the theory, and ease of use.

Chapters 5 and 6 consider the implementation of *elementary floating-point arithmetic* on the computer for a large class of roundings. A particular section of Chapter 5 comments on the IEEE floating-point arithmetic standard. The final section of Chapter 6 contains a brief discussion of all arithmetic operations defined in the product sets mentioned above as well as between these sets. The objective here is to summarize the definition of these operations and to point out that they all can be performed as soon as an *exact scalar product* is available in addition to the operations that have been discussed in Chapters 5 and 6.

Floating-point operations with directed roundings are basic ingredients of interval arithmetic. But with their isolated use in software interval arithmetic is too slow to be widely accepted in the scientific computing community. Chapter 7 shows, in particular, that with very simple circuitry interval arithmetic can be made practically as fast as elementary floating-point arithmetic. To enable high speed, the case selections for interval multiplication (9 cases) and division (14 cases including division by an interval that includes zero) are done in hardware where they can be chosen without any time penalty. The lower bound of the result is computed with rounding downwards and the upper bound with rounding upwards by parallel units simultaneously. The rounding mode needs to be an integral part of the arithmetic operation. Also the basic comparisons for intervals together with the corresponding lattice operations and the result selection in more complicated cases of multiplication and division are done in hardware. There they are executed by parallel units simultaneously. The circuits described in this chapter show that with modest additional hardware costs interval arithmetic can be made almost as fast as simple floating-point arithmetic. Such high speed cannot be obtained just by running many elementary floating-point arithmetic processors in parallel.

A basic requirement of basic computer arithmetic is that all computer approximations of arithmetic in the usual product spaces should deliver a result that differs from the correct result by at most one rounding. This requires scalar products of floating-point vectors to be computed with but a single rounding. The question of how a scalar product with a single rounding can be computed just using elementary floating-point arithmetic has been carefully studied in the literature. A good summary and what is probably the fastest solution is given in [456] and [531]. However, we do not follow this line here. No software simulation can compete with a simple and direct hardware solution.

The most natural way to accumulate numbers is fixed-point accumulation. It is simple, error free and fast. In Chapter 8 circuitry for *exact* computation of the scalar product of two floating-point vectors is developed for different kinds of computers. To make the new capability conveniently available to the user a new data format called *complete* is used together with a few simple arithmetic operations associated with each floating-point format. *Complete arithmetic* computes all scalar products of floating-point vectors exactly. The result of complete arithmetic is always exact; it is complete, not truncated. Not a single bit is lost. A variable of type complete is a fixed-point word wide enough to allow exact accumulation (continued summation) of floating-point numbers and of simple products of such numbers.

If register space for the complete format is available complete arithmetic is very very fast. The arithmetic needed to perform complete arithmetic is not much different from what is available in a conventional CPU. In the case of the IEEE double precision format a *complete register* consists of about $1/2$ K bytes. Straightforward pipelining leads to very fast and simple circuits. The process is at least as fast as any conventional way of accumulating the products including the so-called partial sum technique on existing vector processors which alters the sequence of the summands and causes errors beyond the usual floating-point errors.

Complete arithmetic opens a large field of new applications. An exact scalar product rounded into a floating-point number or a floating-point interval serves as building block for semimorphic operations in the product spaces mentioned above. Fast multiple precision floating-point and multiple precision interval arithmetic are other important applications. All these applications are very very fast. Complete arithmetic is an instrumental addition to floating-point arithmetic. In many instances it allows recovery of information that has been lost during a preceding pure floating-point computation.

Because of the many applications of the hardware support for interval arithmetic developed in Chapter 7, and of the exact scalar product developed in Chapter 8, these two modules of advanced computer arithmetic emerge as its central components.

Fast hardware support for all operations of advanced computer arithmetic *is a fundamental and overdue extension of elementary floating-point arithmetic. Arithmetic operations which can be performed correctly with very high speed and at low cost should never just be done approximately or simulated by slow software. The minor additional hardware cost allows their realization on every CPU. The arithmetic operations of* advanced computer arithmetic *transform the computer from a fast calculating tool into a mathematical instrument.*

Part 3: Mathematical analysis has provided algorithms that deliver highly accurate and completely verified results. Part 3 of the book goes over some examples. Such algorithms are not widely used in the scientific computing community because they are very slow when the underlying arithmetic has to be carried out on conventional processors.

The first section describes some basic properties of interval mathematics and shows how these can be used to compute the range of a function's values. Used with automatic differentiation, these techniques lead to powerful and rigorous methods for global optimization. The following section then deals with differentiation arithmetic or automatic differentiation. Values or enclosures of derivatives are computed directly from numbers or intervals, avoiding the use of a formal expression for the derivative of the function. Evaluation of a function for an interval X delivers a superset of the function's values over X. This overestimation tends to zero with the width of the interval X. Thus for small intervals interval evaluation of a function practically delivers the range of the functions's values. Many numerical methods proceed in small steps. So this property together with differentiation arithmetic to compute enclosures of derivatives is the key technique for validated numerical computation of integrals and for solution of differential equations, and for many other applications.

Newton's method is considered in two sections of Chapter 9. It attains its ultimate elegance and power in the *extended interval Newton method*, which is globally convergent and computes all zeros of a function in a given domain. The key to achieving these fascinating properties is division by an interval that includes zero.

The basic ideas needed for verified solution of systems of linear equations are developed in Section 9.5. Highly accurate bounds for a solution can be computed in a way that proves the existence and uniqueness of the solution within these bounds. Mathematical fixed-point theorems, interval arithmetic combined with defect correction or iterative refinement techniques using complete arithmetic are basic tools for achieving these results.

In Section 9.6 a method is developed that allows highly accurate and guaranteed evaluation of polynomials and of other arithmetic expressions.

Section 9.7 finally shows how fast multiple precision arithmetic and multiple precision interval arithmetic can be provided using complete arithmetic and other tools developed in the book.

Of course, the computer may often have to work harder to produce verified results, but the mathematical certainty makes it worthwhile. After all, the step from assembler to higher programming languages or the use of convenient operating systems also consumes a lot of computing power and nobody complains about it since it greatly enlarges the safety and reliability of the computation.

Computing is being continually and greatly speeded up. An avalanche of numbers is produced by a teraflops or petaflops computer (1 teraflops corresponds to 10^{12} floating-point operations per second). Fast computers are often used for safety critical applications. Severe, expensive, and tragic accidents can occur if the eigenfrequencies of a large electricity generator, for instance, are erroneously computed, or if a nuclear explosion is incorrectly simulated. Floating-point operations are inherently inexact. It is this inexactness at very high speed that calls conventional computing, just using naïve floating-point arithmetic, into question.

This book can, of course, be used as a textbook for lectures on the subject of computer arithmetic. If one is interested only in the more practical aspects of implementing arithmetic on computers, Part 2, with acceptance *a priori* of some results of Part 1, is also suitable as a basis for lectures. Part 3 can be used as an introduction to verified computing.

The second previous book was jointly written with Willard L. Miranker. On this occasion Miranker was very busy with other studies and could not take part, so this new book has been compiled solely by the other author and he takes full responsibility for its text. However, there are contributions and formulations here which go back to Miranker without being explicitly marked as such. I deeply thank Willard for his collaboration on the earlier book as well as on other topics, and for a long friendship. Contact with him was always very inspiring for me and for my Institute.

I would like to thank all former collaborators at my Institute. Many of them have contributed to the contents of this book, have realized advanced computer arithmetic in software on different platforms and in hardware in different technologies, have embedded advanced computer arithmetic into programming languages and implemented corresponding compilers, developed problem solving routines for standard problems of numerical analysis, or applied the new arithmetic to critical problems in the sciences. Among these colleagues are: Christian Ullrich, Edgar Kaucher, Rudi Klatte, Gerd Bohlender, Dalcidio M. Claudio, Kurt Grüner, Jürgen Wolff von Gudenberg, Reinhard Kirchner, Michael Neaga, Siegfried M. Rump, Harald Böhm, Thomas Teufel, Klaus Braune, Walter Krämer, Frithjof Blomquist, Michael Metzger, Günter Schumacher, Rainer Kelch, Wolfram Klein, Wolfgang V. Walter, Hans-Christoph Fischer, Rudolf Lohner, Andreas Knöfel, Lutz Schmidt, Christian Lawo, Alexander Davidenkoff, Dietmar Ratz, Rolf Hammer, Dimitri Shiriaev, Manfred Schlett, Matthias Hocks, Peter Schramm, Ulrike Storck, Christian Baumhof, Andreas Wiethoff, Peter Januschke, Chin Yun Chen, Axel Facius, Stefan Dietrich, and Norbert Bierlox. For their contributions I refer to the bibliograhy.

I owe particular thanks to Axel Facius, to Gerd Bohlender, and to Klaus Braune. Axel Facius keyed in and laid out the entire manuscript in LATEX and he did most of the drawings. Drawings were also done by Gerd Bohlender. Klaus Braune helped to prepare the final version of the text. I thank Bo Einarsson for proofreading the book. I also thank my colleagues at the institute Götz Alefeld and Willy Dörfler for their support of the book project.

I gratefully acknowledge the help of Neville Holmes who went carefully through great parts of the manuscript, sending back corrections and suggestions that led to many improvements. His help was indeed vital for the completion of the book.

The Karlsruher Universitätsgesellschaft supported typing the first draft of the manuscript.

Karlsruhe, November 2007 Ulrich W. Kulisch

Introduction

In general, a digital computer is understood to be an engine that deals with data, data that are stored as binary digits or bits. The bits stored in a computer can be used to represent all kinds of abstract or concrete objects, objects such as whole numbers, letters of an alphabet, the books in a library, the accounts in a bank, the inhabitants of a town, the text of a law or the features of a disease. A user can get the computer to process the stored data, for example, by translation, searching or sorting. If the computer is functioning correctly then its processing is free of error. A computer that is programmed properly is therefore generally considered to be an infallible tool.

It is quite ironical that this view is often not justified when the digital computer is used for its original purpose – numerical or scientific computation. This book treats the arithmetic processing of real numbers, reviews their properties, and describes how computational error can be reduced and avoided by special design of the computer's arithmetic.

Mathematics uses two methods to define the real numbers. The first – occasionally called the genetic method – begins by defining the natural numbers, for instance using the so-called Peano axioms. Then in turn the integral, the rational, the irrational (real) and the complex numbers are defined. The integral numbers (integers) for which subtraction is always possible are defined as pairs of natural numbers, the rational numbers as pairs of integers. The complex numbers for which the square root always exists are defined as pairs of real numbers.

The step from the rational numbers to the real, however, differs essentially from the other extensions. It can not be done by considering pairs of rational numbers. Different methods are used to define the real numbers. They are logically equivalent. The best-known genetic ways to describe the real numbers are as Dedekind cuts in the set of rational numbers, and as fundamental sequences of rational numbers. In both cases infinitely many rational numbers are needed to define a real number.

The second method defines the real numbers as a conditionally complete, linearly ordered field, as described below. It is then shown that any two conditionally complete, linearly ordered fields are isomorphic. This means that there can be at most one such field. The genetic method, sketched above, shows that one does exist and can be constructed.

Thus the real numbers as defined by the genetic method essentially represent the only realization of a conditionally complete, linearly ordered field. The axioms of this field form the basis of the huge body of analysis and its applications.

In axiomatic geometry the nature of a point or a straight line is irrelevant. All that matters is the relation between points, straight lines and so on, as defined by the axioms. Similarly, analysis is an abstraction in which the nature of a real number is irrelevant. All that matters is the relation between real numbers as defined by the axioms.

Pure mathematics is pure largely because the huge corpus of analysis can be put together in an abstract manner by using only the axioms and without using concrete models. Numbers with special properties are represented by symbols like e or π. The real numbers of the genetic method, or the decimal numbers of everyday life, only serve as proof of existence of nontrivial models or as raw material for exercises.

The situation is very different in applied mathematics. Here, the value of a real number has importance far beyond its symbolic representation or underlying abstract theory. Any programmer knows that the name of a number (its address in storage) and its value (the contents of the storage cell) are quite distinct, and that the value of any number is represented in a specific numbering system.

Applied mathematics, therefore, though starting from axiomatic real analysis, also has to consider the representation of real numbers, algorithms for deriving their representation, and algorithms for computation with those representations. Of course, the well-known algorithms for operations on decimally represented numbers, for instance, encompass all the rules of a conditionally complete, linearly ordered field. However this does not mean that only these algorithms are needed for the successful use of real analysis.

Abstract methods can greatly simplify developments and proofs. They allow work to focus on the essence of a problem. Whenever possible concrete number representations should only be used when abstract methods have been exhausted.

What holds for the real numbers is very much the same for the so-called floating point numbers used in computers, and which will be defined later. Here also it is extremely useful to contrast an abstract, axiomatic definition suitable for theoretical studies, and particular representations suitable for calculation.

Real numbers are usually represented by "power" systems, most familiarly with decimal powers. Here equality and the order relation are simple to define. The arithmetic operations, however, cause difficulties.

A real number may only be exactly representable by an infinite b-adic expansion[2], always if it is irrational, often even if it is rational. Such infinite expansions cannot be exactly added or multiplied. Operations for finite b-adic expansions, however, can easily be carried out using simple algorithms based on single digit operations. Increasing the number of digits of a finite expansion used to approximately represent infinite b-adic expansions can make the error in the result of arithmetic operations, in principle, arbitrarily small.

Operations for infinite b-adic expansions are defined as the limit of the sequence of results obtained by operating on increasingly large prefixes of those expansions. In principle, a computer could approximate this limiting process. In practice, the

[2] for definition see Chapter 5

obvious inefficiency of such an approach discourages its serious implementation even on the fastest computers. (Note, however, that in the old days of Fortran people used to run programs first with single precision reals, then with double precision, then, if the results disagreed, with extended precision.)

Numerical computing, as it is practised today, aims for reasonable and useful approximations, not for exact or arbitrarly precise results. On computers the real numbers are approximated by a subset on which all operations are simply and rapidly carried out. The most common choice for this subset is floating-point numbers with a fixed number of digits in the mantissa. (Frequently the word significand is used instead of mantissa).

The task of numerical analysis is to develop and design algorithms which use floating-point numbers to deliver a reasonably good approximation to the exact result. An essential part of this task is to quantify the error of the computed answer. Managing this quite natural error is the crucial challenge of numerical or scientific computing. In this respect, numerical analysis is completely irrelevant to everyday applications of computers like those mentioned in the opening paragraph of this Introduction. For solving problems of this kind, integer arithmetic, which is exact, is used, or should be, whenever arithmetic is needed.

When early floating-point arithmetic was being developed, its design was often governed by objectives such as simplification of its circuitry, ease of programming, and the nature of the technology available. This often added costs and difficulties for the user.

The computer, however, should be a mathematical tool, with its numerical properties and arithmetic operations precisely and mathematically defined. These properties and operations should be simple to describe and easy to understand so that both the designers and the users of computers can work with a more complete knowledge of the computational process.

The concept of semimorphism is central to the approach to computer arithmetic described in this book. The various models described here lead directly to an understanding of the properties of semimorphism. These properties embody an ordering principle that allows the entire treatment of arithmetic in this book to be understood and developed in close analogy to geometry. The properties of a geometry can be defined and derived as invariants with respect to a certain group of transformations.

In much the same way, mathematical properties of a computer will here be defined as invariants with respect to semimorphisms. The algorithms and hardware circuits for the arithmetic operations in various spaces describe the implementation of semimorphisms on computers. The result is an arithmetic with many desirable properties – high speed, optimal accuracy, theoretical describability, closedness of the theory, usability, applicability, and so on. This similarity to geometry allows computer arithmetic to be presented here in a complete and well-rounded way.

Algorithms and circuits for computer arithmetic are implemented in a way that is natural for the nonspecialist. Doing this helps to avoid those misunderstandings of the computer by its users that are caused by tricky implementations.

Directed roundings and interval operations are essential to computer arithmetic. Intervals bring the continuum to the computer. The two floating-point bounds of an interval represent the continuous set of real numbers between them.

Not only has the development of interval arithmetic had a great effect on the theoretical understanding of arithmetic on the computer, but the use of intervals is necessary to the proper control of rounding errors. Interval arithmetic can guarantee the result of a computation, and thus permit use of the computer for *verification* and *decidability*. Such questions cannot be answered by using arithmetic which rounds in the traditional simple manner. And even with more sophisticated circuits, an interval operation can be made as fast as its corresponding floating-point operation.

The spaces in which numerical algorithms are usually defined and studied are the basis of the computer arithmetic described here. All computation begins with integers and real numbers. Thus numerical algorithms are usually defined in the space $\mathbb{R}$ of real numbers and the vectors $V\mathbb{R}$ or matrices $M\mathbb{R}$ over the real numbers. The corresponding complex spaces $\mathbb{C}$, $V\mathbb{C}$, and $M\mathbb{C}$ are sometimes also used. All these spaces are ordered by the relation $\leq$. In $\mathbb{C}$ and in the vector and matrix spaces $\leq$ is defined componentwise.

For decidability and error control, numerical mathematics also considers algorithms for intervals in these spaces. If we denote the set of intervals over an ordered set $\{M, \leq\}$ by IM, we obtain the spaces $I\mathbb{R}$, $IV\mathbb{R}$, $IM\mathbb{R}$, and $I\mathbb{C}$, $IV\mathbb{C}$, and $IM\mathbb{C}$.

In Figure 1, to which we shall repeatedly refer, a table of spaces is presented. The second column of this figure lists the various spaces, introduced above, in which arithmetic is defined. Algorithms derived and defined in these spaces, however, cannot be used on a computer, in which only finite subsets can be represented. For the real numbers $\mathbb{R}$ the so-called floating-point numbers with a fixed number of digits in the mantissa are commonly used as the subset S. If computation within S is not accurate enough, a larger subset D of $\mathbb{R}$ is often used, where $\mathbb{R} \supset D \supset S$.

Arithmetic must be applicable to the various vector, matrix and interval spaces, as they extend S and D as well as $\mathbb{R}$. This yields the finite spaces VS, MS, IS, IVS, IMS, $\mathbb{C}S$, $V\mathbb{C}S$, $M\mathbb{C}S$, $I\mathbb{C}S$, $IV\mathbb{C}S$, and $IM\mathbb{C}S$, and the corresponding spaces over D. These spaces are listed in the third and fourth columns of Figure 1. Thus $\mathbb{C}S$ is the set of all pairs of S, $V\mathbb{C}S$ is the set of all n-tuples of such pairs, $I\mathbb{C}S$ is the set of all intervals over the ordered set $\{\mathbb{C}S, \leq\}$, and so forth.

In practice, floating-point numbers of single and double length are used for the sets S and D. However, in Figure 1 and in the following, S and D are just symbols for generic subsets of $\mathbb{R}$ with arithmetic properties to be defined.

With the sets of the third and fourth columns of Figure 1 defined, we turn now to the question of defining the operations on these sets. These operations need to approximate those that are defined on the corresponding sets listed in the second column.

$$
\begin{array}{ccccccc}
I & & II & & III & & IV \\
 & & \mathbb{R} & \supset & D & \supset & S \\
 & & V\mathbb{R} & \supset & VD & \supset & VS \\
 & & M\mathbb{R} & \supset & MD & \supset & MS \\
\mathbb{P}\mathbb{R} & \supset & I\mathbb{R} & \supset & ID & \supset & IS \\
\mathbb{P}V\mathbb{R} & \supset & IV\mathbb{R} & \supset & IVD & \supset & IVS \\
\mathbb{P}M\mathbb{R} & \supset & IM\mathbb{R} & \supset & IMD & \supset & IMS \\
 & & \mathbb{C} & \supset & \mathbb{C}D & \supset & \mathbb{C}S \\
 & & V\mathbb{C} & \supset & V\mathbb{C}D & \supset & V\mathbb{C}S \\
 & & M\mathbb{C} & \supset & M\mathbb{C}D & \supset & M\mathbb{C}S \\
\mathbb{P}\mathbb{C} & \supset & I\mathbb{C} & \supset & I\mathbb{C}D & \supset & I\mathbb{C}S \\
\mathbb{P}V\mathbb{C} & \supset & IV\mathbb{C} & \supset & IV\mathbb{C}D & \supset & IV\mathbb{C}S \\
\mathbb{P}M\mathbb{C} & \supset & IM\mathbb{C} & \supset & IM\mathbb{C}D & \supset & IM\mathbb{C}S
\end{array}
$$

Figure 1. Collection of spaces used in numerical computation.

Furthermore, the lines in Figure 1 are not arithmetically independent of each other. For instance: a vector can be multiplied by a scalar as well as by a matrix; an interval vector can by multiplied by an interval as well as by an interval matrix.

This leads to the following preliminary and informal definition of *basic computer arithmetic*:

> **Basic computer arithmetic *comprises all the needed operations defined on all the sets listed in the third and fourth columns of Figure 1 as well as on all combinations of those sets.***

The sets S and D may, for instance, be thought of as floating-point numbers of single and double mantissa length. In a programming system data types which might be called `real`, `complex`, `interval`, `cinterval`, `rvector`, `cvector`, `ivector`, `civector`, and so on, would correspond to the spaces of the third and fourth columns of the figure. In a good programming system, operations for these spaces should be available for all combinations of data types within either column.

The first part of this book develops definitions for these many operations and also specifies simple structures as settings for them. The second part then deals with the implementation of these operations on computers. In particular, it is shown that the operations can be built up modularly from relatively few fundamental algorithms and routines. These can be implemented in fast hardware circuits.

Arithmetic operations can now be defined for all the sets in the third and fourth and possibly more such columns of Figure 1.

Let M be one of these sets and let us assume that certain operations and relations are defined for its elements. Further, let $\overline{M}$ be the rules (axioms) given for the elements of M. The commutative law of addition might be one such rule. Then we call the pair $\{M, \overline{M}\}$ a structure and refer to it as the structure of the set M. The structures of the sets $\mathbb{R}$, $V\mathbb{R}$, $M\mathbb{R}$, $\mathbb{C}$, $V\mathbb{C}$, and $M\mathbb{C}$ are well known. Now let M be one of these sets and $\circ$ one of the operations defined on M. Let $\mathbb{P}M$ be the power set of M, the set of all subsets of M. The operation $\circ$ on M can be extended to the power set $\mathbb{P}M$ by the following definition[3]:

$$\bigwedge_{A,B \in \mathbb{P}M} A \circ B := \{a \circ b \mid a \in A \wedge b \in B\}.$$

If we apply this definition to all operations $\circ$ of M, the structure of the power set $\{\mathbb{P}M, \overline{\mathbb{P}M}\}$ can be derived from the structure $\{M, \overline{M}\}$. By this application, arithmetic operations and a corresponding structure are defined on the power sets of the spaces $\mathbb{R}$, $V\mathbb{R}$, $M\mathbb{R}$, $\mathbb{C}$, $V\mathbb{C}$, and $M\mathbb{C}$, the sets of the first column of Figure 1.

In brief, the operations and the structure $\{M, \overline{M}\}$ of the leftmost space of every row in Figure 1 are always known. These operations and structures can be used to define and derive the operations and structures of the thirty remaining spaces. This need not be done for each subset space individually. Instead we do this for all the rows of Figure 1 at once by using an abstract mapping principle. The same principle would be applied if there would be more columns at the right in the table of Figure 1.

Approximating a given structure $\{M, \overline{M}\}$ by a structure $\{N, \overline{N}\}$ might be tried in such a way that the mapping has properties like isomorphism or homomorphism. However, N as a subset of M can be of different cardinality. There are infinitely many elements in the sets of the first two columns of Figure 1 but a finite number in those of the last two columns. There can be no one-to-one mapping, and thus no isomorphism, between sets of different cardinality.

But even a homomorphism would preserve the structure of a group, a ring, or a vector space. Simple examples can be used to show that, in the case of floating-point numbers, no sensible homomorphism can be realized. It would be well, however, to stay as close to homomorphism as possible. We shall see later that the following four properties can be realized for all the set/subset pairs of Figure 1:

If M is a space in Figure 1 with N an adjacent subset to its right, then for every arithmetic operation $\circ$ in M an operation $\boxed{\circ}$ in N is defined by

(RG) $\displaystyle\bigwedge_{a,b \in N} a \boxed{\circ} b := \square(a \circ b),$

where $\square : M \to N$ is a mapping from M onto N which is called a rounding if it has the following properties:

(R1) $\displaystyle\bigwedge_{a \in N} \square a = a,$

[3] See Appendix A for the definition of the symbol $\bigwedge$.

(R2) $\bigwedge_{a,b\in M} (a \le b \Rightarrow \square a \le \square b).$

Many roundings have the additional property

(R4) $\bigwedge_{a\in M} \square(-a) = -\square a.$

These properties (RG), (R1), (R2), and (R4) can be shown to be necessary conditions for a homomorphism between ordered algebraic structures $\{M, \overline{M}\}$ and $\{N, \overline{N}\}$. We therefore call a mapping with these properties a *semimorphism*. In a semimorphism, (RG) defines the approximating operations $\boxdot$ in the subset N for all operations $\circ$ in M. (R1) is a very natural property for a rounding.

(R2) imposes ordering on the semimorphism, so the rounding is *monotone*, and (R4) makes it *antisymmetric*.

Although the properties (R1), (R2), and (R4) do not define the rounding $\square : M \to N$ uniquely, the semimorphism is responsible not only for the mapping of the elements but also for the resulting structure $\{N, \overline{N}\}$. This means that the set of rules $\overline{N}$ valid in N depends essentially on the properties of semimorphism. Indeed, $\overline{N}$ can be defined as the set of rules that is invariant with respect to a semimorphism, that is, $\overline{N} \subseteq \overline{M}$, and the structure $\{N, \overline{N}\}$ is a generalization of $\{M, \overline{M}\}$. If we consider the mapping between the second and third columns in any row of Figure 1, we get a proper generalization $\overline{N} \subset \overline{M}$. Going from the third to the fourth column – and possibly further – we always have $\overline{N} = \overline{M}$. We shall see later that the structures of ordered or weakly ordered or isotonely ordered ringoids or vectoids represent the properties that are invariant with respect to semimorphism. The ringoid is a generalization of the mathematical structure of a ring and the vectoid of that of a vector space.

That semimorphisms preserve reasonable mathematical structures is another strong reason for using semimorphisms to define all arithmetic operations in all subsets of Figure 1.

For the power set and the interval spaces in Figure 1 the order relation in (R2) is the subset relation $\subseteq$. A rounding from any power set or interval set M onto its subset N is defined by properties (R1), (R2) (with $\subseteq$ replacing $\le$), and also by

(R3) $\bigwedge_{a\in M} a \subseteq \square a.$

These set and interval roundings are also antisymmetric, having the property (R4).

Additional important roundings from the real numbers onto the floating-point numbers are the monotone roundings directed downward $\triangledown$ and upward $\vartriangle$. These directed roundings are uniquely defined by (R1), (R2) and (R3). Arithmetic operations are also defined for these roundings by (RG).

With the five rules (RG) and (R1,2,3,4), many arithmetic operations are defined in the computer representable subsets of the twelve spaces in the second column of Figure 1.

A definition of computer arithmetic is only practical if it can be implemented by fast routines and circuitry. We shall later show special hardware circuits which prove not only that all the semimorphisms of Figure 1 are indeed practical, but also that all the arithmetic operations can be realized with the highest possible accuracy. In the product spaces many of these operations are not at all like those based on elementary floating-point arithmetic.

For floating-point numbers it will later be shown that all the arithmetic operations of the third and fourth columns of Figure 1 can be made very fast from a few modular building blocks. These building blocks implement the five operations $+, -, \times, /, \cdot$, each with the three roundings $\square, \bigtriangledown, \bigtriangleup$, and with the case selections needed for interval multiplication and division. Here $\cdot$ means the scalar product of two vectors, $\square$ is an antisymmetric monotone rounding like rounding to nearest, and $\bigtriangledown$ and $\bigtriangleup$ are the downward and upward monotone roundings from the real numbers into the floating-point numbers. All the fifteen operations $\boxed{\circ}, \bigtriangledown\hspace{-0.75em}{\scriptstyle\circ}\ , \bigtriangleup\hspace{-0.75em}{\scriptstyle\circ}\ $ with $\circ \in \{+, -, \times, /, \cdot\}$ of Figure 2 are defined by (RG). In case of the scalar product, a and b are the vectors $a = (a_i)$, $b = (b_i)$ with a finite number of elements.

$$\begin{array}{llllllcl}
\boxplus, & \boxminus, & \boxtimes, & \boxslash, & \boxdot, & a \boxdot b & = & \square \sum_{i=1}^{n} a_i \cdot b_i, \\
\bigtriangledown\hspace{-0.75em}{\scriptstyle +}\ , & \bigtriangledown\hspace{-0.75em}{\scriptstyle -}\ , & \bigtriangledown\hspace{-0.75em}{\scriptstyle \times}\ , & \bigtriangledown\hspace{-0.75em}{\scriptstyle /}\ , & \bigtriangledown\hspace{-0.75em}{\scriptstyle \cdot}\ , & a \bigtriangledown\hspace{-0.75em}{\scriptstyle \cdot}\ b & = & \bigtriangledown \sum_{i=1}^{n} a_i \cdot b_i, \\
\bigtriangleup\hspace{-0.75em}{\scriptstyle +}\ , & \bigtriangleup\hspace{-0.75em}{\scriptstyle -}\ , & \bigtriangleup\hspace{-0.75em}{\scriptstyle \times}\ , & \bigtriangleup\hspace{-0.75em}{\scriptstyle /}\ , & \bigtriangleup\hspace{-0.75em}{\scriptstyle \cdot}\ , & a \bigtriangleup\hspace{-0.75em}{\scriptstyle \cdot}\ b & = & \bigtriangleup \sum_{i=1}^{n} a_i \cdot b_i.
\end{array}$$

Figure 2. The fifteen fundamental computer operations.

Fast hardware circuitry for all these operations is developed in Part 2.

The mapping principle of a semimorphism can also be derived directly from special models of the sets in Figure 1. For instance, consider the mapping of the power set of the complex numbers $\mathbb{PC}$ into the complex number intervals $I\mathbb{C}$. An interval $[a, b]$ of two complex numbers a and b with $a \leq b$ is a rectangle in the complex plane with sides parallel to the axes. Performing an operation $\circ$ for two complex intervals A and B as power set elements (see Figure 3) does not typically yield an interval result, but the more specific result $A \circ B$ in the power set $\mathbb{PC}$.

On a computer the result of an interval operation must be an interval. The power set result $A \circ B$ must therefore be mapped onto the least interval that contains it, a rectangle as shown in Figure 3. This mapping $\square : \mathbb{PC} \to I\mathbb{C}$ is a rounding having the properties (R1,2,4) and (RG) of a semimorphism, and its order relation is the set inclusion $\subseteq$. If the set $A \circ B$ is already an interval the rounding has no effect, so (R1) holds. If we extend the set $A \circ B$ this enlarges the least interval that includes it, so (R2) holds. (R4) holds by symmetry. The result fulfills (RG) $A \boxed{\circ} B := \square(A \circ B)$, by construction. Also, the rounding $\square : \mathbb{PC} \to I\mathbb{C}$ has the property (R3) of being upwardly directed which means that $A \subseteq \square A$ for all $A \in \mathbb{PC}$. We shall see later that

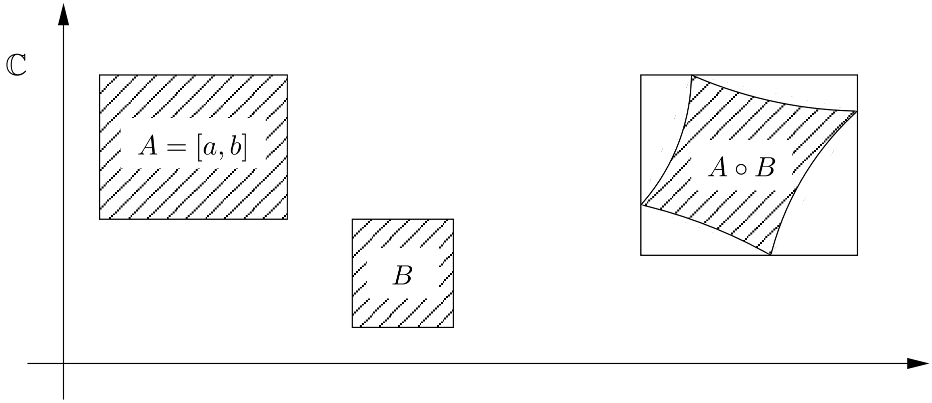


Figure 3. Operation for complex intervals.

the properties (R1,2,3) define the rounding $\square$ uniquely. $\square A$ is the least interval that includes A.

Another example is the basic pair of spaces $\mathbb{R}$ and D of Figure 1. If we add two floating-point numbers a and b (row 1), then the exact sum $a + b$ is not in general a floating-point number in D (Figure 4), and the result must be rounded into D. Figure 4 shows that the process of rounding conforms to (R1), (R2) and (R4). The order relation is $\leq$. Floating-point addition is defined by (RG): $a \boxplus b := \square(a + b)$.

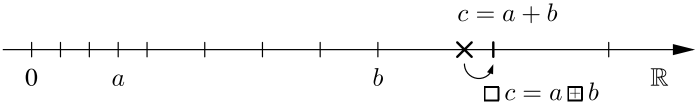


Figure 4. Floating-point addition.

The operations for other pairs of the spaces in Figure 1 are not easy to illustrate simply, but semimorphism provides a mapping on which the implementation of very fast and accurate algorithms and hardware circuits for all those operations can be based.

Property (RG) of a semimorphism requires that every operation be performed in such a way that it produces the same result as if the mathematically correct operation were performed in the basic space M and the result rounded into the computer representable subset N. Unlike conventional floating-point arithmetic's approximation of the arithmetic operations in the product spaces (complex numbers, vectors, matrices), all operations defined by semimorphism are optimal because no better computer representable approximation to the true result (with respect to the prescribed rounding) is possible. In other words, between the exact and the computed results of an operation there is no other element of the corresponding computer representable subset. This can easily be shown. If $a, b \in N$ and $\alpha, \beta \in N$ are the highest lower and lowest upper bound respectively of the correct result $a \circ b$ for any operation $\circ$ in M so that

$$\alpha \leq a \circ b \leq \beta$$

then

$$\square\alpha \underset{\text{(R1)}}{=} \alpha \underset{\text{(R2)}}{\leq} \square(a \circ b) \underset{\text{(RG)}}{=} a \boxed{\circ} b \underset{\text{(R2)}}{\leq} \square\beta \underset{\text{(R1)}}{=} \beta.$$

Thus, all semimorphic computer operations are accurate to within 1 ulp (unit in the last place). Half ulp accuracy is achieved by rounding to nearest. In the product spaces the rounding is defined componentwise so this property holds for every component.

Thus the definition of arithmetic operations by semimorphisms in all subsets of Figure 1 results in computer arithmetic with optimal accuracy. This in turn simplifies the error analysis of numerical algorithms. Further, the following compatibility properties hold between the structures $\{M, \overline{M}\}$ and $\{N, \overline{N}\}$ for all operations $\circ$ in M:

(RG1) $\bigwedge\limits_{a,b \in N} \quad (a \circ b \in N \Rightarrow a \boxed{\circ} b = a \circ b),$

(RG2) $\bigwedge\limits_{a,b,c,d \in N} \quad (a \circ b \leq c \circ d \Rightarrow a \boxed{\circ} b \leq c \boxed{\circ} d),$

(RG4) $\bigwedge\limits_{a \in N} \quad -a = \boxminus a := (-e) \boxdot a.$

For definition of e see Section 1.4. Also, if the rounding $\square : M \to N$ is upwardly directed then property (RG3) holds:

(RG3) $\bigwedge\limits_{a,b \in N} \quad a \circ b \leq a \boxed{\circ} b.$

In summary we remark that

> ***the formulas (RG), (R1), (R2), (R3), and (R4) for semimorphisms can and should be used as an axiomatic definition of computer arithmetic in the context of programming languages.***

A programmer should know just what the result will be of any arithmetic operation used in a program.

Basic computer arithmetic will be extended to what is called *advanced computer arithmetic* in Chapters 7 and 8 of the book.

Part I

Theory of Computer Arithmetic

Chapter 1
First Concepts

In this chapter we give an axiomatic characterization of the essential properties of the sets and subsets shown in Figure 1. We then define the notion of rounding from a set M onto a subset N and study the key properties of certain special roundings and their interactions with simple arithmetic operations. To accomplish this, we employ several lattice theoretic concepts that are developed at the beginning of this chapter.

1.1 Ordered Sets

As part of analysis the spaces listed in column two of Figure 1 carry three kinds of mathematical structures: an algebraic structure, a topological or metric structure, and an order structure. These are coupled by certain compatibility properties, for instance: $a \leq b \Rightarrow a + c \leq b + c$. When mapped on computer representable subsets by a rounding these structures are considerably weakened. Since the rounding is a monotone function the changes to the order structure are minimal. Thus the order structure plays a key role and is of particular importance for the arguments in this book. Therefore we begin this chapter by listing a few well-known concepts and properties of ordered sets.

Definition 1.1. A relation $\leq$ in a set M is called an *order relation*, and $\{M, \leq\}$ is called an *ordered set* if we have

(O1) $\bigwedge_{a \in M} a \leq a$ (reflexivity),

(O2) $\bigwedge_{a,b,c \in M} (a \leq b \wedge b \leq c \Rightarrow a \leq c)$ (transitivity),

(O3) $\bigwedge_{a,b \in M} (a \leq b \wedge b \leq a \Rightarrow a = b)$ (antisymmetry).

An ordered set is called *linearly* or *totally ordered* if in addition

(O4) $\bigwedge_{a,b \in M} a \leq b \vee b \leq a.$ ■

$\{M, \leq\}$ being an ordered set just means that there is an relation defined in M. It does not mean that for any two elements $a, b \in M$ either $a \leq b$ or $b \leq a$ holds. The latter is valid in linearly ordered sets. If for two elements $a, b \in M$ neither $a \leq b$ nor $b \leq a$, then a and b are called *incomparable*, in notation $a \| b$. Ordered sets are sometimes also called partially ordered sets. Since we shall consider many special partially ordered sets, we will suppress the modifier to avoid bulky expressions. This practice is also quite common in the literature.

If $\{M, \leq\}$ is an ordered (resp. a linearly ordered) set and $T \subseteq M$, then $\{T, \leq\}$ is also an ordered (resp. a linearly ordered) set.

To each pair $\{a, b\}$ of elements in an ordered set $\{M, \leq\}$, where $a \leq b$, the *interval* $[a, b]$ is defined by

$$[a, b] := \{x \in M \mid a \leq x \leq b\}.$$

A subset $T \subseteq M$ is called *convex* if with two elements $a, b \in T$, the whole interval $[a, b] \in T$. The smallest convex superset of a set is called its *convex hull.*

Definition 1.2. A relation $<$ in a set M is called *antireflexive* and the pair $\{M, <\}$ an *antireflexively ordered set* if we have

(AO1) $\bigwedge\limits_{a \in M} \neg(a < a)$ (antireflexivity),

(AO2) $\bigwedge\limits_{a,b,c \in M} (a < b \wedge b < c \Rightarrow a < c)$ (transitivity).

■

The following lemma describes a well-known relation between the orderings of the Definitions 1.1 and 1.2.

Lemma 1.3. *(a) In an ordered set $\{M, \leq\}$ an antireflexive ordering is defined by*

$$a < b \quad :\Leftrightarrow \quad (a \leq b \wedge a \neq b).$$

(b) In an antireflexively ordered set $\{M, <\}$ an order relation is defined by

$$a \leq b \quad :\Leftrightarrow \quad (a < b \vee a = b).$$

■

The proof of this lemma, being straightforward, is omitted. We note that throughout this book we shall use the sign $<$ in an ordered set $\{M, \leq\}$ and the sign $\leq$ in an antireflexively ordered set $\{M, <\}$ in the sense of Lemma 1.3. This is not always the case in the literature.

We list a few well-known examples of ordered sets:

Examples. 1. The set $\{\mathbb{R}, \leq\}$ of real numbers is a linearly ordered set.

2. If M is a set and $\mathbb{P}M$ denotes the *power set* of M, which is defined as the set of all subsets of M, then with inclusion as an order relation $\{\mathbb{P}M, \subseteq\}$ is an ordered set, and we have the properties

(O1) $\bigwedge\limits_{A \in \mathbb{P}M} A \subseteq A,$

(O2) $\bigwedge\limits_{A,B,C \in \mathbb{P}M} (A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C),$

(O3) $\bigwedge\limits_{A,B \in \mathbb{P}M} (A \subseteq B \wedge B \subseteq A \Rightarrow A = B).$

3. If $\{M, \leq\}$ is an ordered set and M^n denotes the *product set*, which is defined as the set of all n-tuples of elements of M and

$$x := (x_i) := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad y := (y_i) := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \in M^n,$$

then by the definition

$$x \leq y \quad :\Leftrightarrow \quad \bigwedge_{i=1(1)n} x_i \leq y_i,$$

$\{M^n, \leq\}$ becomes an ordered set.

In an ordered set $\{M, \leq\}$ an element a is called the *lower neighbor* of an element b if $a < b$ and no other element of M lies between them; i.e., there exists no element $c \in M$ such that $a < c < b$. The concept of lower neighbor can be used to draw a figure of any finite ordered set. We just have to assign every element of M to a point of the plane and place a lower than b whenever $a < b$. Then we connect every point to each of its lower neighbors by a straight line segment. The resulting figure is called the *order diagram* of the ordered set $\{M, \leq\}$.

Figure 1.1 shows the order diagram of ordered sets of 9 and 16 elements. The set of Figure 1.1 (b) consists of two subsets, the respective elements of which are incomparable.

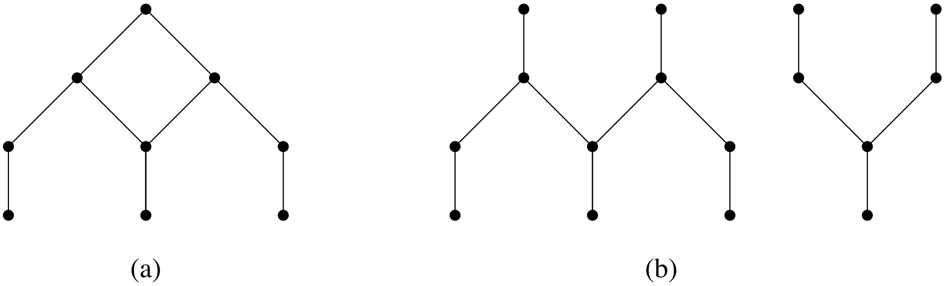


Figure 1.1. Order diagrams.

An element x of an ordered set $\{M, \leq\}$ is called a *maximal* (resp. *minimal*) *element* if

$$\bigwedge_{a \in M} \neg(x < a) \qquad \left(\text{resp. } \bigwedge_{a \in M} \neg(a < x)\right).$$

Every finite ordered set has at least one maximal and one minimal element. To see this, let $M = \{a_1, a_2, \ldots, a_n\}$ and set $x_1 = a_1$. Set $x_2 = a_2$ if $a_1 < a_2$, but set $x_2 = x_1$ otherwise. In general, set $x_k = a_k$ if $x_{k-1} < a_k$ and $x_k = x_{k-1}$ otherwise.

Then x_n is a maximal element. An ordered set may have more than one maximal or minimal element. See Figure 1.1. An infinite set, however, need have neither a maximal nor a minimal element.

Now we can show that every finite, nonvoid ordered set can be represented by an order diagram. If $M = \{a\}$, this is clear. Let us assume now that the statement is true for sets with $n-1$ elements, and let M be a set with n elements. Since M consists of a finite number of elements, it has at least one maximal element. Call $S \subseteq M$ the subset of M containing $n-1$ elements, which is obtained by removing one such maximal element. S has an order diagram. Now we join the element a to the order diagram of S and connect it with all its lower neighbors. The resulting figure is an order diagram of M.

Properties of infinite ordered sets may also be illustrated by order diagrams.

In the remainder of this section we define and discuss various constructs associated with ordered sets.

Definition 1.4. An element of an ordered set $\{M, \leq\}$ is called the *least element* $o(M)$ (resp. the *greatest element* $i(M)$) if

$$\bigwedge_{a \in M} o(M) \leq a \qquad \left(\text{resp. } \bigwedge_{a \in M} a \leq i(M)\right). \qquad \blacksquare$$

That both $o(M)$ and $i(M)$ are well defined is the assertion of the following theorem.

Theorem 1.5. *An ordered set has at most one least and at most one greatest element.*

Proof. Suppose a and b are least elements of M. Then $a \leq b \wedge b \leq a \Rightarrow a = b$ by (O3). ■

The concepts of least and greatest elements and minimal and maximal elements are of course different. They can be illustrated by order diagrams. See, for instance, Figure 1.1. A least element is always minimal and a greatest element always maximal. The converse is not true. See Figure 1.1.

The power set $\{\mathbb{P}M, \subseteq\}$ of a set M is an important example that contains a least and a greatest element. In particular, $o(\mathbb{P}M) = \varnothing$, and $i(\mathbb{P}M) = M$.

Definition 1.6. Let $\{M, \leq\}$ be a nonvoid ordered set and $S \subseteq M$. An element $a \in M$ is called a *lower* (resp. an *upper*) *bound* of S in M if

$$\bigwedge_{b \in S} a \leq b \qquad \left(\text{resp. } \bigwedge_{b \in S} b \leq a\right).$$

We denote the set of all lower (resp. upper) bounds of S in M by

$$L(S) := \Big\{a \in M \mid \bigwedge_{b \in S} a \leq b\Big\} \qquad \Big(\text{resp. } U(S) := \Big\{a \in M \mid \bigwedge_{b \in S} b \leq a\Big\}\Big).$$

If $S = \{s\}$, we simply write $L(s)$ resp. $U(s)$ instead of $L(\{s\})$ resp. $U(\{s\})$. S is called *bounded* in M if $L(S) \neq \varnothing$ and $U(S) \neq \varnothing$. The greatest lower bound (resp. the least upper bound) if it exists, is called the *infimum* (resp. the *supremum*) of S. Symbolically

$$\inf S := i(L(S)) \qquad (\text{resp. } \sup S := o(U(S)))\,. \qquad \blacksquare$$

Not every subset of an ordered set has an infimum or a supremum or even lower or upper bounds. For an illustration see, for instance, the ordered sets drawn in Figure 1.1.

According to Theorem 1.5, the infimum (resp. the supremum) of a subset $S \subseteq M$, if it exists, is uniquely determined.

The case $S = \varnothing$ is not excluded in Definition 1.6. Since in the case of the empty set the definition of bounds requires nothing,

$$\bigwedge_{a\in M} \bigwedge_{b\in\varnothing} a \leq b \qquad \wedge \qquad \bigwedge_{a\in M} \bigwedge_{b\in\varnothing} b \leq a$$

every element of M is a lower and an upper bound of the empty subset $\varnothing \subseteq M$.

If an ordered set $\{M, \leq\}$ has a least element $o(M)$ and a greatest element $i(M)$, we have

$$\inf M = o(M), \qquad \sup M = i(M),$$

and

$$\inf \varnothing = i(M), \qquad \sup \varnothing = o(M).$$

If $\varnothing \neq S \subseteq M$, it is always true that $\inf S \leq \sup S$.

If a subset S of an ordered set $\{M, \leq\}$ has an infimum (resp. a supremum) t, where in particular $t \in S$, then t is also least and therefore also a minimal element of S (resp. greatest and therefore also a maximal element of S). Conversely, if a subset S of an ordered set has a least (resp. greatest) element t, then t is also the infimum (resp. supremum) of S in M.

Now let $\{M, \leq\}$ be an ordered set and S_1, S_2 nonvoid subsets of M each of which is assumed to have an infimum and a supremum in M. Then the following properties hold:

(a) $S_1 \subseteq S_2 \Rightarrow \inf S_2 \leq \inf S_1 \wedge \sup S_1 \leq \sup S_2$,

(b) $\bigwedge\limits_{s_1\in S_1} \bigwedge\limits_{s_2\in S_2} (s_1 \leq s_2 \Rightarrow s_1 \leq \sup S_1 \leq \inf S_2 \leq s_2)$.

The proofs of these statements come directly from the preceding definitions and properties. We indicate the one for (b):

$$\bigwedge_{s_1\in S_1} \bigwedge_{s_2\in S_2} (s_1 \leq \sup S_1 \wedge \inf S_2 \leq s_2 \wedge s_1 \leq \inf S_2 \wedge \sup S_1 \leq s_2) \Rightarrow \sup S_1 \leq \inf S_2.$$

1.2 Complete Lattices and Complete Subnets

We begin our discussion of lattices with the following definition.

Definition 1.7. Let $\{M, \leq\}$ be an ordered set. Then

(O5) M is called a *lattice* if for any two elements $a, b \in M$, the $\inf\{a, b\}$ and the $\sup\{a, b\}$ exist;

(O6) M is called *conditionally complete* if for every nonempty, bounded subset $S \subseteq M$, the $\inf S$ and the $\sup S$ exist;

(O7) M is called *completely ordered* or a *complete lattice* if every subset $S \subseteq M$ has an infimum and a supremum. ■

Every finite subset $S = \{a_1, a_2, \ldots, a_n\}$ of a lattice has an infimum and a supremum. We prove this statement by induction. By definition any subset of two elements has an infimum and a supremum. Let us assume now that the assertion is correct for subsets of $n-1$ elements. Then $\inf\{a_1, a_2, \ldots, a_{n-1}\}$ exists. We consider the element $d := \inf\{\inf\{a_1, a_2, \ldots, a_{n-1}\}, a_n\}$. Then d is obviously a lower bound of S. Now let c be any lower bound of S. Then $c \leq \inf\{a_1, a_2, \ldots, a_{n-1}\}$ and $c \leq a_n$ and therefore $c \leq d$, i.e. d is the greatest lower bound of S, completing the induction. Moreover we have also observed that $\inf\{a_1, a_2, \ldots, a_n\} = \inf\{\inf\{a_1, a_2, \ldots, a_{n-1}\}, a_n\}$, i.e., that the inf (resp. sup) is associative.

Every finite lattice, therefore, has a least and a greatest element. Every finite lattice, furthermore, is complete, i.e., is a complete lattice, since besides the empty set, infima and suprema are only to be considered for finite subsets, and we have already observed that the infimum and supremum of the empty subset equal the greatest and the least element, respectively.

Since every finite lattice is an ordered set, every finite lattice can be represented by an order diagram. The order diagram of a lattice is distinguished by the fact that any two elements are downwardly and upwardly connected with other elements of the set.

Of course every completely ordered set is a lattice and every complete lattice is conditionally complete.

Since the definitions of the infimum and the supremum in an ordered set are completely dual, the following *duality principle* is valid throughout lattice theory:

> *If in any valid lattice theoretic statement or theorem the operations* $\leq$, inf, sup *are replaced by* $\geq$, sup, inf, *respectively, a valid lattice theoretic statement or theorem results.*

In the definition of a complete lattice, the case $S = M$ is included. Therefore, $\inf M$ and $\sup M$ exist. Since they are elements of M, $\inf M$ is the least element and $\sup M$ is the greatest element of M. Every complete lattice, therefore, has a least element $o(M)$ and a greatest element $i(M)$.

To continue our development, it is convenient to have the following theorem established.

Theorem 1.8. *Let $\{M, \leq\}$ be an ordered set with the property that every subset has an infimum (or a supremum). Then $\{M, \leq\}$ is a complete lattice.*

Proof.[1] We are given that every subset $S \subseteq M$ has an infimum, and we show that it has a supremum also. Let $U(S)$ be the set of upper bounds of S. (We may suppose that $U(S) \neq \varnothing$ since $\inf \varnothing$ exists by hypothesis and $\inf \varnothing = i(M)$). By assumption the element $u_0 = \inf U(S)$ exists. We show that $u_0 = \sup S$. We have

$$\bigwedge_{s \in S} \bigwedge_{u \in U(S)} s \leq u \Rightarrow \bigwedge_{s \in S} s \leq u_0 = \inf U(S),$$

i.e., u_0 is upper bound of S. Now if u is any upper bound of S, then $u \in U(S)$ and consequently $u_0 = \inf U(S) \leq u$. Therefore, u_0 is the least upper bound of S, i.e., $u_0 = \sup S = \inf U(S)$. ■

According to Theorem 1.8, to verify that an ordered set is a complete lattice it is sufficient to show that every subset has a greatest lower bound *or* has a least upper bound.

The concepts of a conditionally completely ordered set and a complete lattice are closely related. If a conditionally completely ordered set has a least and a greatest element it is a complete lattice.

Now let $\{M, \leq\}$ be a conditionally completely ordered set without a least (resp. greatest) element. We add to M an additional element y (resp. z) which by definition is assumed to have the property

$$\bigwedge_{x \in M} y \leq x \qquad \left(\text{resp.} \bigwedge_{x \in M} x \leq z\right).$$

Then y is the least (resp. z greatest) element of $\{M \cup \{y\}, \leq\}$ (resp. $\{M \cup \{z\}, \leq\}$), and if M has neither a least nor a greatest element then $\{M \cup \{y\} \cup \{z\}, \leq\}$ is a complete lattice. Every ordered set that is conditionally complete thus can be made into a complete lattice by adjoining a least and a greatest element.

This is a well-known procedure in the case of real numbers $\mathbb{R}$. $\{\mathbb{R}, \leq\}$ is a conditionally completely ordered set. In real analysis it is shown that every nonempty bounded subset of real numbers has an infimum and a supremum. If we adjoin $-\infty$ and ∞ to form $\{\mathbb{R} \cup \{-\infty\} \cup \{\infty\}, \leq\}$, we obtain a complete lattice with the least element $-\infty$ and the greatest element ∞. Similarly, we obtain a complete lattice by adjoining the endpoints to an open interval of real numbers.

The following two theorems provide additional examples of complete lattices.

[1] We prove this theorem only for the case that an infimum always exists. The proof in the case of the supremum is dual. In the proofs of many following theorems, we omit the dual case without comment.

Theorem 1.9. *Let $\{M_i, \leq_i\}$, $i = 1(1)n$, be complete lattices. The product set $M = M_1 \times M_2 \times \ldots \times M_n$ is defined as the set of all n-tuples $a = (a_1, a_2, \ldots, a_n)$ with $a_i \in M_i, i = 1(1)n$. Let $b = (b_1, b_2, \ldots, b_n) \in M$. If we define a relation $\leq$ in M by*

$$a \leq b \quad :\Leftrightarrow \quad \bigwedge_{i=1(1)n} a_i \leq_i b_i,$$

then $\{M, \leq\}$ is a complete lattice.

Proof. It is clear that $\{M, \leq\}$ is an ordered set. Let $S \subseteq M$ and consider the set of projections $S_j := \{s_j \mid s \in S\}$, where $s = (s_1, s_2, \ldots, s_n)$. Then

$$\bigwedge_{i=1(1)n} \bigwedge_{s_i \in S_i} \inf S_i \leq s_i \quad \Rightarrow \quad \bigwedge_{s \in S} (\inf S_1, \inf S_2, \ldots, \inf S_n) \leq s.$$

This means that $(\inf S_i) := (\inf S_1, \inf S_2, \ldots, \inf S_n)$ is a lower bound of S. We show that it is the greatest lower bound. Let $c := (c_i) := (c_1, c_2, \ldots, c_n)$ be any lower bound of S. Then

$$\bigwedge_{i=1(1)n} \bigwedge_{s_i \in S_i} c_i \leq s_i \quad \Rightarrow \quad \bigwedge_{i=1(1)n} c_i \leq \inf S_i \quad \Rightarrow \quad (c_i) \leq (\inf S_i),$$

i.e., $(\inf S_i)$ is the greatest lower bound of S. This means that $\inf S = (\inf S_i)$.

By duality we get $\sup S = (\sup S_i)$. ■

Theorem 1.10. *The power set of a set M with inclusion $\subseteq$ as an order relation, $\{\mathbb{P}M, \subseteq\}$, is a complete lattice. For each subset of $\mathbb{P}M$, the infimum is the intersection of the elements of the subset, and the supremum is the union.*

Proof. $\{\mathbb{P}M, \subseteq\}$ is an ordered set with the least element $o(\mathbb{P}M) = \varnothing$ and the greatest element $i(\mathbb{P}M) = M$. Let $\mathcal{A} \subseteq \mathbb{P}M$ and consider the sets of all lower bounds and upper bounds of $\mathcal{A}$:

$$L(\mathcal{A}) := \Big\{B \in \mathbb{P}M \mid \bigwedge_{A \in \mathcal{A}} B \subseteq A\Big\}, \qquad U(\mathcal{A}) := \Big\{B \in \mathbb{P}M \mid \bigwedge_{A \in \mathcal{A}} A \subseteq B\Big\}.$$

Then $\inf \mathcal{A}$ and $\sup \mathcal{A}$ are, if they exist, defined by

$$\inf \mathcal{A} := i(L(\mathcal{A})), \qquad \sup \mathcal{A} := o(U(\mathcal{A})).$$

Now let $X, Y \in \mathbb{P}M$ have the following properties:

$$X := \Big\{a \in M \mid \bigwedge_{A \in \mathcal{A}} a \in A\Big\}, \qquad Y := \Big\{a \in M \mid \bigvee_{A \in \mathcal{A}} a \in A\Big\}.$$

Then X is the intersection and Y is the union. We show that $X = \inf \mathcal{A}$ and $Y = \sup \mathcal{A}$. By the definition of a subset and the definition of X, we get

$$\bigwedge_{A \in \mathcal{A}} X \subseteq A,$$

i.e., X is lower bound of $\mathcal{A}$, $X \in L(\mathcal{A})$. We still have to show that X is the greatest lower bound. Let $C \in \mathbb{P}M$ be any other lower bound of $\mathcal{A}$. Then

$$\bigwedge_{A \in \mathcal{A}} C \subseteq A,$$

or by definition of a subset,

$$\bigwedge_{c \in C} \bigwedge_{A \in \mathcal{A}} c \in A.$$

Therefore, $C \subseteq X$, i.e., X is the greatest lower bound, $X = \inf \mathcal{A}$. By duality we obtain $\sup \mathcal{A} = Y$. ■

As an illustration, Figure 1.2 shows the order diagram of the power set $\mathbb{P}M$ of the finite set $M := \{a, b, c\}$.

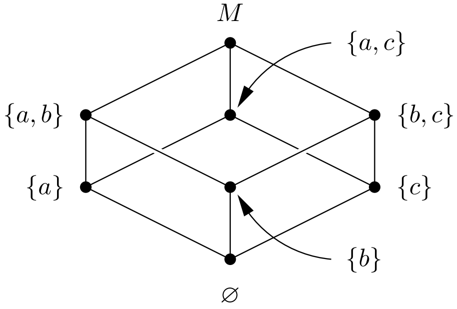


Figure 1.2. Order diagram of the power set of the set $M = \{a, b, c\}$.

Now let $\{M, \leq\}$ be a lattice and $S \subseteq M$. Then $\{S, \leq\}$ is an ordered set. $\{S, \leq\}$ may be a lattice. Let us consider the example represented in Figure 1.3.

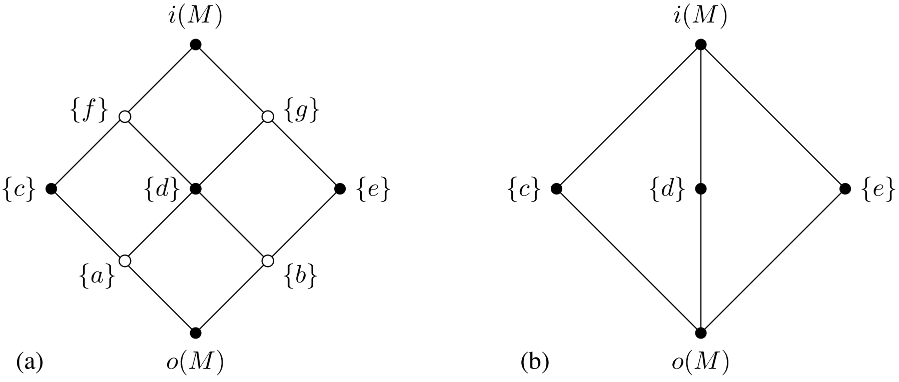


Figure 1.3. Example for the concept of subnet.

The ordered set $\{M, \leq\}$ of the nine points in Figure 1.3 (a) clearly represents a lattice. The subset $S \subseteq M$ of the solid points in Figure 1.3 with respect to the same order relation also represents a lattice, Figure 1.3 (b). In general, however, the infimum and supremum taken in $\{S, \leq\}$ are different from those taken in $\{M, \leq\}$. For example, in $\{M, \leq\}$, $\sup\{c, d\} = f$, while in $\{S, \leq\}$, $\sup\{c, d\} = i(M)$. This leads to the following definition.

Definition 1.11. Let $\{M, \leq\}$ be an ordered set and $S \subseteq M$. If $\{S, \leq\}$ is a lattice, it is called a *subnet* of $\{M, \leq\}$. A subnet is called an *inf-subnet* (resp. a *sup-subnet*) if

$$\bigwedge_{a,b\in S} \inf_M\{a, b\} = \inf_S\{a, b\} \qquad \left(\text{resp. } \bigwedge_{a,b\in S} \sup_M\{a, b\} = \sup_S\{a, b\}\right),$$

where the subscripts M and S indicate the set in which the infimum (resp. supremum) is taken. A subnet of a lattice is called a *sublattice* if it is both an inf-subnet and a sup-subnet. ■

Similar properties hold in complete lattices. Therefore, we give the following definition.

Definition 1.12. Let $\{M, \leq\}$ be a complete lattice and $S \subseteq M$. If $\{S, \leq\}$ is also a complete lattice, it is called a *complete subnet* of $\{M, \leq\}$. A complete subnet is called a *complete inf-subnet* (resp. a *complete sup-subnet*) if

$$\bigwedge_{A\subseteq S} \inf_M A = \inf_S A \qquad \left(\text{resp. } \bigwedge_{A\subseteq S} \sup_M A = \sup_S A\right).$$

A complete subnet is called a *complete sublattice* if it is both a complete inf-subnet and a complete sup-subnet. In this case the subscripts M and S can be omitted. ■

This definition leads directly to the following theorem and remark.

Theorem 1.13. *Let $\{M, \leq\}$ be a complete lattice and $\{S, \leq\}$ a complete subnet. Then*

$$\bigwedge_{A\subseteq S} \inf_S A \leq \inf_M A \wedge \sup_M A \leq \sup_S A. \qquad ■$$

Remark 1.14. In a complete lattice $\{M, \leq\}$, the infimum and supremum also exist for the empty set $\varnothing$. Therefore we have:

(a) The greatest element of M is also the greatest element of every complete inf-subnet, i.e., $i(M) = \inf\varnothing = i(S)$.

(b) The least element of M is also the least element of every complete sup-subnet, i.e., $o(M) = \sup\varnothing = o(S)$.

(c) In a complete sublattice S of a complete lattice M, the least and the greatest elements of M and S are identical: $o(M) = o(S) = \sup \varnothing$ und $i(M) = i(S) = \inf \varnothing$. ■

In the definition of a complete inf-subnet and a complete sup-subnet, it is presumed that S is a complete lattice. The following theorem enables us to eliminate this requirement.

Theorem 1.15. *Let $\{M, \leq\}$ be a complete lattice and $S \subseteq M$. $\{S, \leq\}$ is a complete inf-subnet (resp. a complete sup-subnet) of $\{M, \leq\}$ if and only if*

$$\bigwedge_{A \subseteq S} \inf_M A \in S \qquad \left(\textit{resp.} \bigwedge_{A \subseteq S} \sup_M A \in S \right).$$

Proof. If $\{S, \leq\}$ is a complete inf-subnet, we have $\inf_M A = \inf_S A$. Since $\inf_S A \in S$, this demonstrates the necessity of the hypothesis. We now demonstrate the sufficiency. Let $A \subseteq S$ and $I := \inf_M A \in S$. Then for all $a \in A$, $I \leq a$, i.e., I is lower bound of A in S. Moreover,

$$\bigwedge_{x \in M} \left(\bigwedge_{a \in A} x \leq a \Rightarrow x \leq I \right) \quad \Rightarrow \quad \bigwedge_{x \in S} \left(\bigwedge_{a \in A} x \leq a \Rightarrow x \leq I \right),$$

i.e., I is greatest lower bound of A in $\{S, \leq\}$. This implies that $\inf_M A = \inf_S A$. Since every subset $A \subseteq S$ has an infimum in S, Theorem 1.8 implies that $\{S, \leq\}$ is a complete lattice. ■

The following corollary is a direct consequence of Theorem 1.15.

Corollary 1.16. *Let $\{M, \leq\}$ be a complete lattice and $S \subseteq M$. $\{S, \leq\}$ is a complete sublattice of $\{M, \leq\}$ if and only if*

$$\bigwedge_{A \subseteq S} (\inf_M A \in S \wedge \sup_M A \in S). \qquad ■$$

We illustrate these concepts with two simple examples.

Examples. 1. Let $\{M, \leq\}$ be an ordered set, $a \in M$ and $S := \{s \in M \mid s \leq a\}$ (resp. $S := \{s \in M \mid a \leq s\}$). Then we have:

(a) If $\{M, \leq\}$ is a lattice, then $\{S, \leq\}$ is a sublattice.

(b) If $\{M, \leq\}$ is a complete lattice and $\overline{S} := S \cup \{i(M)\}$, then $\{\overline{S}, \leq\}$ is a complete sublattice.

Proof. (a) Let $x, y \in S$. Then $x \leq a$ und $y \leq a$ and therefore $\inf_M\{x, y\} \leq \sup_M\{x, y\} \leq a$, i.e., $\inf_M\{x, y\}$, $\sup_M\{x, y\} \in S$, which proves the assertion by Corollary 1.16.

(b) Let $\varnothing \neq A \subseteq S$. Then for all $x \in S$, $x \leq a$, and therefore $\inf_M S \leq \sup_M S \leq a$, i.e., $\inf_M S, \sup_M S \in S$. But $\varnothing \subseteq S$ also. Then in order to apply Corollary 1.16, $\inf_M \varnothing = i(M)$ and $\sup_M \varnothing = o(M)$ must be elements of $\overline{S}$. Since $o(M) \leq a$, it is automatically an element of S and therefore of $\overline{S}$. That $i(M) \in \overline{S}$, however, has been assumed explicitly. ■

2. Let $\{M, \leq\}$ be an ordered set and $S := \{s \in M \mid s \in [a, b]\}$, where $[a, b]$ denotes an interval in M. Then we have:

(a) If $\{M, \leq\}$ is a lattice, then $\{S, \leq\}$ is a sublattice.

(b) If $\{M, \leq\}$ is a complete lattice, then $\{S \cup o(M) \cup i(M), \leq\}$ is a complete sublattice.

The proofs of these statements are analogous to the proofs for the previous example.

1.3 Screens and Roundings

We will now give an abstract characterization of the essential properties of the sets and subsets displayed in Figure 1 which are essential for our description of computer arithmetic. To motivate the next definition, let us consider two simple examples from Figure 1. We recall that all sets listed in Figure 1 are ordered with respect to certain order relations.

Consider the set $IV_2\mathbb{R}$ of interval vectors of dimension 2. The elements of this set are intervals of two dimensional real vectors. Such an element describes a rectangle in the x, y plane with sides parallel to the axes. Such interval vectors are special elements of the powerset $\mathbb{P}V_2\mathbb{R}$ of two dimensional real vectors. $\mathbb{P}V_2\mathbb{R}$ is defined as the set of all subsets of two dimensional real vectors. The following relations hold between these two sets $\mathbb{P}V_2\mathbb{R}$ and $IV_2\mathbb{R}$:

(i) For each element $A \in \mathbb{P}V_2\mathbb{R}$, there exist upper bounds in $IV_2\mathbb{R}$ with respect to the order relation $\subseteq$. See Figure 1.4.

(ii) For all $A \in \mathbb{P}V_2\mathbb{R}$, the set of upper bounds in the subset $IV_2\mathbb{R}$ has a least element. See Figure 1.4, where this least element is called C.

We shall see that these two properties describe the relationship between any set in Figure 1 and its subsets as listed on its right.

In particular, consider the first row of Figure 1. For $\mathbb{R}$ and the subset S taken to be floating-point numbers, we obtain the above two properties once again:

(i) For each element $a \in \mathbb{R}$, there exist upper bounds in S with respect to the order relation $\leq$. See Figure 1.5.

(ii) For all $a \in \mathbb{R}$, the set of upper bounds in the subset S has a least element. See Figure 1.5.

In this example, corresponding properties also hold for the set of lower bounds.

These properties motivate the concept of a screen, which is formalized in the following definition.

Definition 1.17. Let $\{M, \leq\}$ be an ordered set. For each element $a \in M$, let $L(a)$ (resp. $U(a)$) be the set of lower (resp. upper) bounds of a. A subset $S \subseteq M$ is called a *screen* of M if it fulfills the properties

(S1) $\bigwedge\limits_{a\in M} L(a) \cap S \neq \varnothing \quad \wedge \quad \bigwedge\limits_{a\in M} U(a) \cap S \neq \varnothing,$

(S2) $\bigwedge\limits_{a\in M} \bigvee\limits_{x\in L(a)\cap S} \bigwedge\limits_{b\in L(a)\cap S} b \leq x \quad \wedge \quad \bigwedge\limits_{a\in M} \bigvee\limits_{y\in U(a)\cap S} \bigwedge\limits_{b\in U(a)\cap S} y \leq b,$

i.e., the set $L(a) \cap S$ has a greatest element $x = i(L(a) \cap S)$ and $U(a) \cap S$ has a least element $y = o(U(a) \cap S)$.

If only the left-hand-side properties of (S1) and (S2) hold, then S is called a *lower semiscreen*, and if only the right-hand-side properties hold, S is called an *upper semiscreen*. Usually we shall write $\{S, \leq\}$ to denote the screen or semiscreen to emphasize the ordering. ■

Screens and upper semiscreens play a central role in the description of computer arithmetic. For the sake of conciseness of expression, we shall, therefore, often speak of an upper screen instead of an upper semiscreen.

Since in a screen of an ordered set the elements $i(L(a)\cap S)$ and $o(U(a)\cap S)$ always exist, we can define mappings $\varphi : M \to S$ and $\psi : M \to S$ by

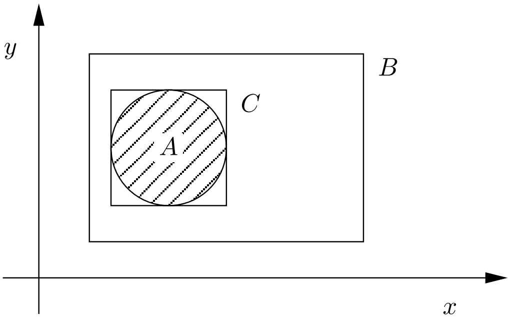


Figure 1.4. Illustration of the concept of a screen. $A \in \mathbb{P}V_2\mathbb{R}$, $B, C \in IV_2\mathbb{R}$; $A \subseteq B$, $A \subseteq C$, and for all $D \in IV_2\mathbb{R}$: $A \subseteq D \Rightarrow C \subseteq D$.

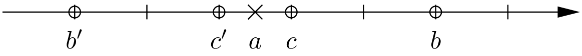


Figure 1.5. Illustration of the concept of a screen. $a \in \mathbb{R}$, $b, c \in S$; $a \leq b$, $a \leq c$, and for all $d \in S$: $a \leq d \Rightarrow c \leq d$.

(R) $\bigwedge_{a \in M} \varphi a := i(L(a) \cap S) \quad \wedge \quad \bigwedge_{a \in M} \psi a := o(U(a) \cap S).$

These mappings have the properties given in the following lemma.

Lemma 1.18. *Let $\{M, \leq\}$ be an ordered set and $\{S, \leq\}$ a screen of M. The mappings φ and ψ defined by (R) have the following properties:*

(R1) $\bigwedge_{a \in S} \varphi a = a, \qquad \bigwedge_{a \in S} \psi a = a,$

(R2) $\bigwedge_{a,b \in M} (a \leq b \Rightarrow \varphi a \leq \varphi b), \qquad \bigwedge_{a,b \in M} (a \leq b \Rightarrow \psi a \leq \psi b),$

(R3) $\bigwedge_{a \in M} \varphi a \leq a, \qquad \bigwedge_{a \in M} a \leq \psi a.$

Proof. (R1): If $a \in S$, then $a \in U(a) \cap S$, and therefore $\psi a = o(U(a) \cap S) = a$.

(R2): $a \leq b \Rightarrow U(b) \cap S \subseteq U(a) \cap S \Rightarrow \psi a = o(U(a) \cap S) \leq o(U(b) \cap S) = \psi b$.

(R3): $\psi a := o(U(a) \cap S) \in U(a) \cap S$. Therefore $a \leq \psi a$.

The proof of the properties for φ is dual. ■

If S is a lower (resp. an upper) screen of M, then only the function φ (resp. ψ) can be defined. Then the properties (R1, 2, 3) can be demonstrated only for φ (resp. ψ).

The following theorem expresses a relationship between screens and subnets.

Theorem 1.19. *A subset S of a complete lattice $\{M, \leq\}$ is an upper (resp. lower) screen of M if and only if it is a complete inf-subnet (resp. a complete sup-subnet).*

Proof. (a) First we show that (S1) and (S2) $\Rightarrow$ $\{S, \leq\}$ is a complete inf-subnet. Choosing $a = i(M)$ and using (S1), $U(a) \cap S \neq \varnothing$, we have that $i(M) = i(S)$. Because of (S2) we can define the mapping $\psi : M \to S$ (which is the subject of Lemma 1.18) with the property $\psi a = o(U(a) \cap S)$. Now let $B \subseteq S$. Since $B \subseteq M$ the hypothesis implies that the element $x := \inf_M B$ exists and we have

$$\bigwedge_{b \in B} (x \leq b \underset{\text{(R2)}}{\Rightarrow} \psi x \leq \psi b \underset{\text{(R1)}}{=} b),$$

i.e., ψx is lower bound of B. Therefore we have $\psi x \leq \inf_M B = x$. But by (R3), $x \leq \psi x$. From both inequalities we get by (O3) $x = \psi x$. Since $\psi : M \to S$, then $x = \inf_M B \in S$. Therefore by Theorem 1.15, $\{S, \leq\}$ is a complete inf-subnet of $\{M, \leq\}$, and for all $B \subseteq S$, $\inf_M B = \inf_S B$.

(b) Now we show that (S1) and (S2) hold in any complete inf-subnet $S \subseteq M$. In Remark 1.14 we observed that in a complete inf-subnet $i(M) = i(S) = \inf \varnothing$. Therefore (S1) holds.

Further, for all $A \subseteq S$ we have by hypothesis $\inf_S A = \inf_M A$.

Therefore for all $a \in M$,

$$\inf(U(a) \cap S) \in S. \tag{1.3.1}$$

Moreover, for all $b \in U(a) \cap S$, $a \leq b$, and therefore also

$$a \leq \inf(U(a) \cap S). \tag{1.3.2}$$

By (1.3.1) and (1.3.2) we get $\inf(U(a) \cap S) \in U(a) \cap S$, i.e., $\inf(U(a) \cap S)$ is the least element of $U(a) \cap S$ since it is an element of this set itself. We have, therefore, $\inf(U(a) \cap S) = o(U(a) \cap S) \in U(a) \cap S$, which completes the proof of the theorem. ■

Corollary 1.20. *A subset S of a complete lattice $\{M, \leq\}$ is a screen of M if and only if it is a complete sublattice.* ■

In an upper screen of a complete lattice, (S2) requires that for any $a \in M$ the set of upper bounds in S has a least element which itself is an upper bound of a. The example given in Figure 1.6 shows that the corresponding property for the lower bounds of a is not necessarily valid.

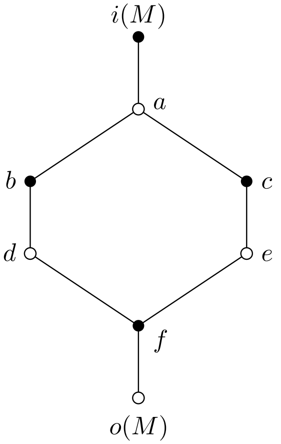


Figure 1.6. Illustration of an upper screen.

Let $\{S, \leq\}$ be the subnet of solid points in Figure 1.6, i.e., $S = \{i, b, c, f\}$. $\{S, \leq\}$ is obviously an upper screen of $\{M, \leq\}$ since $i(M) = i(S)$ and for all $A \subseteq S$, $\inf_S A = \inf_M A$.

$\{S, \leq\}$, however, is not a lower screen of M. For instance $o(M) \neq o(S) = f$ and $\sup_M\{b, c\} = a \neq \sup_S\{b, c\} = i$. By Theorem 1.19, for all $x \in M$, the set of upper bounds $U(x) \cap S$ has a least element $\inf(U(x) \cap S)$ with the property $x \leq \inf(U(x) \cap S)$. On the contrary, in general the set $L(x) \cap S$ has no greatest element nor $\sup_S(L(x) \cap S) \leq x \vee \sup_S(L(x) \cap S) \in L(x) \cap S$. In the example given in Figure 1.6, we have for $x = a$, for instance,

$$L(a) \cap S = \{b, c, f\} \quad \wedge \quad a < \sup_S(L(a) \cap S) = i(M) \notin L(a) \cap S.$$

This example shows furthermore that in an upper screen the least elements $o(M)$ and $o(S)$ are not necessarily the same.

Now let S be a complete inf-subnet (resp. a complete sup-subnet) of a complete lattice $\{M, \leq\}$. Then for all $A \subseteq S$, $\inf_M A = \inf_S A$ (resp. $\sup_M A = \sup_S A$), while the suprema (resp. the infima) may differ. See Theorem 1.13. The difference is eliminated by employing the function φ (resp. ψ) introduced in (R) (see page 25). This property is the subject of the following theorem.

Theorem 1.21. *Let $\{M, \leq\}$ be a complete lattice and $\{S, \leq\}$ a lower (resp. an upper) screen of M and $\varphi : M \to S$ (resp. $\psi : M \to S$) defined by*

(R) $\bigwedge\limits_{a \in M} \varphi a := \inf(U(a) \cap S)$ *(resp.* $\bigwedge\limits_{a \in M} \psi a := \sup(L(a) \cap S)$*).*

Then

$$\bigwedge_{A \subseteq S} \inf_S A = \varphi(\inf_M A) \qquad \textit{(resp. } \bigwedge_{A \subseteq S} \sup_S A = \psi(\sup_M A)\textit{).}$$

Proof.

$$\bigwedge_{a \in A \subseteq S} a \leq \sup_M A \underset{\text{(R2)}}{\Rightarrow} \bigwedge_{a \in A} \psi a \underset{\text{(R1)}}{=} a \underset{\text{(R2)}}{\leq} \psi(\sup_M A) \in S.$$

Therefore

$$\sup_S A \leq \psi(\sup_M A). \tag{1.3.3}$$

By Theorem 1.13, however, in a complete subnet we have generally

$$\bigwedge_{A \subseteq S} \sup_M A \leq \sup_S A \underset{\text{(R2), (R1)}}{\Rightarrow} \psi(\sup_M A) \leq \sup_S A. \tag{1.3.4}$$

Applying (O3) to (1.3.3) and (1.3.4) proves the theorem. ■

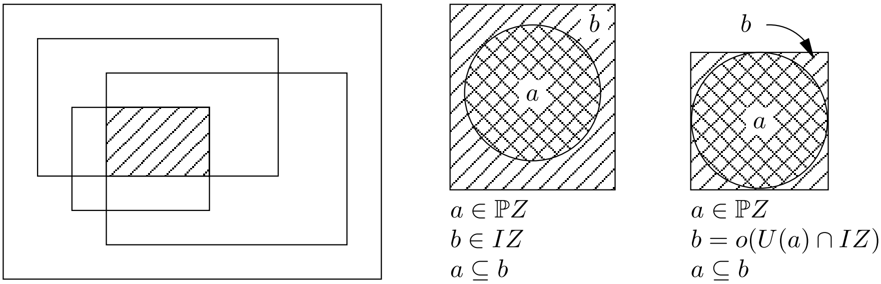
To illustrate the definition and the theorems given in this chapter, let us consider a few examples.

Examples. 1. Let Z be a bounded set of complex numbers $Z := \{\zeta := \xi + i\eta \in \mathbb{C} \mid |\xi| \leq r \wedge |\eta| \leq r\}$. The power set $\{\mathbb{P}Z, \subseteq\}$ is a complete lattice. We consider the set *IZ* of all rectangles of $\mathbb{P}Z$ with sides parallel to the axes. The elements of *IZ* are intervals in the complex plane. Also let $\varnothing \in IZ$. We show that $\{IZ, \subseteq\}$ is an upper screen of $\{\mathbb{P}Z, \subseteq\}$.

To see this, by Theorem 1.15 we have only to show that for every subset $A \subseteq IZ$, the intersection, which is the infimum in $\{\mathbb{P}Z, \subseteq\}$, is an element of *IZ*. If $A = \varnothing$, we have

$$\inf_{\mathbb{P}Z} \varnothing = \inf_{IZ} \varnothing = i(\mathbb{P}Z) = i(IZ) = Z.$$

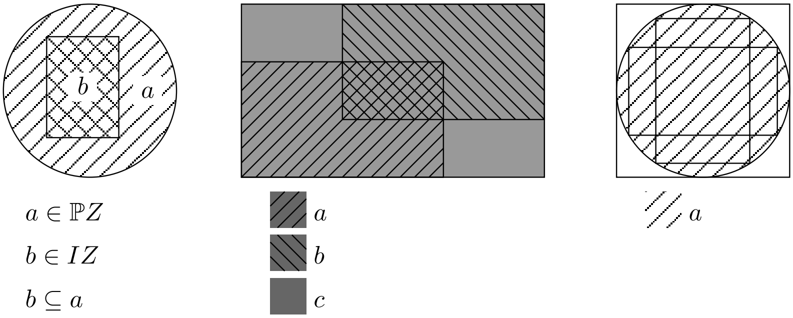
If $\varnothing \neq A \subseteq IZ$, it is easy to see by Figure 1.7(a) that the intersection of the elements of A is again an element of *IZ*. The properties (S1) and (S2) are illustrated in Figure 1.7(b) and (c).



(a) (b) (c)

Figure 1.7. Illustration of an upper screen.

Further, we have $o(\mathbb{P}Z) = o(IZ) = \varnothing$. Also the condition (S1) for a lower screen holds. See Figure 1.8(a). The necessary and sufficient criterion for a lower screen given in Theorem 1.15, however is not valid. For example, in Figure 1.8(b) we have $\sup_{\mathbb{P}Z}\{a, b\} = a \cup b \subset \sup_{IZ}\{a, b\} = c$. Therefore (S2) is not fulfilled either. In Figure 1.8(c), for instance, $\sup_{IZ}(L(a) \cap IZ) = \inf_{IZ}(U(a) \cap IZ) \supseteq a$. Figure 1.8(b) further illustrates Theorem 1.21. We have $\sup_{IZ}\{a, b\} = \psi(\sup_{\mathbb{P}Z}\{a, b\})$.



(a) (b) (c)

Figure 1.8. IZ is not a lower screen of $\mathbb{P}Z$.

2. Let X be a bounded set of real numbers $X := \{x \in \mathbb{R} \mid |x| \leq r\}$. The power set $\{\mathbb{P}X, \subseteq\}$ is a complete lattice. For all $a_1, a_2 \in X$ with $a_1 \leq a_2$ let $IX := \{[a_1, a_2] \mid a_1, a_2 \in X\}$ be the set of intervals over X. Also let $\varnothing \in IX$. Then $IX \subset \mathbb{P}X$ and $\{IX, \subseteq\}$ is an upper screen of $\{\mathbb{P}X, \subseteq\}$. This example just represents the reduction of example 1 to one dimension. All properties can be proved similarly.

3. Again let $X := \{x \in \mathbb{R} \mid |x| \leq r\}$. Then $\{X, \leq\}$ is a linearly ordered, complete lattice. If S is a finite subset of X with the property $-r \in S$ and $r \in S$, then $\{S, \leq\}$ is a complete lattice. It is easy to see by Theorem 1.15 or by Definition 1.17 and

Corollary 1.20 that $\{S, \leq\}$ is a screen of $\{X, \leq\}$. See Figure 1.9. There, for all $A \subseteq S, \inf_X A = \inf_S A = o(A) \in S$ and $\sup_X A = \sup_S A = i(A) \in S$.

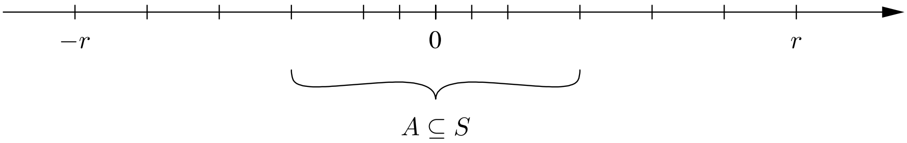


Figure 1.9. Illustration of a screen.

4. As in example 2, let IX again denote the set of intervals over X augmented by the empty set and S the subset of X, defined in example 3. Now let IS denote the set of intervals over X with endpoints in S and augmented by the empty set. In example 2 we saw that $\{IX, \subseteq\}$ is an upper screen of $\{\mathbb{P}X, \subseteq\}$. Now we show that $\{IS, \subseteq\}$ is a screen of $\{IX, \subseteq\}$. $\{IS, \subseteq\}$ is a complete lattice with the property $o(IS) = o(IX) = \varnothing$ and $i(IS) = i(IX) = X$. The infimum (resp. supremum) in IS is the intersection (resp. the interval hull) as in IX. Therefore, for every subset of intervals in IS the infimum and supremum is the same as in IX.

5. Now let Z again be a bounded set of complex numbers as defined in example 1. We have seen there that the set of intervals $\{IZ, \leq\}$ over Z is an upper screen of the power set $\{\mathbb{P}Z, \subseteq\}$. As in example 3, let S again denote a finite subset of the set X. With S we define a discrete subset N of complex numbers by

$$N := \{\zeta := \xi + i\eta \mid \xi, \eta \in S \wedge |\xi| \leq r \wedge |\eta| \leq r\}.$$

Now we consider the set of intervals IN over Z with bounds in N. Then $\{IN, \subseteq\}$ is a screen of $\{IZ, \subseteq\}$. The infimum (resp. supremum) in IN is the intersection (resp. the interval hull) as in IZ. For all subsets $A \subseteq IN$ we have $\inf_{IZ} A = \inf_{IN} A \in IN$ and $\sup_{IZ} A = \sup_{IN} A \in IN$. See Figure 1.10.

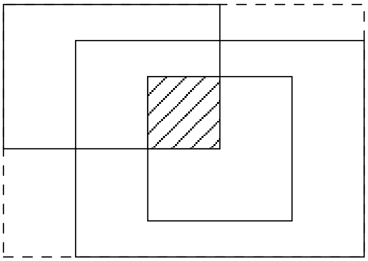


Figure 1.10. Illustration of a screen.

We shall see later that the concepts of a screen and of an upper screen encompass all essential properties of the sets listed in Figure 1. Using these concepts, we now specify the notion of rounding.

Definition 1.22. Let M be a set and $S \subseteq M$. A mapping $\square : M \to S$ is called a *rounding* if it has the property

(R1) $\bigwedge\limits_{a \in S} \square a = a$ (projection).

A rounding of a complete lattice into a lower (resp. an upper) screen (resp. a screen) is called *monotone* if

(R2) $\bigwedge\limits_{a,b \in M} (a \leq b \Rightarrow \square a \leq \square b)$ (monotone).

It is called *downwardly* (resp. *upwardly directed* if

(R3) $\bigwedge\limits_{a \in M} \square a \leq a \quad \left(\text{resp. } \bigwedge\limits_{a \in M} a \leq \square a\right)$ (directed).

■

A general property of monotone mappings, applied to roundings, is given in the following lemma.

Lemma 1.23. *Let $\{M, \leq\}$ be a complete lattice and $\{S, \leq\}$ a lower (resp. an upper) screen of M and $\square : M \to S$ a monotone rounding. Then for all $a \in S$, the set $\square^{-1} a \subseteq M$ is convex.*

Proof. Let $a_1, a_2 \in \square^{-1} a \subseteq M$. Let $b \in M$ have the property $a_1 \leq b \leq a_2$. Then by (R2), we get $\square a_1 = a \leq \square b \leq \square a_2 = a$, and by (O3), $\square b = a$, i.e., $b \in \square^{-1} a$. ■

In Definition 1.22 three properties of roundings are enumerated, and we may ask if there do indeed exist roundings with all three of these properties. A characterization of such roundings, which supplies an affirmative answer as well, is the subject of the following theorem.

Theorem 1.24. *Let $\{M, \leq\}$ be a complete lattice and $\{S, \leq\}$ a lower (resp. upper) screen. A mapping $\square : M \to S$ is a monotone downwardly (resp. upwardly) directed rounding if and only if it has the property*

(R) $\bigwedge\limits_{a \in M} \square a := i(L(a) \cap S) \quad \left(\textit{resp. } \bigwedge\limits_{a \in M} \square a := o(U(a) \cap S)\right).$

Proof. (a) It is shown in Lemma 1.18 that the mapping defined by (R) has the properties (R1), (R2), and (R3).

(b) We still have to show that the mapping $\square : M \to S$ with the properties (R1), (R2) and (R3) also fulfills (R). By (R3), $\square a \in L(a) \cap S$. Let b be any element of $L(a) \cap S$. Then $b \leq a$ and by (R1) and (R2), $\square b = b \leq \square a$, i.e., $\square a$ is the greatest element of $L(a) \cap S$. ■

Since in Theorem 1.24 $\{S, \leq\}$ is a lower screen, it is a complete sup-subnet of $\{M, \leq\}$. Then there exists the element $\sup(L(a) \cap S) \in S \subseteq M$. Since the greatest element of a set is always its supremum we also have

$$\textbf{(R)} \quad \bigwedge_{a\in M} \square a := \sup(L(a)\cap S) \qquad \left(\text{resp. } \bigwedge_{a\in M} \square a := \inf(U(a)\cap S)\right).$$

The following remark characterizes the uniqueness property expressed in Theorem 1.24.

Remark 1.25. Since the infimum and the supremum of a subset of a complete lattice are unique, there exists only one monotone downwardly (resp. upwardly) directed rounding of a complete lattice into a lower (resp. upper) screen. We shall therefore often use the special symbols $\bigtriangledown$ (resp. $\bigtriangleup$) to denote these mappings. They have the property

$$\textbf{(R)} \quad \bigwedge_{a\in M} \bigtriangledown a = \sup(L(a)\cap S) \qquad \left(\text{resp. } \bigwedge_{a\in M} \bigtriangleup a = \inf(U(a)\cap S)\right). \qquad \blacksquare$$

These observations are collected into the statement of the following corollary.

Corollary 1.26. *If $\{M,\leq\}$ is a complete lattice and $\{S,\leq\}$ a screen, then there exists exactly one monotone downwardly directed rounding $\bigtriangledown : M \to S$ and exactly one monotone upwardly directed rounding $\bigtriangleup : M \to S$. These roundings can be defined by property (R).* ■

The two directed roundings $\bigtriangledown$ and $\bigtriangleup$ are key elements in the set of all roundings. The composition $\bigtriangleup_2\bigtriangleup_1$ of two such roundings, i.e., a mapping from M into a screen $D \subset M$ followed by a mapping of D into a screen $S \subset D$, is itself a monotone upwardly directed rounding. In a linearly ordered complete lattice every monotone rounding into a screen can be expressed in terms of the two monotone directed roundings $\bigtriangledown$ and $\bigtriangleup$. These properties are made precise by the following two theorems.

Theorem 1.27. *Let $\{M,\leq\}$ be a complete lattice and let $\{S,\leq\}$ and $\{D,\leq\}$ both be lower (resp. upper) screens of $\{M,\leq\}$ with the property $S \subset D \subseteq M$. Further, let $\bigtriangledown : M \to S$, $\bigtriangledown_1 : M \to D$, $\bigtriangledown_2 : D \to S$ (resp. $\bigtriangleup : M \to S$, $\bigtriangleup_1 : M \to D$, $\bigtriangleup_2 : D \to S$) be the associated monotone downwardly (resp. upwardly) directed roundings. Then*

$$\bigwedge_{a\in M} \bigtriangledown a = \bigtriangledown_2(\bigtriangledown_1 a) \qquad \left(\textit{resp. } \bigwedge_{a\in M} \bigtriangleup a = \bigtriangleup_2(\bigtriangleup_1 a)\right).$$

Proof. $S \subset D \Rightarrow L(a)\cap S \subseteq L(a)\cap D \Rightarrow \bigtriangledown a = i(L(a)\cap S) \leq \bigtriangledown_1 a = i(L(a)\cap D)$. If we apply the mapping $\bigtriangledown_2$ to this inequality, we obtain

$$\bigtriangledown_2(\bigtriangledown a) \underset{\text{(R1)}}{=} \bigtriangledown a \underset{\text{(R2)}}{\leq} \bigtriangledown_2(\bigtriangledown_1 a). \tag{1.3.5}$$

By (R3) we have $\nabla_2(\nabla_1 a) \leq a$, and therefore

$$\nabla(\nabla_2(\nabla_1 a)) \underset{\text{(R1)}}{=} \nabla_2(\nabla_1 a) \underset{\text{(R2)}}{\leq} \nabla a. \tag{1.3.6}$$

From (1.3.5) and (1.3.6) we get $\nabla a = \nabla_2(\nabla_1 a)$ by (O3). ■

In practical applications of this theorem, S may be a floating-point system and D the set of floating-point numbers of double length on the same computer. We show however, by means of a simple example, that the statement of Theorem 1.27 does not hold generally for monotone but undirected roundings of a complete lattice into a screen.

Example 1.28. Let $M := [0,8] \subset \mathbb{R}$, $S := \{0,4,8\}$, and $D := \{0,1,2,3,4,5,6,7,8\}$. Let the roundings $\square : M \to S$, $\square_1 : M \to D$, and $\square_2 : D \to S$ be defined as mappings to the nearest element of the screen, where we additionally assume that the midpoint of two neighboring screenpoints will be mapped onto the lower neighbor in all cases. Then we get for $a = 2.4 \in M$, $\square_1 a = 2$ and $\square_2(\square_1 a) = 0 \neq \square a = 4$.

In linearly ordered complete lattices we find that in addition to the monotone directed roundings, all monotone roundings play an important role. We discuss a few of their properties beginning with the following lemma.

Lemma 1.29. *Let $\{M, \leq\}$ be a linearly ordered complete lattice and $\{S, \leq\}$ be a screen of $\{M, \leq\}$, and let $\nabla : M \to S$ and $\triangle : M \to S$ be the monotone directed roundings. Then for all $a \in M$, there exists no element $b \in S$ with the property $\nabla a < b < \triangle a$.*

Proof. We have $\nabla a = i(L(a) \cap S)$ and $\triangle a = o(U(a) \cap S)$. Suppose that the conclusion of the lemma is false. Then using (O4), we find that for any $a \in M$, there exists an element $b \in S$ such that $i(L(a) \cap S) < b \leq a \vee a \leq b < o(U(a) \cap S)$. This is a contradiction of the definition of the greatest or of the least element. ■

We use this lemma to obtain the following theorem, which characterizes monotone roundings in linearly ordered sets.

Theorem 1.30. *Let $\{M, \leq\}$ be a linearly ordered complete lattice, $\{S, \leq\}$ a screen of $\{M, \leq\}$, and $\nabla : M \to S$ (resp. $\triangle : M \to S$) the monotone downwardly (resp. upwardly) directed rounding. For each element $a \in M$ let $I := [\nabla a, \triangle a]$ and let I_1 and I_2 with $I_1 < I_2$ be subsets[2] of M which partition $I := I_1 \cup I_2$. Then the mapping $\square : M \to S$ is a monotone rounding if and only if*

$$\bigwedge_{a \in S \subseteq M} \square a = a \quad \wedge \quad \bigwedge_{a \in M \setminus S} \square a = \begin{cases} \nabla a & \textit{for all } a \in I_1 \\ \triangle a & \textit{for all } a \in I_2 \end{cases}.$$

[2] If U, V are subsets of $\{M, \leq\}$, $U < V$ means $u < v$ for all $u \in U$ and all $v \in V$.

Proof. (a) It is clear that every such mapping is a monotone rounding.

(b) We still have to show that every monotone rounding has the property stated in the theorem. Now let $I_1 := \{a \in I \mid \square a = \bigtriangledown a\}$ and let $I_2 := \{a \in I \mid \square a = \triangle a\}$. Since $\square$ is a monotone rounding I_1 and I_2 are convex sets by Lemma 1.23, and since for $a \notin S$: $\bigtriangledown a < \triangle a$, $I_1 < I_2$. Since for all $a \in I$, $\bigtriangledown a \leq a \leq \triangle a$, then (R1) and (R2) yield $\bigtriangledown a \leq \square a \leq \triangle a$. Now Lemma 1.29 implies that $I_1 \cup I_2 = I$. ■

Theorem 1.30 asserts that every monotone rounding of a linearly ordered complete lattice into a screen can be expressed by the monotone directed roundings $\triangle$ and $\bigtriangledown$. Different monotone roundings are distinguished from each other in an interval $I := [a_1, a_2]$ between neighboring screenpoints a_1 and a_2 only by the way I is split into associated subsets $I_1 < I_2$. The monotone directed roundings are extreme cases with respect to this splitting:

$$\bigtriangledown : I_1 = I \setminus \{a_2\},\ I_2 = \{a_2\} \quad \text{and} \quad \triangle : I_1 = \{a_1\},\ I_2 = I \setminus \{a_1\}.$$

Finally we show by simple examples that in general Lemma 1.29 and Theorem 1.30 are not valid in the case of a complete but not linearly ordered lattice. Let $\{M, \leq\}$ be the complete lattice that appears in Figure 1.11(a).

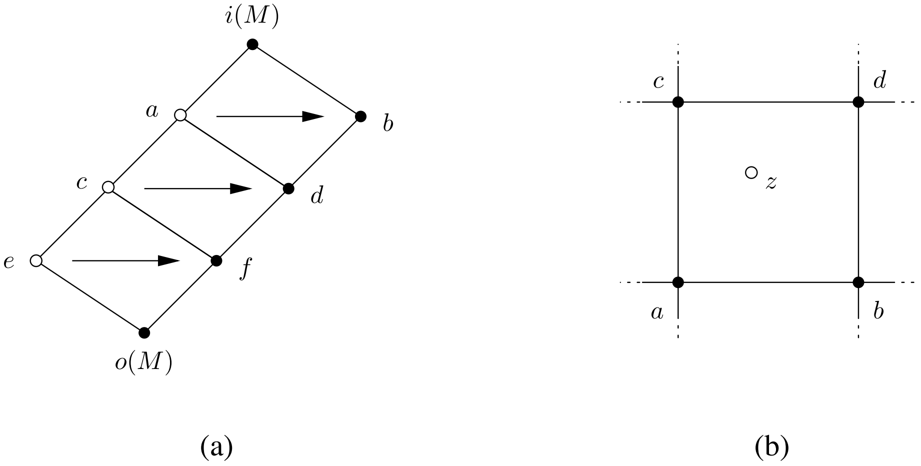


Figure 1.11. Roundings in non linearly ordered sets.

The subset $\{S, \leq\}$ consisting of the solid points in Figure 1.11(a) obviously is a screen of $\{M, \leq\}$. We define a mapping $\square : M \to S$ by the following properties:

(i) All screenpoints are fixed points of the mapping.

(ii) $\square a = b$, $\square c = d$, $\square e = f$. See Figure 1.11(a).

Then $\square$ is a monotone rounding. However, neither Lemma 1.29 nor Theorem 1.30 holds. For instance, we obtain $i(L(c) \cap S) = f < \square c = d < o(U(c) \cap S) = i(M)$, and consequently $\bigtriangledown c = f < \square c = d < b < \triangle c = i(M)$.

As another example we consider the set of complex numbers $\mathbb{C}$ with the order relation defined componentwise. $\{\mathbb{C}, \leq\}$ is a complete lattice. The subset $S \subset \mathbb{C}$ where the real and imaginary parts are restricted to integers is obviously a screen of $\mathbb{C}$. Figure 1.11(b) shows a small part of S. For all complex numbers z in the open square shown in the figure it holds that: $\bigtriangledown z = a < c < \bigtriangleup z = d$ and $\bigtriangledown z = a < b < \bigtriangleup z = d$.

1.4 Arithmetic Operations and Roundings

If M_1, M_2, and M are nonempty sets, then a mapping $\circ : M_1 \times M_2 \to M$ of the product set $M_1 \times M_2$ into M is called an *operation*. The image $\circ(a, b)$ of the operands $a \in M_1$ and $b \in M_2$ is usually written in the dyadic form $a \circ b$. If, in particular, $M_1 = M_2 = M$, one speaks of an *inner operation* in contrast to an *outer operation*, where $\circ : M_1 \times M \to M$. In the latter case M_1 is called the operator set. Examples of an inner operation are the addition and multiplication of real numbers and of an outer operation the multiplication of a vector by a scalar or the matrix-vector multiplication. A nonempty set with operations defined for its elements is called an *algebraic structure*.

The simplest algebraic structure is a set M with one inner operation $\circ$. It is called a *groupoid* $\{M, \circ\}$. An element $e \in M$ is called a *left neutral* (resp. a *right neutral*) *element* if

$$\bigwedge_{a \in M} e \circ a = a \qquad \left(\text{resp. } \bigwedge_{a \in M} a \circ e = a\right).$$

e is called a *neutral element* if it is a left as well as a right neutral element. A neutral element is always unique since the presumption of two such elements e and e' yields $e = e \circ e' = e'$.

$\{\mathbb{R}, +\}$ and $\{\mathbb{R}, \cdot\}$ are examples of groupoids with the neutral element 0 and 1 respectively. The groupoids $\{\mathbb{R}, -\}$ (resp. $\{\mathbb{R}, /\}$) have only right neutral elements 0 (resp. 1).

On computers, subtraction is not always the inverse operation for addition, nor division for multiplication. All operations have a more independent nature. It is, however, essential that all operations occurring in the spaces of Figure 1 have a right neutral element. See, for instance, Theorem 1.32 and the example following it.

An operation $\circ : M_1 \times M_2 \to M$ or $a \circ b$ is called *ordered* if for subsets $N_1 \subseteq M_1$ and $N_2 \subseteq M_2$ and fixed $(a', b') \in N_1 \times N_2$ the mappings $a \circ b'$ and $a' \circ b$ are both monotone. A nonempty set with ordered operations is called an *ordered algebraic structure*.

$\{\mathbb{R}, +, \leq\}$ and $\{\mathbb{R}^n, +, \leq\}$ are examples of ordered groups. $\{\mathbb{R}, +, \cdot, \leq\}$ is an ordered ring.

In particular, we shall call the triple $\{M, \circ, \leq\}$ an *ordered groupoid* if $\{M, \circ\}$ is a groupoid, $\{M, \leq\}$ is an ordered set, and the following condition (OA) holds:

(OA) $\bigwedge\limits_{a,b,c \in M} (a \leq b \Rightarrow a \circ c \leq b \circ c \wedge c \circ a \leq c \circ b).$

(OA) is referred to as the *compatibility property* between the algebraic and the order structure. (OA) is equivalent to

(OA') $\bigwedge\limits_{a,b,c,d \in M} (a \leq b \wedge c \leq d \Rightarrow a \circ c \leq b \circ d).$

To get (OA') from (OA), (OA) has to be applied twice while (OA') ⇒ (OA) by taking $c = d$.

A key objective of this treatise concerns the question of how operations that are defined in a certain set M can be approximated on a screen or on an upper screen S. We shall see later that natural mapping properties such as isomorphism or homomorphism for characterizing this approximation cannot be achieved. Nevertheless, we may develop reasonable and simple *compatibility properties* between the operations in S and in M. Such properties are characterized by the following definition:

Definition 1.31. Let $\{M, \leq\}$ be a complete lattice, $\{M, \circ\}$ a groupoid, and $\{S, \leq\}$ a lower (resp. an upper) screen of $\{M, \leq\}$. A groupoid $\{S, \boxed{\circ}\}$ is called a *screen groupoid* (or *rounded groupoid*) if

(RG1) $\bigwedge\limits_{a,b \in S} (a \circ b \in S \Rightarrow a \boxed{\circ} b = a \circ b).$

A screen groupoid is called *monotone* if

(RG2) $\bigwedge\limits_{a,b,c,d \in S} (a \circ b \leq c \circ d \Rightarrow a \boxed{\circ} b \leq c \boxed{\circ} d).$

A screen groupoid is called a *lower* (resp. *upper*) *screen groupoid* if

(RG3) $\bigwedge\limits_{a,b \in S} a \boxed{\circ} b \leq a \circ b \qquad \left(\text{resp. } \bigwedge\limits_{a,b \in S} a \circ b \leq a \boxed{\circ} b\right).$ ■

In Definition 1.31 three properties of screen groupoids are enumerated. We may ask if there do indeed exist groupoids with all three of these properties and how such groupoids can be produced. The following theorem supplies an answer to this question.

Theorem 1.32. *Let $\{M, \leq\}$ be a complete lattice, $\{M, \circ\}$ a groupoid, $\{S, \leq\}$ a lower (resp. upper) screen or a screen of $\{M, \leq\}$. Let $\square : M \to S$ be a mapping and let an operation $\boxed{\circ}$ in S be defined by*

(RG) $\bigwedge\limits_{a,b \in S} a \boxed{\circ} b := \square(a \circ b).$

(a) If $\square$ has the property (Ri), $i = 1, 2, 3$ defined for roundings, then $\{S, \boxed{\circ}\}$ has the property (RGi), $i = 1, 2, 3$, respectively.

(b) If the groupoid $\{M, \circ\}$ has a right neutral element e and $e \in S$, then (RG1), (RG2), and (RG3) imply (RG).

Proof. (a) We omit the proof of this property since it is straightforward.

(b) We give the proof in the case of a lower screen. Let $a, b \in S$ and $x := \bigtriangledown(a \circ b)$. Then by (R3) we obtain

$$x := x \circ e = \bigtriangledown(a \circ b) \leq a \circ b. \tag{1.4.1}$$

Since $e \in S$ we obtain

$$\bigwedge_{a \in S} a \circ e = a \in S \underset{\text{(RG1)}}{\Rightarrow} \bigwedge_{a \in S} (a \boxdot e = a \circ e = a),$$

i.e., e is also a right neutral element in $\{S, \boxdot\}$.

Applying (RG2) to (1.4.1), therefore, we get

$$x \boxdot e = x = \bigtriangledown(a \circ b) \leq a \boxdot b. \tag{1.4.2}$$

(RG3) yields $a \boxdot b \leq a \circ b$. If we apply the rounding $\bigtriangledown$ to this inequality, we get by (R1) and (R2)

$$\bigtriangledown(a \boxdot b) = a \boxdot b \leq \bigtriangledown(a \circ b). \tag{1.4.3}$$

From (1.4.2), (1.4.3) and (O3) we obtain $a \boxdot b = \bigtriangledown(a \circ b)$. ∎

Remark 1.33. The monotone directed roundings $\bigtriangledown$ and $\bigtriangleup$ are unique. Thus under the hypothesis of Theorem 1.32 there exists exactly one monotone lower (resp. upper) screen groupoid on a lower (resp. upper) screen or a screen. Because of this fact, we shall use the special signs $\mathbin{\bigtriangledown\kern-0.75em\circ}$ (resp. $\mathbin{\bigtriangleup\kern-0.75em\circ}$) for the associated operations. Theorem 1.32 implies that $\mathbin{\bigtriangledown\kern-0.75em\circ}$ (resp. $\mathbin{\bigtriangleup\kern-0.75em\circ}$) can be defined by the property

$$\text{(RG)} \quad \bigwedge_{a,b \in S} a \mathbin{\bigtriangledown\kern-0.75em\circ} b = \bigtriangledown(a \circ b) \qquad \left(\text{resp. } \bigwedge_{a,b \in S} a \mathbin{\bigtriangleup\kern-0.75em\circ} b = \bigtriangleup(a \circ b)\right).$$ ∎

This formula can now be used to construct the result of a computation in a lower (resp. upper) screen groupoid. We illustrate this by an example.

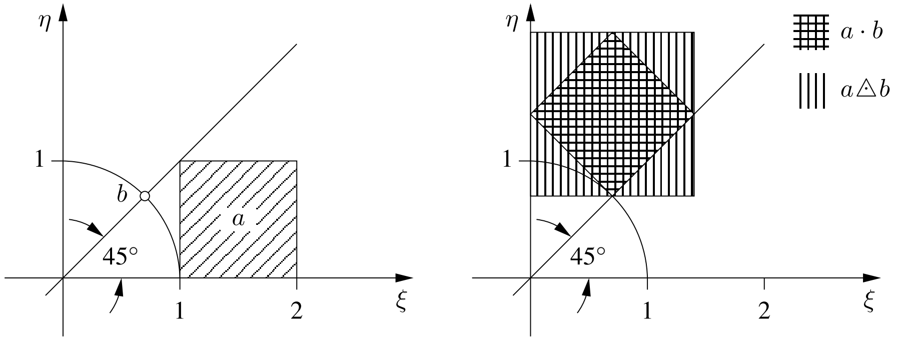
Example 1.34. Let $Z := \{\zeta := \xi + i\eta \in \mathbb{C} \mid |\xi| \leq r \wedge |\eta| \leq r\}$, $M := \mathbb{P}Z$ and $S \subseteq M$ the set of all rectangles of $\mathbb{P}Z$ with sides parallel to the axes. We already know from example 1 in Section 1.3 that $\{S, \subseteq\}$ is an upper screen of $\{M, \subseteq\}$. Now we define a multiplication in M by

$$\bigwedge_{a,b \in M} a \cdot b := \{\alpha \cdot \beta \mid \alpha, \beta \in Z \wedge \alpha \in a \wedge \beta \in b\}.$$

In order not to leave the set Z while executing the product, we replace the real and/or imaginary part of the product $\alpha \cdot \beta$ by r whenever the former and/or the latter do in fact exceed r. Then $\{M, \cdot\}$ is a groupoid with the neutral element $\{1\}$. Now set $r = 5$ and consider two special elements $a, b \in M$. See Figure 1.12(a).

$$a := \{\zeta := \xi + i\eta \in \mathbb{C} \mid \xi \in [1,2] \wedge \eta \in [0,1]\},$$
$$b := \{e^{i\frac{\pi}{4}}\} = \{\frac{1}{2}\sqrt{2}(1+i)\}.$$

Here a, b are elements of $S \subset M$. Figure 1.12(b) shows the product $a \cdot b$ as well as the construction of the product $a \vartriangle b$ in the upper screen groupoid using the formula $a \vartriangle b = \vartriangle(a \cdot b)$.



(a) (b)

Figure 1.12. Operation in an upper screen groupoid.

When the groupoids are ordered, additional properties can be derived for them. If $\{M, \circ, \leq\}$ is an ordered groupoid, then every monotone screen groupoid $\{S, \boxdot, \leq\}$ is also an ordered groupoid. This follows directly from (OA') and (RG2):

$$\bigwedge_{a,b,c,d \in S} (a \leq b \wedge c \leq d \underset{\text{(OA')}}{\Rightarrow} a \circ c \leq b \circ d \underset{\text{(RG2)}}{\Rightarrow} a \boxdot c \leq b \boxdot d), \tag{1.4.4}$$

i.e., (OA) and (OA') hold in $\{S, \boxdot, \leq\}$ also.

We have already observed that $\{\mathbb{R}, +, \leq\}$ is an ordered groupoid. If $S \subset \mathbb{R}$, for instance, denotes a set of floating-point numbers, we can conclude from (1.4.4) that $\{S, \triangledown\!\!\!\!+, \leq\}$ and $\{S, \vartriangle\!\!\!\!+, \leq\}$ are ordered groupoids.

The developments of this section already have many applications. They can be used, for instance, to compute lower and upper bounds of a finite sum

$$s := \sum_{i=1}^{n} s_i, \quad \text{with } s_i \in S,\ i = 1(1)n,$$

on the computer. Here again, let $S \subset \mathbb{R}$ denote a set of floating-point numbers. We obtain successively:

$$s_1 \mathbin{\bigtriangledown\!\!\!\!\!+} s_2 \underset{\text{(RG)}}{:=} \bigtriangledown(s_1 + s_2) \underset{\text{(R3)}}{\leq} s_1 + s_2, \tag{1.4.5}$$

$$\begin{aligned} s_1 \mathbin{\bigtriangledown\!\!\!\!\!+} s_2 \mathbin{\bigtriangledown\!\!\!\!\!+} s_3 &\underset{\text{(RG)}}{:=} \bigtriangledown((s_1 \mathbin{\bigtriangledown\!\!\!\!\!+} s_2) + s_3) \underset{\text{(R3)}}{\leq} s_1 \mathbin{\bigtriangledown\!\!\!\!\!+} s_2 + s_3 \\ &\underset{\text{(1.4.5), (OA)}_{\mathbb{R}}}{\leq} s_1 + s_2 + s_3. \end{aligned} \tag{1.4.6}$$

Continuing this way leads to:

$$s_1 \mathbin{\bigtriangledown\!\!\!\!\!+} s_2 \mathbin{\bigtriangledown\!\!\!\!\!+} s_3 \mathbin{\bigtriangledown\!\!\!\!\!+} \cdots \mathbin{\bigtriangledown\!\!\!\!\!+} s_n \leq s = \sum_{i=1}^{n} s_i \leq s_1 \mathbin{\bigtriangleup\!\!\!\!\!+} s_2 \mathbin{\bigtriangleup\!\!\!\!\!+} s_3 \mathbin{\bigtriangleup\!\!\!\!\!+} \cdots \mathbin{\bigtriangleup\!\!\!\!\!+} s_n. \tag{1.4.7}$$

If only lower bounds $s_i^{(l)}$ and upper bounds $s_i^{(u)}$ of the summands are available: $s_i^{(l)} \leq s_i \leq s_i^{(u)}$, these bounds can be used to obtain lower and upper bounds of the sum s:

$$\begin{aligned} s_1^{(l)} \mathbin{\bigtriangledown\!\!\!\!\!+} s_2^{(l)} \mathbin{\bigtriangledown\!\!\!\!\!+} \cdots \mathbin{\bigtriangledown\!\!\!\!\!+} s_n^{(l)} &\underset{\text{(1.4.7)}}{\leq} \sum_{i=1}^{n} s_i^{(l)} \underset{\text{(OA)}_{\mathbb{R}}}{\leq} \sum_{i=1}^{n} s_i \underset{\text{(OA)}_{\mathbb{R}}}{\leq} \sum_{i=1}^{n} s_i^{(u)} \\ &\underset{\text{(1.4.7)}}{\leq} s_1^{(u)} \mathbin{\bigtriangleup\!\!\!\!\!+} s_2^{(u)} \mathbin{\bigtriangleup\!\!\!\!\!+} \cdots \mathbin{\bigtriangleup\!\!\!\!\!+} s_n^{(u)}. \end{aligned} \tag{1.4.8}$$

Equation (1.4.8) can be used, for instance, to compute lower and upper bounds of a scalar product

$$\sum_{i=1}^{n} a_i \cdot b_i, \quad a_i, b_i \in S$$

on the computer. In this case the bounds $s_i^{(l)}$ and $s_i^{(u)}$ for the summands $s_i := a_i \cdot b_i$ in (1.4.8) are obtained by

$$s_i^{(l)} := a_i \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_i \underset{\text{(RG)}}{:=} \bigtriangledown(a_i \cdot b_i) \leq s_i \leq s_i^{(u)} := a_i \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_i \underset{\text{(RG)}}{:=} \bigtriangleup(a_i \cdot b_i).$$

We stress the fact that when computing bounds for a scalar product this way the lower (resp. upper) bound of the sum is obtained by performing all arithmetic operations with rounding downwards (resp. upwards). Since changing the rounding mode is a slow process on many computers, the described methods are relatively fast. We mention, however, that the computed bounds are not optimal. We shall see later in this treatise that **most accurate** bounds for scalar products can be computed much faster by fixed-point accumulation and by making use of pipelining in a very natural way.

In a similar manner bounds for sums of other expressions can be computed by (1.4.8) if bounds for the summands are available. Sums of quotients (Taylor series) might be an example.

Additional properties of ordered groupoids are given by the following theorem:

Theorem 1.35. *Let $\{M, \circ, \leq\}$ be a completely ordered groupoid with a right neutral element $e \in M$ and $\{S, \mathbin{\bigtriangledown\!\!\!\!\!\!\circ}, \leq\}$ the monotone lower screen groupoid (resp. $\{S, \mathbin{\bigtriangleup\!\!\!\!\!\!\circ}, \leq\}$ the monotone upper screen groupoid) with $e \in S$. Then for all $a, b \in M$ the following inequalities hold:*

$$(\bigtriangledown a) \mathbin{\bigtriangledown\!\!\!\!\!\!\circ} (\bigtriangledown b) \leq \bigtriangledown(a \circ b) \leq a \circ b$$
$$\left(\textit{resp. } a \circ b \leq \bigtriangleup(a \circ b) \leq (\bigtriangleup a) \mathbin{\bigtriangleup\!\!\!\!\!\!\circ} (\bigtriangleup b)\right).$$

Proof. $\bigtriangledown a \leq a \wedge \bigtriangledown b \leq b \underset{\text{(OA')}}{\Rightarrow} (\bigtriangledown a) \circ (\bigtriangledown b) \leq a \circ b \underset{\text{(R2)}}{\Rightarrow} \bigtriangledown((\bigtriangledown a) \circ (\bigtriangledown b)) \underset{\text{(RG)}}{=} (\bigtriangledown a) \mathbin{\bigtriangledown\!\!\!\!\!\!\circ} (\bigtriangledown b) \leq \bigtriangledown(a \circ b) \underset{\text{(R3)}}{\leq} a \circ b.$ ■

The inequalities $(\bigtriangledown a) \mathbin{\bigtriangledown\!\!\!\!\!\!\circ} (\bigtriangledown b) \leq \bigtriangledown(a \circ b)$ (resp. $\bigtriangleup(a \circ b) \leq (\bigtriangleup a) \mathbin{\bigtriangleup\!\!\!\!\!\!\circ} (\bigtriangleup b)$) of this theorem assert in general that the monotone lower (resp. upper) screen groupoid is not a homomorphic image of the ordered groupoid $\{M, \circ, \leq\}$. In concrete cases such as in interval arithmetic, it is easy to show by simple examples that the equality sign is not valid in general.

The outer inequality of Theorem 1.35

$$(\bigtriangledown a) \mathbin{\bigtriangledown\!\!\!\!\!\!\circ} (\bigtriangledown b) \leq a \circ b \qquad \left(\text{resp. } a \circ b \leq (\bigtriangleup a) \mathbin{\bigtriangleup\!\!\!\!\!\!\circ} (\bigtriangleup b)\right)$$

asserts that the computation on the screen is always on one side of the computation in the groupoid $\{M, \circ, \leq\}$. This property remains valid even in expressions containing many operations.

Moreover, if several different operations are defined in M and if they are all approximated in S by lower (resp. upper) screen groupoids, then the previous assertion is also valid for expressions containing several different operations. In particular, this is the case in interval arithmetic, where the calculations are done on an upper screen with inclusion as an order relation.

Properties of comparison of computations in different lower (resp. upper) screen groupoids are derived in the following theorem.

Theorem 1.36. *Let $\{M, \circ, \leq\}$ be a completely ordered groupoid with the right neutral element e. Let S and D be lower (resp. upper) screens of $\{M, \leq\}$ with the property $e \in S \subseteq D \subseteq M$ and $\bigtriangledown : M \to S$, $\bigtriangledown_1 : M \to D$, $\bigtriangledown_2 : D \to S$ the monotone downwardly directed roundings (resp. $\bigtriangleup : M \to S$, $\bigtriangleup_1 : M \to D$, $\bigtriangleup_2 : D \to S$ the monotone upwardly directed roundings) and $\{S, \mathbin{\bigtriangledown\!\!\!\!\!\!\circ}, \leq\}$, $\{D, \mathbin{\bigtriangledown\!\!\!\!\!\!\circ}_1, \leq\}$,*

$\{S, \mathbin{\overset{\circ}{\triangledown}}_2, \leq\}$ *(resp.* $\{S, \mathbin{\overset{\circ}{\vartriangle}}, \leq\}$*,* $\{D, \mathbin{\overset{\circ}{\vartriangle}}_1, \leq\}$*,* $\{S, \mathbin{\overset{\circ}{\vartriangle}}_2, \leq\}$*) the corresponding monotone lower (resp. upper) screen groupoids. Then*

(a) $\{S, \mathbin{\overset{\circ}{\triangledown}}, \leq\} = \{S, \mathbin{\overset{\circ}{\triangledown}}_2, \leq\}$
$($*resp.* $\{S, \mathbin{\overset{\circ}{\vartriangle}}, \leq\} = \{S, \mathbin{\overset{\circ}{\vartriangle}}_2, \leq\})$,

(b) $\bigwedge_{a,b\in M} (\triangledown a)\mathbin{\overset{\circ}{\triangledown}}(\triangledown b) \leq (\triangledown_1 a)\mathbin{\overset{\circ}{\triangledown}}_1(\triangledown_1 b) \leq a \circ b$
$\left(\textit{resp.} \bigwedge_{a,b\in M} a \circ b \leq (\vartriangle_1 a)\mathbin{\overset{\circ}{\vartriangle}}_1(\vartriangle_1 b) \leq (\vartriangle a)\mathbin{\overset{\circ}{\vartriangle}}(\vartriangle b)\right)$,

i.e., operations in a finer monotone lower (resp. upper) screen groupoid always lead to nondegraded results.

Proof. (a) By Theorem 1.32 we have for all $a, b \in S : a\mathbin{\overset{\circ}{\triangledown}}_2 b = \triangledown_2(a\mathbin{\overset{\circ}{\triangledown}}_1 b)$ and therefore by Theorem 1.27:

$$a\mathbin{\overset{\circ}{\triangledown}}b = \triangledown(a \circ b) = \triangledown_2(\triangledown_1(a \circ b)) = \triangledown_2(a\mathbin{\overset{\circ}{\triangledown}}_1 b) = a\mathbin{\overset{\circ}{\triangledown}}_2 b.$$

(b) We have

$$\begin{aligned}
&\triangledown_2(\triangledown_1 a) \leq \triangledown_1 a \wedge \triangledown_2(\triangledown_1 b) \leq \triangledown_1 b \\
&\qquad \underset{\text{(OA')}}{\Rightarrow} \triangledown_2(\triangledown_1 a) \circ \triangledown_2(\triangledown_1 b) \leq (\triangledown_1 a) \circ (\triangledown_1 b) \\
&\qquad \underset{\text{(RG2)}}{\Rightarrow} \triangledown_2(\triangledown_1 a)\mathbin{\overset{\circ}{\triangledown}}\triangledown_2(\triangledown_1 b) \leq (\triangledown_1 a)\mathbin{\overset{\circ}{\triangledown}}(\triangledown_1 b).
\end{aligned}$$ ■

Some of the concepts and results of this section will also be needed in the case of outer operations. To have them available, we state the following Definition 1.37 and Theorem 1.38.

Definition 1.37. Let $\{M, \leq\}$ be a complete lattice and $\{S, \leq\}$ a lower (resp. an upper) screen of $\{M, \leq\}$. Further, let N be an operator set of M with an outer operation $\circ : N \times M \to N$. Let $T \subseteq N$. An operation $\boxdot : T \times S \to S$ is called an *outer screen operation* if

(RG1) $\bigwedge_{\alpha\in T} \bigwedge_{a\in S} (\alpha \circ a \in S \Rightarrow \alpha \boxdot a = \alpha \circ a).$

An outer screen operation is called *monotone* if

(RG2) $\bigwedge_{\alpha,\beta\in T} \bigwedge_{a,b\in S} (\alpha \circ a \leq \beta \circ b \Rightarrow \alpha \boxdot a \leq \beta \boxdot b).$

An outer screen operation is called *lower* (resp. *upper screen operation* if

(RG3) $\bigwedge_{\alpha\in T} \bigwedge_{a\in S} \alpha \boxdot a \leq \alpha \circ a \qquad \left(\text{resp.} \bigwedge_{\alpha\in T} \bigwedge_{a\in S} \alpha \circ a \leq \alpha \boxdot a\right).$ ■

Theorem 1.38. *Let $\{M, \leq\}$ be a complete lattice, $\{S, \leq\}$ a lower (resp. upper) screen or a screen of $\{M, \leq\}$, and $\square : M \to S$ a mapping. Further, let N be an operator set of M with an operation $\circ : N \times M \to M$. Let $T \subseteq N$, and let an operation $\boxdot : T \times S \to S$ be defined by*

(RG) $\bigwedge\limits_{\alpha \in T} \bigwedge\limits_{a \in S} \alpha \boxdot a := \square(\alpha \circ a).$

(a) If $\square$ has the property (Ri), $i = 1, 2, 3$, defined for roundings, then the operation $\boxdot$ has the property (RGi), $i = 1, 2, 3$, respectively.

(b) If there exists an identity operator $\epsilon \in N$ with the property

$$\bigwedge_{a \in N} \epsilon \circ a = a,$$

and if $\epsilon \in T \subseteq N$, then (RG1), (RG2) and (RG3) imply (RG). ■

We omit the proof of this theorem since it is completely analogous to that of Theorem 1.32. Since the monotone directed roundings are unique, there exists exactly one monotone lower (resp. upper) screen operation, which we therefore denote by $\mathbin{\bigtriangledown\mkern-14mu\circ}$ (resp. $\mathbin{\bigtriangleup\mkern-14mu\circ}$). By Theorem 1.38 these operations can also be defined by the property

(RG) $\bigwedge\limits_{\alpha \in T} \bigwedge\limits_{a \in S} \alpha \mathbin{\bigtriangledown\mkern-14mu\circ} a := \triangledown(\alpha \circ a) \qquad \left(\text{resp. } \bigwedge\limits_{\alpha \in T} \bigwedge\limits_{a \in S} \alpha \mathbin{\bigtriangleup\mkern-14mu\circ} a := \triangle(\alpha \circ a)\right).$

Chapter 2
Ringoids and Vectoids

In this chapter we shall develop the concepts of a ringoid and a vectoid as well as those of weakly ordered or ordered or inclusion-isotonally ordered ringoids and vectoids. The development will be self-contained and, in particular, independent of Chapter 1. Chapters 3 and 4 will bring the contents of the first two chapters together.

This chapter begins with the definition of a ringoid and the derivation of its most important properties. Then we show that the power set of a ringoid is a ringoid, that the matrices over a weakly ordered or an ordered ringoid form a weakly ordered or an ordered ringoid, and that the complexification of a weakly ordered ringoid also leads to a weakly ordered ringoid. Then we define the concept of a vectoid and derive its properties. The power set of a vectoid turns out to be a vectoid. The n-tuples over a ringoid R form a vectoid over R, as do the matrices over R. Finally we show that ringoids and vectoids also occur in sets of mappings into a ringoid R.

2.1 Ringoids

We begin with the following definition of a ringoid.

Definition 2.1. A nonempty set R in which an addition $(+)$ and a multiplication[1] $(\cdot)$ are defined is called a *ringoid* if the following properties hold:

(D1) $\bigwedge_{a,b \in R} a + b = b + a,$

(D2) $\bigvee_{o \in R} \bigwedge_{a \in R} a + o = a,$

(D3) $\bigvee_{e \in R \setminus \{o\}} \bigwedge_{a \in R} a \cdot e = e \cdot a = a,$

(D4) $\bigwedge_{a \in R} a \cdot o = o \cdot a = o.$

(D5) There exists an element $x \in R$ such that

(a) $e + x = o,$

[1] We often write ab instead of $a \cdot b$. We adopt the convention that multiplication and division are to be executed before addition and subtraction. However, the usual priorities dictated by parentheses are assumed. Several operations of the same priority are to be executed from left to right, i.e., $a_1 \circ a_2 \circ \ldots \circ a_{n-1} \circ a_n := (a_1 \circ a_2 \circ \ldots \circ a_{n-1}) \circ a_n$, for $n \geq 3$.

(b) $x \cdot x = e$,

(c) $\bigwedge_{a,b \in R} x(a+b) = (xa) + (xb)$,

(d) $\bigwedge_{a,b \in R} x(ab) = (xa)b = a(xb)$.

(D6) x is unique.

A ringoid R is called *division ringoid* if a division $/ : R \times R \backslash N \to R$ is defined with respect to which properties (D7,8,9) hold. Here $N \subseteq R$ is some subset of R which contains o.

(D7) $\bigwedge_{a \in R} a/e = a$,

(D8) $\bigwedge_{a \in R \backslash N} o/a = o$,

(D9) $\bigwedge_{a \in R} \bigwedge_{b \in R \backslash N} x(a/b) = (xa)/b = a/(xb)$.

A ringoid or a division ringoid is called *weakly ordered* if $\{R, \leq\}$ is an ordered set and

(OD1) $\bigwedge_{a,b,c \in R} (a \leq b \Rightarrow a + c \leq b + c)$,

(OD2) $\bigwedge_{a,b \in R} (a \leq b \Rightarrow xb \leq xa)$.

A weakly ordered ringoid is called *ordered* if

(OD3) $\bigwedge_{a,b,c \in R} (o \leq a \leq b \wedge c \geq o \Rightarrow ac \leq bc \wedge ca \leq cb)$.

A weakly ordered division ringoid is called *ordered* if

(OD4) (a) $\bigwedge_{a,b \in R} \bigwedge_{c \in R \backslash N} (o \leq a \leq b \wedge c > o \Rightarrow o \leq a/c \leq b/c)$,

(b) $\bigwedge_{a,b \in R \backslash N} \bigwedge_{c \in R} (o < a \leq b \wedge c \geq o \Rightarrow c/a \geq c/b \geq o)$.

A ringoid is called *inclusion-isotonally ordered* with respect to an order relation $\{R, \subseteq\}$ if for all operations $\circ \in \{+, \cdot\}$

(OD5) $\bigwedge_{a,b,c,d \in R} (a \subseteq b \wedge c \subseteq d \Rightarrow a \circ c \subseteq b \circ d)$.

A division ringoid is called *inclusion-isotonally ordered* if

(OD6) $\bigwedge_{a,b \in R} \bigwedge_{c,d \in R \backslash N} (a \subseteq b \wedge c \subseteq d \Rightarrow a/c \subseteq b/d)$. ■

A ringoid as well as a division ringoid is just a set of elements in which three special elements o, e, and x exist and for which the rules concerning the operations $+$, $\cdot$, $/$ are given by Definition 2.1. (D7) and (D9) imply that $e \notin N$ and that for all $a \notin N$, $xa \notin N$ as well. A ringoid may be weakly ordered or ordered with respect to one order relation as well as inclusion-isotonally ordered with respect to another order relation. The third special element x in a ringoid has many properties that in the real or complex number field in particular distinguish the element -1. This motivates the following definition.

Definition 2.2. In a ringoid $\{R, +, \cdot\}$ we define a minus operator by

$$\bigwedge_{a \in R} -a := xa, \tag{2.1.1}$$

and a subtraction by

$$\bigwedge_{a,b \in R} a - b := a + (-b). \qquad \blacksquare$$

Setting $a = e$ in (2.1.1) and using (D3) gives

$$x = -e. \tag{2.1.2}$$

Using (2.1.2), several of the rules of Definition 2.1 can be written in a simpler form:

(D5) There exists an element $-e \in R$ such that

(a) $e + (-e) = o$,

(b) $(-e) \cdot (-e) = e$,

(c) $\bigwedge_{a,b \in R} -(a + b) = (-a) + (-b)$,

(d) $\bigwedge_{a,b \in R} -(ab) = (-a)b = a(-b)$.

(D9) $\bigwedge_{a \in R} \bigwedge_{b \in R \setminus N} -(a/b) = (-a)/b = a/(-b)$.

(OD2) $\bigwedge_{a,b \in R} (a \leq b \Rightarrow -b \leq -a)$,

Every ring with a unit element is also a ringoid. Thus a ringoid represents a certain generalization of a ring with a unit element. In order to see this, we recall that a ring with a unit element is an additive group with an associative multiplication with respect to which a neutral element exists. The two distributive laws $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$ are also valid. From these properties the rules (D1), (D2), (D3), (D5a), (D5c), (D5d) and (D6) of a ringoid follow immediately. The proofs of (D4) and (D5b) run as follows:

(D4): $a \cdot o = a \cdot ((-e) + e) = -a + a = o = ((-e) + e) \cdot a = o \cdot a$,

(D5b): $(-e) \cdot o = (-e)((-e) + e) = (-e)(-e) + (-e) = o \Rightarrow (-e)(-e) = e$.

Here the last implication follows from the uniqueness of the additive inverse of $-e$ in the ring.

In a ringoid many familiar properties of a ring with a unit element still hold. Some of them are summarized in the following theorem.

Theorem 2.3. *Let $\{R, +, \cdot\}$ be a ringoid with the neutral elements o and e. Then for all $a, b, c, d \in R$ the following properties hold:*

(a) o and e are unique, and $-e \neq o$,

(b) $o - a = -a = (-e) \cdot a = a \cdot (-e)$,

(c) $-(-a) = a$,

(d) $-(a - b) = -a + b = b - a$,

(e) $(-a)(-b) = a \cdot b$.

(f) $-e$ *is the unique solution of the equation* $(-e) \cdot z = e$.

(g) $a \cdot e = o$ *implies* $a = o$ *and* $-a = o$ *implies* $a = o$.

(h) $a - z = a \Rightarrow z = o$, *i.e., o is the only right neutral element of subtraction.*

In a division ringoid $\{R, N, +, \cdot, /\}$ *the following additional properties hold:*

(i) $(-a)/(-b) = a/b$,

(j) $(-e)/(-e) = e$.

(k) *If for all a,* $a/a = e$, *then e is the only right neutral element of division.*

In a weakly ordered ringoid $\{R, +, \cdot, \leq\}$ *we have*

(l) $-e \neq e$,

(m) $a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d$,

(n) $a < b \Rightarrow -b < -a$.

In an ordered ringoid we have moreover

(o) $o \leq a \leq b \wedge o \leq c \leq d \Rightarrow o \leq ac \leq bd \wedge o \leq ca \leq db$,

(p) $a \leq b \leq o \wedge c \leq d \leq o \Rightarrow o \leq bd \leq ac \wedge o \leq db \leq ca$,

(q) $a \leq b \leq o \wedge o \leq c \leq d \Rightarrow ad \leq bc \leq o \wedge da \leq cb \leq o$.

In a linearly ordered ringoid $\{R, +, \cdot, \leq\}$ *we have further*

(r) $a \leq b \wedge c \geq o \Rightarrow ac \leq bc \wedge ca \leq cb$,

(s) $a \leq b \wedge c \leq o \Rightarrow ac \geq bc \wedge ca \geq cb$,

(t) $a + (-a) = o$, *i.e.,* $-a$ *is an additive inverse of a,*

(u) $e > o \wedge -e < o$.

In an ordered division ringoid $\{R, N, +, \cdot, /, \leq\}$ *we have*

(v) $a > o \wedge b > o \Rightarrow a/b \geq o$,

(w) $a < o \wedge b > o \Rightarrow a/b \leq o \wedge b/a \leq o$,

(x) $a < o \wedge b < o \Rightarrow a/b \geq o$.

In a completely and weakly ordered ringoid we have

(y) $\bigwedge_{\emptyset \neq A \in \mathbb{P}R} (\inf A = -\sup(-A) \wedge \sup A = -\inf(-A))$.

In an inclusion-isotonally ordered ringoid $\{R, +, \cdot, \subseteq\}$, *(OD5) is also valid with subtraction*

(z) $\bigwedge_{a,b,c,d \in R} (a \subseteq b \wedge c \subseteq d \Rightarrow a - c \subseteq b - d)$.

Proof. The proof of most of the properties is straightforward. Thus we give a concise sketch of each such proof. This procedure will be typical in this chapter.

(a) A neutral element is unique. $(-e)(-e) = e \underset{\text{(D4)}}{\Rightarrow} -e \neq o$.

(b) $o - a = o + (-a) = -a = (-e)a \underset{\text{(D5d)}}{=} a(-e)$.

(c) $-(-a) = (-e)((-e)a) \underset{\text{(D5d)}}{=} ((-e)(-e))a \underset{\text{(D5b)}}{=} a$.

(d) $-(a-b) = (-e)(a+(-e)b) \underset{\text{(D5c)}}{=} (-e)a + (-e)((-e)b) \underset{\text{(c)}}{=} -a + b \underset{\text{(D1)}}{=} b - a$.

(e) $(-a)(-b) = ((-e)a)((-e)b) \underset{\text{(D5d)}}{=} (-e)((-e)a)b \underset{\text{(c)}}{=} ab$.

(f) Assume that a solves $(-e)z = e$. Then $-e = (-e) \cdot e = (-e)((-e)a) \underset{\text{(c)}}{=} a$.

(g) $e \cdot a = o \Rightarrow a = o$ by (D3). $-a = (-e)a = o \underset{\text{(D4)}}{\Rightarrow} (-e)((-e)a) = o \underset{\text{(c)}}{=} a = o$.

(h) $a - o = a + (-o) \underset{\text{(D4)}}{=} a + o \underset{\text{(D2)}}{=} a$, i.e., o is a right neutral element of subtraction. Assume o' is another one. Then for all $a \in R$, $a - o' = a + (-o') = a \underset{\text{(a)}}{\Rightarrow} -o' = o$ $\underset{\text{(g)}}{\Rightarrow} o' = o$.

(i) $(-a)/(-b) = ((-e)a)/((-e)b) \underset{\text{(D9)}}{=} ((-e)((-e)a))/b \underset{\text{(c)}}{=} a/b$.

(j) $(-e)/(-e) \underset{\text{(i)}}{=} e/e \underset{\text{(D7)}}{=} e$.

(k) Assume that e and e' are right neutral elements. Then $e = e'/e' = e'$.

(l) Assume that $-e = e$. Then $a \leq b \Rightarrow -a \leq -b$ in contradiction to (OD2).

(m) $a \leq b \wedge c \leq d \Rightarrow a + c \leq b + c \leq b + d$.

(n) $a < b :\Leftrightarrow a \leq b \wedge a \neq b \underset{\text{(OD2)}}{\Rightarrow} -b \leq -a \wedge -b \neq -a \Leftrightarrow: -b < -a$.

(o) $o \leq a \leq b \wedge o \leq c \leq d \underset{\text{(OD3)}}{\Rightarrow} o \leq ac \leq bc \leq bd$.

(p) $a \leq b \leq o \wedge c \leq d \leq o$
$\Rightarrow o \leq -b \leq -a \wedge o \leq -d \leq -c \underset{\text{(o)}}{\Rightarrow} o \leq (-b)(-d) \leq (-a)(-c) \underset{\text{(e)}}{\Rightarrow} o \leq bd \leq ac$.

(q) $a \leq b \leq o \wedge o \leq c \leq d$
$\Rightarrow o \leq -b \leq -a \wedge o \leq c \leq d \underset{\text{(o)}}{\Rightarrow} o \leq (-b)c \leq (-a)d \Rightarrow ad \leq bc \leq o$.

(r) (1) $o \leq a \leq b \wedge c \geq o \underset{\text{(OD3)}}{\Rightarrow} o \leq ac \leq bc$,

(2) $a \leq b \leq o \wedge c \geq o \underset{\text{(q)}}{\Rightarrow} ac \leq bc \leq o$,

(3) $a \leq o \leq b \wedge c \geq o \Rightarrow ac \leq o \leq bc$.

(s) $a \leq b \wedge c \leq o \Rightarrow a \leq b \wedge o \leq -c \underset{\text{(r)}}{\Rightarrow} a(-c) \leq b(-c) \underset{\text{(OD2)}}{\Rightarrow} bc \leq ac$.

(t) $b := a + (-a) \Rightarrow -b = (-e)(a + (-a)) \underset{(\text{D5c}),(\text{D2})}{=} a + (-a) = b.$
$(\text{O4}) \Rightarrow b \geq o \vee b \leq o \underset{(\text{OD2})}{\Rightarrow} -b = b \leq o \vee -b = b \geq o \underset{(\text{O3})}{\Rightarrow} b = o.$

(u) $(\text{D3}) \Rightarrow e \neq o \underset{(\text{O4})}{\Rightarrow} e > o \vee e < o$. Assume $e < o \underset{(\text{p})}{\Rightarrow} e \cdot e = e \geq o$, i.e., $e > o$, which is a contradiction $\Rightarrow e > o \Rightarrow -e < o.$

(v) (OD4).

(w) $a < o \wedge b > o \underset{(\text{n})}{\Rightarrow} -a > o \wedge b > o \underset{(\text{v})}{\Rightarrow} (-a)/b \geq o \underset{(\text{OD2})}{\Rightarrow} (-a)/(-b) \underset{(\text{i})}{=} a/b \leq o.$

(x) $a < o \wedge b < o \underset{(\text{n})}{\Rightarrow} -a > o \wedge -b > o \underset{(\text{v})}{\Rightarrow} (-a)/(-b) = a/b \geq o.$

(y) (1) $\bigwedge_{a \in A} (\inf A \leq a \underset{(\text{OD2})}{\Rightarrow} -a \leq -\inf A \Rightarrow \sup(\{-e\} \cdot A) \leq -\inf A$
$\underset{(\text{OD2})}{\Rightarrow} \inf A \leq -\sup(-A)).$

(2) $\bigwedge_{a \in A} (-a \leq \sup(\{-e\} \cdot A) \underset{(\text{OD2})}{\Rightarrow} -\sup(-A) \leq a$
$\Rightarrow -\sup(-A) \leq \inf A).$

(1) and (2) $\underset{(\text{O3})}{\Rightarrow} \inf A = -\sup(-A).$

(z) Employing (OD5) with multiplication we obtain
$$\bigwedge_{c,d \in R} (c \subseteq d \Rightarrow (-e) \cdot c \subseteq (-e) \cdot d \Rightarrow -c \subseteq -d,$$
$$a \subseteq b \wedge c \subseteq d \Rightarrow a \subseteq b \wedge -c \subseteq -d$$
$$\Rightarrow a - c = a + (-c) \subseteq b + (-d) = b - d).$$ ■

Since Theorem 2.3 is quite ramified, we offer the following comments by way of summary. Theorem 2.3 shows that in a ringoid or division ringoid the minus operator has the same properties as in the real or complex number field. Compared to other algebraic structures, a ringoid is distinguished by the fact that the existence of inverse elements is not assumed. Nevertheless, subtraction is not an independent operation. It can be defined by means of the operations of multiplication and addition.

In a weakly ordered (resp. an ordered) ringoid for all elements that are comparable with o with respect to $\leq$ and $\geq$, the same rules for inequalities hold as for complex (resp. real) numbers. Since in a linearly ordered ringoid, all elements are comparable to o with respect to $\leq$ and $\geq$, then for all elements the same rules for inequalities hold as for inequalities among real numbers.

It is evident but important to note for further applications that $\{\mathbb{R}, +, \cdot\}$ and $\{\mathbb{C}, +, \cdot\}$ are ringoids, while $\{\mathbb{R}, \{0\}, +, \cdot, /\}$ and $\{\mathbb{C}, \{0\}, +, \cdot, /\}$ are division ringoids. $\{\mathbb{R}, +, \cdot, \leq\}$ is a linearly ordered ringoid. However, if the order relation is defined componentwise, $\{\mathbb{C}, +, \cdot, \leq\}$ is only a weakly ordered ringoid. The properties (OD1) and (OD2) are easily verified. We show that (OD3) is not valid by means of a simple example. Let

$$0 \leq \alpha := e^{i\frac{3\pi}{8}} \leq \beta := 2e^{i\frac{3\pi}{8}}, \quad 0 \leq \gamma := e^{i\frac{\pi}{4}}.$$

Then

$$\alpha \cdot \gamma = e^{i\frac{5\pi}{8}} \quad \text{and} \quad \beta \cdot \gamma = 2e^{i\frac{5\pi}{8}}$$

and thus $\text{Re}(\beta \cdot \gamma) \leq \text{Re}(\alpha \cdot \gamma)$.

The following theorem plays a key role in subsequent chapters when we study the spaces listed in Figure 1.

Theorem 2.4. *In a linearly ordered ringoid $\{R, +, \cdot, \leq\}$ the property (D6) is a consequence of the remaining properties (D1,2,3,4,5), (O1,2,3,4), (OD1,2,3).*

Proof. Setting $b = e$ in (D5d) implies

$$\bigwedge_{a \in R} x \cdot a = a \cdot x. \tag{2.1.3}$$

Suppose that two elements $x, y \in R$ fulfill the properties (D5) and (OD2). Then, without loss of generality, we can assume by (O4) that $x \leq y$. Then

$$x \leq y \underset{\text{(OD2)}}{\Rightarrow} xy \leq xx \underset{\text{(D5b)}}{=} e \wedge yy \underset{\text{(D5b)}}{=} e \leq yx. \tag{2.1.4}$$

By (2.1.3) $xy = yx$, and therefore by (2.1.4) and (O3) $xy = e$. If we multiply the equation by x, we obtain $y = x$. ■

Now let R be a nonempty set with elements $a, b, c, \ldots$, and let $\mathbb{P}R$ be the power set of R. For any operation $\circ$ defined for elements of R, we define corresponding operations $\circ$ in $\mathbb{P}R$ by

$$\bigwedge_{A,B \in \mathbb{P}R} A \circ B := \{a \circ b \mid a \in A \wedge b \in B\}. \tag{2.1.5}$$

The least element in $\mathbb{P}R$ with respect to set inclusion as an order relation is the empty set $\varnothing$. The empty set is subset of any set. If in (2.1.5) A or B is the empty set, then the result of the operation is defined to be the empty set. By interpretation as a limit process the latter case is formally included in (2.1.5).

If $\{R, \circ\}$ is a group with neutral element e, then the power set $\{\mathbb{P}R, \circ\}$ is not a group. To see this, we consider the equation $A \circ X = \{e\}$ in $\mathbb{P}R$:

$$A \circ X = \{a \circ x \mid a \in A \wedge x \in X\} = \{e\}.$$

The only solutions of this equation occur for $A = \{a\}$ and $X = \{a^{-1}\}$ with $a \in R$.

From this observation we conclude that the power set of a ring, a field, a vector space, a matrix algebra, is not, respectively, a ring, a field, a vector space, a matrix algebra. However, the following theorem shows that ringoid properties are preserved upon passage to the power set. Because of the property $\{o\} \cdot \varnothing = \varnothing \cdot \{o\} = \varnothing \neq \{o\}$, however, the empty set has to be excluded.

Theorem 2.5. *If* $\{R, +, \cdot\}$ *is a ringoid with the special elements* o, e, *and* x, *then* $\{\mathbb{P}R \setminus \{\varnothing\}, +, \cdot\}$ *is also a ringoid with the special elements* $\{o\}$, $\{e\}$, *and* $\{x\}$. *If* $\{R, N, +, \cdot, /\}$ *is a division ringoid, then* $\{\mathbb{P}R \setminus \{\varnothing\}, N^*, +, \cdot, /\}$ *is also a division ringoid, where* $N^* := \{A \in \mathbb{P}R \mid A \cap N \neq \varnothing\}$. *Furthermore,* $\{\mathbb{P}R \setminus \{\varnothing\}, +, \cdot, \subseteq\}$ *(resp.* $\{\mathbb{P}R \setminus \{\varnothing\}, N^*, +, \cdot, /, \subseteq\}$*) is an inclusion-isotonally ordered ringoid (resp. division ringoid).*

Proof. The properties (D1,2,3,4,7,8), and (OD5) (resp. (OD6)) follow directly from the definition of the operations in the power set and the corresponding properties in R. The properties (D5) and (D9) in $\mathbb{P}R$ are easily verified for $X = \{-e\}$. It remains to prove (D6):

Let S denote the subset of $\mathbb{P}R$ consisting of the elements $\{a\}$, $\{b\}$, $\{c\}, \ldots$ with $a, b, c, \ldots \in R$. Then the mapping $\varphi : R \to S$ defined by $\bigwedge_{a \in R} \varphi a = \{a\}$ is obviously an isomorphism.[2]

We have to show that $X = \{-e\}$ is the only element in $\mathbb{P}R$ that has the properties (D5a,b,c,d). Let X be any element with these properties. Then we have, in particular

(D5b) $X \cdot X = \{x \cdot y \mid x, y \in X\} = \{e\}$,

(D5d) $\bigwedge\limits_{A,B \in \mathbb{P}R} X(AB) = (XA)B = A(XB)$.

From (D5b) we obtain directly

$$\bigwedge_{x,y \in X} x \cdot x = x \cdot y = y \cdot x = y \cdot y = e. \tag{2.1.6}$$

With this we obtain from (D5d) with $A = \{x\}$, $B = \{y\}$ for any $x, y \in X$:

$$X \cdot \{e\} = \{e\} \cdot \{y\} = \{x\} \cdot \{e\}. \tag{2.1.7}$$

This means that X contains only one element $x \in R$. $X = \{x\}$ still has to satisfy (D5a,b,c,d) in $\mathbb{P}R$ for all $A, B \in \mathbb{P}R$, and in particular for $A = \{a\}$ and $B = \{b\}$ in S. Since S is isomorphic to R we obtain $x = -e$. Therefore, $X = \{-e\}$ is the only element in $\mathbb{P}R$ that satisfies (D5). ■

Using Theorem 2.5, it is clear that $\{\mathbb{P}\mathbb{R} \setminus \{\varnothing\}, N, +, \cdot, /, \subseteq\}$ with $N := \{A \in \mathbb{P}\mathbb{R} \mid 0 \in A\}$ and $\{\mathbb{P}\mathbb{C} \setminus \{\varnothing\}, N', +, \cdot, /, \subseteq\}$ with $N' := \{A \in \mathbb{P}\mathbb{C} \mid 0 \in A\}$ are inclusion-isotonally ordered division ringoids with the neutral elements $\{0\}$ and $\{1\}$ and with $X = \{-1\}$.

[2] Let M and N be algebraic structures and let a one-to-one correspondence exist between the operations $\circ$ in M and $\diamond$ in N. A one-to-one mapping $\varphi : M \to N$ is called an isomorphism if $\bigwedge_{a,b \in M} \varphi a \diamond \varphi b = \varphi(a \circ b)$. Then M and N are called isomorphic.

Now we consider matrices over a given ringoid R. Let $P := \{1, 2, \ldots, m\}$ and $Q := \{1, 2, \ldots, n\}$. A matrix is defined as a mapping $\varphi : P \times Q \to R$. We denote the set of all such matrices by $M_{mn}R$. Then

$$M_{mn}R = \left\{ \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \middle| \, a_{ij} \in R, \ i = 1(1)m, \ j = 1(1)n \right\}.$$

We shall often use the more concise notation $A = (a_{ij})$, $B = (b_{ij}) \in M_{mn}R$. Equality of and addition and multiplication for matrices are defined by

$$(a_{ij}) = (b_{ij}) \ :\Leftrightarrow \ a_{ij} = b_{ij}, \ \text{ for all } i = 1(1)m \text{ and } j = 1(1)n,$$

$$\bigwedge_{(a_{ij}),(b_{ij}) \in M_{mn}R} \quad (a_{ij}) + (b_{ij}) := (a_{ij} + b_{ij}),$$

$$\bigwedge_{(a_{ij}) \in M_{mn}R} \ \bigwedge_{(b_{ij}) \in M_{np}R} \quad (a_{ij}) \cdot (b_{ij}) := \left(\sum_{j=1}^{n} a_{ij} b_{jk} \right) \in M_{mp}R.$$

If $\{R, \leq\}$ is an ordered set, an order relation $\leq$ is defined in $M_{mn}R$ by

$$(a_{ij}) \leq (b_{ij}) :\Leftrightarrow a_{ij} \leq b_{ij}, \text{ for all } i = 1(1)m \text{ and } j = 1(1)n.$$

If $m = n$, we simply write M_nR.

The following theorem establishes ringoid properties of M_nR.

Theorem 2.6. *If $\{R, +, \cdot\}$ is a ringoid with the neutral elements o and e, then $\{M_nR, +, \cdot\}$ is also a ringoid, and its neutral elements are*

$$O = \begin{pmatrix} o & o & \ldots & o \\ o & o & \ddots & \vdots \\ \vdots & \ddots & \ddots & o \\ o & \ldots & o & o \end{pmatrix}, \quad E = \begin{pmatrix} e & o & \ldots & o \\ o & e & \ddots & \vdots \\ \vdots & \ddots & \ddots & o \\ o & \ldots & o & e \end{pmatrix},$$

$$X = -E = \begin{pmatrix} -e & o & \ldots & o \\ o & -e & \ddots & \vdots \\ \vdots & \ddots & \ddots & o \\ o & \ldots & o & -e \end{pmatrix}.$$

If, moreover, $\{R, +, \cdot, \leq\}$ is weakly ordered (resp. ordered), then $\{M_nR, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid.

Proof. The properties (D1,2,3,4) and (OD1,2) follow immediately from the definition of the operations in M_nR and the corresponding properties in R. The properties (D5) are easily verified for $X = -E$. It remains to demonstrate (D6) and (OD3).

(D6): Let S denote the set of diagonal matrices[3] with all diagonal elements the same. Then the mapping $\varphi : R \to S$ defined by

$$\varphi a = \begin{pmatrix} a & o & \dots & o \\ o & a & \ddots & \vdots \\ \vdots & \ddots & \ddots & o \\ o & \dots & o & a \end{pmatrix}$$

is obviously an isomorphism.

Now let $X = (x_{ij})$ be any element of M_nR which fulfills (D5). Then we have in particular because of (D5a):

$$E + X = O.$$

This means that $x_{ij} = o$ for all $i \neq j$, i.e., X is a diagonal matrix with diagonal entries x_{ii}, $i = 1(1)n$. From (D5d) we obtain for $B = E$:

$$X \cdot A = A \cdot X \;\text{ for all }\; A = (a_{ij}) \in M_nR.$$

If we now choose $a_{ij} = e$ for all $i, j = 1(1)n$, we obtain $x_{11} = x_{22} = \dots = x_{nn}$, and because of the isomorphism φ, $x_{ii} = x = -e$ for all $i = 1(1)n$, i.e., X is a diagonal matrix with the constant diagonal element $x = -e$. (D6) holds in M_nR.

(OD3): $O \le A \le B \wedge C \ge O \Rightarrow o \le a_{ij} \le b_{ij} \wedge c_{ij} \ge o$, for all $i = 1(1)n$

$\underset{\text{(OD3)}_R}{\Rightarrow} \; o \le a_{ir}c_{rj} \le b_{ir}c_{rj}$, for all $i, j, r = 1(1)n$

$\underset{\text{Theorem 2.3(m)}}{\Rightarrow} \; \sum\limits_{r=1}^{n} a_{ir}c_{rj} \le \sum\limits_{r=1}^{n} b_{ir}c_{rj}$, for all $i, j = 1(1)n$

$\Rightarrow AC \le BC.$ ■

Theorem 2.6 shows that $\{M_n\mathbb{R}, +, \cdot, \le\}$ is an ordered ringoid and that $\{M_n\mathbb{C}, +, \cdot, \le\}$ is a weakly ordered ringoid. Furthermore, from Theorem 2.5 we conclude that both $\{\mathbb{P}M_n\mathbb{R} \setminus \{\varnothing\}, +, \cdot, \subseteq\}$ and $\{\mathbb{P}M_n\mathbb{C}\setminus\{\varnothing\}, +, \cdot, \subseteq\}$ are inclusion-isotonally ordered ringoids.

Now we consider the process of complexification of a given ringoid (resp. division ringoid) R. We denote the set of all pairs over R by $\mathbb{C}R := \{(a_1, a_2) \mid a_1, a_2 \in R\}$. Here a_1 is called the real part of (a_1, a_2), while a_2 is called the imaginary part. Equality, addition, multiplication, and division of elements $\alpha = (a_1, a_2)$, $\beta = (b_1, b_2) \in$

[3] A matrix is called a diagonal matrix if $a_{ij} = 0$ for all $i \neq j$. If the diagonal entries are all zero, we call it the zero matrix; if the diagonal entries are all e, we call it the unit matrix.

$\mathbb{C}R$ are defined by

$$\begin{aligned}
\alpha = \beta &:\Leftrightarrow a_1 = b_1 \wedge a_2 = b_2, \\
\alpha + \beta &:= (a_1 + b_1, a_2 + b_2), \text{ for all } \alpha, \beta \in \mathbb{C}R, \\
\alpha \cdot \beta &:= (a_1 b_1 - a_2 b_2, a_1 b_2 + a_2 b_1), \text{ for all } \alpha, \beta \in \mathbb{C}R, \\
\alpha / \beta &:= ((a_1 b_1 + a_2 b_2)/b, (a_2 b_1 - a_1 b_2)/b), \text{ with } b = b_1 b_1 + b_2 b_2, \\
&\qquad \text{for all } \alpha, \beta \in \mathbb{C}R \setminus \overline{N} \text{ and} \\
&\qquad \overline{N} := \{\gamma = (c_1, c_2) \in \mathbb{C}R \mid c_1 c_1 + c_2 c_2 \in N\}.
\end{aligned}$$

If $\{R, \leq\}$ is an ordered set, an order relation $\leq$ is defined in $\mathbb{C}R$ by

$$(a_1, a_2) \leq (b_1, b_2) :\Leftrightarrow a_1 \leq b_1 \wedge a_2 \leq b_2.$$

The following theorem shows that ringoid properties are preserved under complexification.

Theorem 2.7. *If $\{R, +, \cdot\}$ is a ringoid with the neutral elements o and e, then $\{\mathbb{C}R, +, \cdot\}$ is also a ringoid with the neutral elements $\omega = (o, o)$ and $\epsilon = (e, o)$. Moreover $-\epsilon = (-e, o)$. If $\{R, N, +, \cdot, /\}$ is a division ringoid, then $\{\mathbb{C}R, \overline{N}, +, \cdot, /\}$ is also a division ringoid. If $\{R, +, \cdot, \leq\}$ is a weakly ordered ringoid, $\{\mathbb{C}R, +, \cdot, \leq\}$ is likewise a weakly ordered ringoid.*

Proof. The properties (D1,2,3,4,7,8) and (OD1,2) follow directly from the definition of the operations and the corresponding properties in R. The properties (D5) and (D9) can be shown to be valid for $\xi = (-e, o) \in \mathbb{C}R$: (D5a) and (D5b) are easily verified for this ξ. (D5c), (D5d) and (D9) can be proved in a straightforward manner by using the property

$$\xi\alpha = (-e, o)(a_1, a_2) = (-a_1, -a_2).$$

The only remaining property is (D6).

(D6): If we denote the set of all elements of $\mathbb{C}R$ with absent imaginary part by S, then the mapping $\varphi : R \to S$, defined by $\varphi a = (a, o)$ for all $a \in R$, is obviously an isomorphism.

Now let us assume that $\eta = (x, y)$ is any element of $\mathbb{C}R$ that satisfies (D5). η must satisfy (D5a), which means

$$\epsilon + \eta = \omega \Leftrightarrow e + x = o \wedge y = o,$$

i.e., η is of the form $\eta = (x, o)$. Since η satisfies (D5a,b,c,d) for all $\alpha, \beta \in \mathbb{C}R$, it does so in particular for $\alpha = (a, o)$ and $\beta = (b, o)$ with $a, b \in R$. Because of the isomorphism $R \leftrightarrow S$, the properties (D5a,b,c,d) in $\mathbb{C}R$ for elements (a, o) and (b, o) are equivalent to (D5a,b,c,d) in R. Since R is a ringoid, there exists only one element $x = -e$. Therefore $\xi = (-e, o)$ is the only element in $\mathbb{C}R$ that satisfies (D5). ■

2.2 Vectoids

We begin our discussion of vectoids with the following definition.

Definition 2.8. Let $\{R, +, \cdot\}$ be a ringoid with elements $a, b, c, \ldots$, and with neutral elements o and e. Let $\{V, +\}$ be a groupoid with elements $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \ldots$, and following properties:

(V1) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b}\in V} \mathfrak{a} + \mathfrak{b} = \mathfrak{b} + \mathfrak{a}$,

(V2) $\bigvee\limits_{\mathfrak{o}\in V} \bigwedge\limits_{\mathfrak{a}\in V} \mathfrak{a} + \mathfrak{o} = \mathfrak{a}$.

V is called an *R-vectoid* $\{V, R\}$ if there is an outer multiplication $\cdot : R \times V \to V$ defined, which with the abbreviation

$$\bigwedge_{\mathfrak{a}\in V} -\mathfrak{a} = (-e) \cdot \mathfrak{a}$$

has the following properties:

(VD1) $\bigwedge\limits_{a\in R} \bigwedge\limits_{\mathfrak{a}\in V} (a \cdot \mathfrak{o} = \mathfrak{o} \wedge o \cdot \mathfrak{a} = \mathfrak{o})$,

(VD2) $\bigwedge\limits_{\mathfrak{a}\in V} e \cdot \mathfrak{a} = \mathfrak{a}$,

(VD3) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b}\in V} -(\mathfrak{a} + \mathfrak{b}) = (-\mathfrak{a}) + (-\mathfrak{b})$,

(VD4) $\bigwedge\limits_{a\in R} \bigwedge\limits_{\mathfrak{a}\in V} -(a \cdot \mathfrak{a}) = (-a) \cdot \mathfrak{a} = a \cdot (-\mathfrak{a})$.

An R-vectoid is called *multiplicative* if a multiplication in V with the following properties is defined:

(V3) $\bigvee\limits_{\mathfrak{e}\in V\setminus\{\mathfrak{o}\}} \bigwedge\limits_{\mathfrak{a}\in V} \mathfrak{a} \cdot \mathfrak{e} = \mathfrak{e} \cdot \mathfrak{a} = \mathfrak{a}$,

(V4) $\bigwedge\limits_{\mathfrak{a}\in V} \mathfrak{a} \cdot \mathfrak{o} = \mathfrak{o} \cdot \mathfrak{a} = \mathfrak{o}$,

(V5) $\mathfrak{e} - \mathfrak{e} = \mathfrak{o}$,

(VD5) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b}\in V} -(\mathfrak{a}\mathfrak{b}) = (-\mathfrak{a})\mathfrak{b} = \mathfrak{a}(-\mathfrak{b})$.

Now let $\{R, +, \cdot, \leq\}$ be a weakly ordered ringoid. Then an R-vectoid or a multiplicative R-vectoid is called *weakly ordered* if $\{V, \leq\}$ is an ordered set[4] which has the following two properties:

(OV1) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b},\mathfrak{c}\in V} (\mathfrak{a} \leq \mathfrak{b} \Rightarrow \mathfrak{a} + \mathfrak{c} \leq \mathfrak{b} + \mathfrak{c})$,

(OV2) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b}\in V} (\mathfrak{a} \leq \mathfrak{b} \Rightarrow -\mathfrak{b} \leq -\mathfrak{a})$.

[4] Since it is always clear by the context which order relation is meant, we denote the order relation in V and R by the same sign $\leq$.

A weakly ordered R-vectoid is called *ordered* if

(OV3) $\bigwedge\limits_{a,b\in R}\ \bigwedge\limits_{\mathfrak{a},\mathfrak{b}\in V} (o \leq a \leq b \wedge \mathfrak{o} \leq \mathfrak{a} \Rightarrow a\cdot\mathfrak{a} \leq b\cdot\mathfrak{a} \quad \wedge$
$\qquad o \leq a \wedge \mathfrak{o} \leq \mathfrak{a} \leq \mathfrak{b} \Rightarrow \quad a\cdot\mathfrak{a} \leq a\cdot\mathfrak{b}).$

A multiplicative vectoid is called *ordered* if it is an ordered vectoid and

(OV4) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b},\mathfrak{c}\in V} (\mathfrak{o} \leq \mathfrak{a} \leq \mathfrak{b} \wedge \mathfrak{o} \leq \mathfrak{c} \Rightarrow \mathfrak{a}\cdot\mathfrak{c} \leq \mathfrak{b}\cdot\mathfrak{c} \wedge \mathfrak{c}\cdot\mathfrak{a} \leq \mathfrak{c}\cdot\mathfrak{b}).$

An R-vectoid that may be multiplicative is called *inclusion-isotonally ordered* with respect to order relations $\{R,\subseteq\}$ and $\{V,\subseteq\}$ if

(OV5) (a) $\bigwedge\limits_{a,b\in R}\ \bigwedge\limits_{\mathfrak{c},\mathfrak{d}\in V} (a \subseteq b \wedge \mathfrak{c} \subseteq \mathfrak{d} \Rightarrow a\cdot\mathfrak{c} \subseteq b\cdot\mathfrak{d}),$

(b) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b},\mathfrak{c},\mathfrak{d}\in V} (\mathfrak{a} \subseteq \mathfrak{b} \wedge \mathfrak{c} \subseteq \mathfrak{d} \Rightarrow \mathfrak{a}\circ\mathfrak{c} \subseteq \mathfrak{b}\circ\mathfrak{d}), \circ \in \{+,\cdot\}.$ ■

A vectoid (resp. a multiplicative vectoid) is just a set of elements in which one (resp. two) special element $\mathfrak{o}$ (and $\mathfrak{e}$) exists and which obeys rules concerning inner and outer operations as enumerated in Definition 2.8. A vectoid may be weakly ordered or ordered with respect to one order relation and inclusion-isotonally ordered with respect to another order relation. In the following definition, we use the minus operator of a vectoid in order to define a subtraction.

Definition 2.9. In an R-vectoid $\{V,R\}$ we define a subtraction by

$$\bigwedge_{\mathfrak{a},\mathfrak{b}\in V} \mathfrak{a}-\mathfrak{b} = \mathfrak{a}+(-\mathfrak{b}).$$ ■

Every vector space is a vectoid. This means that a vectoid represents a certain substructure or a generalization of a vector space. To see this, we recall that the latter is an additive group with an outer multiplication with the elements of a field with the following properties:

(1) $a(\mathfrak{a}+\mathfrak{b}) = a\mathfrak{a}+a\mathfrak{b}$,

(2) $(a+b)\mathfrak{a} = a\mathfrak{a}+b\mathfrak{a}$,

(3) $a(b\mathfrak{a}) = (ab)\mathfrak{a}$,

(4) $e\mathfrak{a} = \mathfrak{a}$.

These properties imply (V1) and (V2) directly. (VD2) is the same as (4). (VD4) follows from (3) upon setting $a = -e$ (resp. $b = -e$), and (VD3) follows from (1) upon setting $a = -e$. To show (VD1), we argue as follows:

$$o\mathfrak{a} = (e-e)\mathfrak{a} = \mathfrak{a}-\mathfrak{a} = \mathfrak{o},$$
$$a\mathfrak{o} = a\mathfrak{o}+a\mathfrak{o}-a\mathfrak{o} \underset{(1)}{=} a(\mathfrak{o}+\mathfrak{o})-a\mathfrak{o} = a\mathfrak{o}-a\mathfrak{o} = \mathfrak{o}.$$

A vectoid possesses many of the familiar properties of a vector space. Some of them are summarized by the following theorem.

Theorem 2.10. *Let $\{V, R\}$ be a vectoid with the neutral element $\mathfrak{o}$ and with the neutral element $\mathfrak{e}$ if a multiplication exists. Then for all $a, b \in R$, and for all $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d} \in V$, the following properties hold:*

(a) $\mathfrak{o}$ *and* $\mathfrak{e}$ *are unique,*

(b) $\mathfrak{o} - \mathfrak{a} = -\mathfrak{a}$,

(c) $-(-\mathfrak{a}) = \mathfrak{a}$,

(d) $-(\mathfrak{a} - \mathfrak{b}) = -\mathfrak{a} + \mathfrak{b} = \mathfrak{b} - \mathfrak{a}$,

(e) $(-a)(-\mathfrak{a}) = a\mathfrak{a}$,

(f) $-\mathfrak{a} = \mathfrak{o} \Leftrightarrow \mathfrak{a} = \mathfrak{o}$,

(g) $\mathfrak{a} - \mathfrak{z} = \mathfrak{o} \Leftrightarrow \mathfrak{z} = \mathfrak{o}$*, i.e.,* $\mathfrak{o}$ *is the only right neutral element of subtraction.*

In a multiplicative vectoid, we have further

(h) $-\mathfrak{a} = (-\mathfrak{e}) \cdot \mathfrak{a} = \mathfrak{a} \cdot (-\mathfrak{e})$,

(i) $(-\mathfrak{a}) \cdot (-\mathfrak{b}) = \mathfrak{a} \cdot \mathfrak{b}$,

(j) $-\mathfrak{e}$ *is the unique solution of the equation* $(-\mathfrak{e}) \cdot \mathfrak{z} = \mathfrak{e}$.

In a weakly ordered vectoid $\{V, R, \leq\}$*, we have*

(k) $\mathfrak{a} \leq \mathfrak{b} \wedge \mathfrak{c} \leq \mathfrak{d} \Rightarrow \mathfrak{a} + \mathfrak{c} \leq \mathfrak{b} + \mathfrak{d}$,

(l) $\mathfrak{a} \leq \mathfrak{b} \Rightarrow -\mathfrak{b} \leq -\mathfrak{a}$.

In an ordered vectoid, we have additionally

(m) $o \leq a \leq b \wedge \mathfrak{o} \leq \mathfrak{c} \leq \mathfrak{d} \Rightarrow \mathfrak{o} \leq a\mathfrak{c} \leq b\mathfrak{d}$,

(n) $a \leq b \leq o \wedge \mathfrak{o} \leq \mathfrak{c} \leq \mathfrak{d} \Rightarrow a\mathfrak{d} \leq b\mathfrak{c} \leq \mathfrak{o}$,

(o) $a \leq b \leq o \wedge \mathfrak{c} \leq \mathfrak{d} \leq \mathfrak{o} \Rightarrow \mathfrak{o} \leq b\mathfrak{d} \leq a\mathfrak{c}$,

(p) $o \leq a \leq b \wedge \mathfrak{c} \leq \mathfrak{d} \leq \mathfrak{o} \Rightarrow b\mathfrak{c} \leq a\mathfrak{d} \leq \mathfrak{o}$.

In an ordered multiplicative vectoid, we have further

(q) $\mathfrak{o} \leq \mathfrak{a} \leq \mathfrak{b} \wedge \mathfrak{o} \leq \mathfrak{c} \leq \mathfrak{d} \Rightarrow \mathfrak{o} \leq \mathfrak{a}\mathfrak{c} \leq \mathfrak{b}\mathfrak{d} \wedge \mathfrak{o} \leq \mathfrak{c}\mathfrak{a} \leq \mathfrak{d}\mathfrak{b}$,

(r) $\mathfrak{a} \leq \mathfrak{b} \leq \mathfrak{o} \wedge \mathfrak{o} \leq \mathfrak{c} \leq \mathfrak{d} \Rightarrow \mathfrak{a}\mathfrak{d} \leq \mathfrak{b}\mathfrak{c} \leq \mathfrak{o} \wedge \mathfrak{d}\mathfrak{a} \leq \mathfrak{c}\mathfrak{b} \leq \mathfrak{o}$,

(s) $\mathfrak{a} \leq \mathfrak{b} \leq \mathfrak{o} \wedge \mathfrak{c} \leq \mathfrak{d} \leq \mathfrak{o} \Rightarrow \mathfrak{o} \leq \mathfrak{b}\mathfrak{d} \leq \mathfrak{a}\mathfrak{c} \wedge \mathfrak{o} \leq \mathfrak{d}\mathfrak{b} \leq \mathfrak{c}\mathfrak{a}$,

(t) $\mathfrak{o} \leq \mathfrak{a} \leq \mathfrak{b} \wedge \mathfrak{c} \leq \mathfrak{d} \leq \mathfrak{o} \Rightarrow \mathfrak{b}\mathfrak{c} \leq \mathfrak{a}\mathfrak{d} \leq \mathfrak{o} \wedge \mathfrak{c}\mathfrak{b} \leq \mathfrak{d}\mathfrak{a} \leq \mathfrak{o}$.

In a completely and weakly ordered vectoid, we have

(u) $\bigwedge\limits_{\mathfrak{A} \in \mathbb{P}V} \inf \mathfrak{A} = -\sup(-\mathfrak{A}) \wedge \sup \mathfrak{A} = -\inf(-\mathfrak{A})$.

In an inclusion-isotonally ordered vectoid $\{V, R, \subseteq\}$*, (OV5) holds for subtraction:*

(v) $\bigwedge\limits_{\mathfrak{a},\mathfrak{b},\mathfrak{c},\mathfrak{d} \in V} (\mathfrak{a} \subseteq \mathfrak{b} \wedge \mathfrak{c} \subseteq \mathfrak{d} \Rightarrow \mathfrak{a} - \mathfrak{c} \subseteq \mathfrak{b} - \mathfrak{d})$.

Proof. We omit the explicit proof of this theorem since, using the corresponding properties of a vectoid, it is completely analogous to that of Theorem 2.3. ■

Theorem 2.10 allows us to assert that in a vectoid or a multiplicative vectoid the same rules for the minus operator hold as in a vector space or a matrix algebra over the real or complex number fields. Compared to other algebraic structures, a vectoid is distinguished by the fact that the existence of inverse elements is not assumed. Nevertheless, subtraction is not an independent operation. It can be defined by means of the outer multiplication and addition.

In an ordered vectoid (resp. multiplicative vectoid) for all elements that are comparable with $\mathfrak{o}$ with respect to $\leq$ and $\geq$, the same rules for inequalities hold as in the vector space $\mathbb{R}^n$ over the real numbers (resp. in the algebra of matrices over the real numbers). In a weakly ordered vectoid for all elements that are comparable with $\mathfrak{o}$ with respect to $\leq$ and $\geq$, the same rules for inequalities hold as in the vector space $\mathbb{C}^n$ over the complex numbers.

In a multiplicative vectoid, the inner operations $+$ and $\cdot$ always have the properties (D1,2,3,4,5) of a ringoid. The properties (D1,2,3,4,5a) are identical with (V1,2,3,4,5), while (D5b,c,d) can easily be proved:

(D5b) $(-\mathfrak{e})(-\mathfrak{e}) \underset{\text{Theorem 2.10(i)}}{=} \mathfrak{e} \cdot \mathfrak{e} = \mathfrak{e}$,

(D5c) $(-\mathfrak{e})(\mathfrak{a} + \mathfrak{b}) \underset{\text{(h)}}{=} -(\mathfrak{a} + \mathfrak{b}) \underset{\text{(VD3)}}{=} (-\mathfrak{a}) + (-\mathfrak{b}) \underset{\text{(h)}}{=} (-\mathfrak{e})\mathfrak{a} + (-\mathfrak{e})\mathfrak{b}$,

(D5d) $(-\mathfrak{e})(\mathfrak{a} \cdot \mathfrak{b}) \underset{\text{(h)}}{=} -(\mathfrak{a}\mathfrak{b}) \underset{\text{(VD5)}}{=} (-\mathfrak{a})\mathfrak{b} = \mathfrak{a}(-\mathfrak{b}) \underset{\text{(h)}}{=} ((-\mathfrak{e})\mathfrak{a})\mathfrak{b} = \mathfrak{a}((-\mathfrak{e})\mathfrak{b})$.

It can be shown by examples [132], however, that a multiplicative vectoid is not necessarily a ringoid.

Powersets of vectoids are vectoids as well. This property is made precise by the following theorem. As in the case of a ringoid, the empty set again has to be excluded. (VD1) cannot be satisfied for the empty set. We have instead $\emptyset \cdot \{\mathfrak{o}\} = \{o\}\emptyset = \emptyset \neq \{\mathfrak{o}\}$.

Theorem 2.11. *If $\{V, R\}$ is a vectoid with the neutral element $\mathfrak{o}$, then $\{\mathbb{P}V \setminus \{\emptyset\}, \mathbb{P}R \setminus \{\emptyset\}, \subseteq\}$ is an inclusion-isotonally ordered vectoid with the neutral element $\{\mathfrak{o}\}$. If $\{V, R\}$ is multiplicative and $\mathfrak{e}$ is the neutral element of multiplication, then $\{\mathbb{P}V \setminus \{\emptyset\}, \mathbb{P}R \setminus \{\emptyset\}, \subseteq\}$ is an inclusion-isotonally ordered multiplicative vectoid and $\{\mathfrak{e}\}$ is its neutral element of multiplication.*

Proof. The proof is straightforward. All properties can be obtained from the corresponding properties in $\{V, R\}$ and the definition of the operations in the power set. ■

Once again let us consider matrices over a given ringoid R. In addition to the inner operations and the order relation defined above in $M_{mn}R$, we now define an outer multiplication by

$$\bigwedge_{a \in R} \bigwedge_{(a_{ij}) \in M_{mn}R} a \cdot (a_{ij}) := (a \cdot a_{ij}).$$

Then the following theorem holds.

Theorem 2.12. *Let $\{R, +, \cdot\}$ be a ringoid with the neutral elements o and e. Then $\{M_{mn}R, R\}$ is a vectoid. The neutral element is the matrix all components of which are o. If $\{R, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid, then $\{M_{mn}R, R, \leq\}$ is a weakly ordered (resp. an ordered) vectoid.*

Proof. We omit the proof of this theorem since it is straightforward. ■

For $m = 1$ we obtain matrices that consist only of one column. Such matrices are called vectors. The product set $V_nR := R \times R \times \ldots \times R = M_{1n}$ represents the set of all vectors with n components. In this case the last theorem specializes into the following form.

Theorem 2.13. *Let $\{R, +, \cdot\}$ be a ringoid with the neutral elements o and e. Then $\{V_nR, R\}$ is a vectoid. The neutral element is the zero vector, i.e., the vector all components of which are o. If $\{R, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid, then $\{V_nR, R, \leq\}$ is a weakly ordered (resp. an ordered) vectoid.* ■

Another important specialization of the $m \times n$-matrices is the set of $n \times n$-matrices M_nR. In this case we obtain the following specialization of Theorem 2.12.

Theorem 2.14. *Let $\{R, +, \cdot\}$ be a ringoid with neutral elements o and e. Then $\{M_nR, R\}$ is a multiplicative vectoid. The neutral elements are the zero matrix and the unit matrix. If $\{R, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid, then $\{M_nR, R, \leq\}$ is a weakly ordered (resp. an ordered) multiplicative vectoid.*

Proof. (V1,2,3,4,5) follow immediately from the properties (D1,2,3,4,5a) of the ringoid M_nR. Likewise, (OV4) follows from (OD3). The properties (VD1,2,3,4) and (OV1,2,3) were already verified in Theorem 2.12. The remaining property (VD5) follows from (D5d) in M_nR and the property (h) of Theorem 2.10. ■

Among the consequences of these theorems we may note again that $\{V_n\mathbb{R}, \mathbb{R}, \leq\}$ is an ordered vectoid, while $\{V_n\mathbb{C}, \mathbb{C}, \leq\}$ is a weakly ordered vectoid, $\{M_n\mathbb{R}, \mathbb{R}, \leq\}$ is an ordered multiplicative vectoid, and $\{M_n\mathbb{C}, \mathbb{C}, \leq\}$ is a weakly ordered multiplicative vectoid. We shall use these theorems later for many other applications.

We have also defined a multiplication of an $m \times n$-matrix by an $n \times p$-matrix. The most important case of this multiplication is that of a matrix by a vector. The result then is a vector, and the structure to be described in the following theorem matches that of a vectoid.

Theorem 2.15. *Let $\{R, +, \cdot\}$ be a ringoid, $\{M_nR, +, \cdot\}$ the ringoid of $n \times n$-matrices over R, and V_nR the set of n-tuples over R. Then $\{V_nR, M_nR\}$ is a vectoid. The neutral element is the zero vector. If $\{M_nR, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid, then $\{V_nR, M_nR, \leq\}$ is a weakly ordered (resp. an ordered) vectoid.*

Proof. The properties (V1,2), (VD1,2,4), (OV1,2) follow immediately from the definition of the operations. The properties (VD3) and (OV3) can be proved in complete analogy of (D5c) and (OD3) in $M_n R$ (Theorem 2.6). ■

As a consequence of this theorem, of course, $\{V_n\mathbb{R}, M_n\mathbb{R}, \leq\}$ is an ordered vectoid, while $\{V_n\mathbb{C}, M_n\mathbb{C}, \leq\}$ is a weakly ordered vectoid.

We have now established in this chapter that the spaces listed in the leftmost element of every row in Figure 1 are ringoids and vectoids with certain order properties such as being weakly ordered or ordered or inclusion-isotonally ordered.

We conclude this section by pointing out that ringoids and vectoids also occur in sets of mappings. Let $\{R, +, \cdot\}$ be a ringoid with the neutral elements o and e and let M be a set of mappings of a given set S into R, i.e., $M := \{x \mid x : S \to R\}$. In M we define an equality, addition, and outer and inner multiplication respectively by

$$\begin{aligned}
x = y \quad &:\Leftrightarrow \quad x(t) = y(t), \quad \text{for all } t \in S,\\
\bigwedge_{x,y\in M} (x+y)(t) \quad &:= \quad x(t) + y(t), \quad \text{for all } t \in S,\\
\bigwedge_{a\in R}\ \bigwedge_{x\in M} (a\cdot x)(t) \quad &:= \quad a \cdot x(t), \quad \text{for all } t \in S,\\
\bigwedge_{x,y\in M} (x\cdot y)(t) \quad &:= \quad x(t)\cdot y(t), \quad \text{for all } t \in S.
\end{aligned}$$

If $\{R, \leq\}$ is an ordered set, we define an order relation $\leq$ in M by

$$x \leq y \quad :\Leftrightarrow \quad x(t) \leq y(t), \quad \text{for all } t \in S.$$

Then the following theorem, which characterizes sets of mappings in terms of the structures in question, is easily verified.

Theorem 2.16. *Let $\{R, +, \cdot\}$ be a ringoid with the neutral elements o and e. Then $\{M, R\}$ is a multiplicative vectoid. The neutral elements are $o(t) = o$ and $e(t) = e$ for all $t \in S$. If $\{R, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid, then $\{M, R, \leq\}$ is a weakly ordered (resp. an ordered) vectoid. With respect to the inner operations $\{M, +, \cdot\}$ is a ringoid.* ■

In conclusion, a clarifying remark on the results that have been obtained may be useful. At first sight it may seem strange that for the power set the empty set has to be excluded in order to obtain the ringoid and vectoid structures. This, however, is only a notational problem. The power set of a set R is one of the most frequently used complete lattices. Its least element is the empty set and the greatest element is the set R itself. Thus, the concept of the power set is well established. That the least element does not satisfy certain algebraic properties should not be surprising. The ordering of algebraic structures is very often only conditionally complete and it is well known that completion by adding a least and a greatest element is often impossible without violating their algebraic properties.

The real numbers $\mathbb{R}$, for instance, can be defined as a conditionally complete linearly ordered field. Derived spaces such as those in the second column of Figure 1 are also conditionally complete with respect to the order relation $\leq$. If the real numbers are completed in the customary manner, the least element $-\infty$ and the greatest element $+\infty$, which fail to satisfy the algebraic properties, need not be notationally excluded.

Because of the central role of the real numbers computing is usually performed in a conditionally completely ordered algebraic structure. We shall see in the following chapters that on a computer, because of rounding or for other reasons, the least or the greatest element of the order structure can nevertheless occur as result of an algebraic operation.

In order not to interrupt the computation in such cases, attempts have been undertaken to define algebraic operations also for the least and the greatest element. This may lead to reasonable results in some cases; in others it does not. We do not follow these lines in this treatise. We think that exceptional cases in a computation should be a challenge for the user to think further about his problem and to deal with such cases individually.

Remark 2.17. The discrepancy between a conditionally complete and a completely ordered structure has a notational consequence for the following chapters. In Definition 1.22 monotone and directed roundings have been defined as mappings of a complete lattice into a lower and upper screen, respectively. In this chapter we have defined and studied the concepts of a ringoid and a vectoid. We have seen that in a weakly ordered or ordered ringoid or vectoid the least and the greatest element might not satisfy the algebraic properties of the ringoid or vectoid. Henceforth, we therefore assume that the ordering of a weakly ordered or ordered ringoid or vectoid is conditionally complete. Then we have to distinguish between the symbol for the ringoid or vectoid and its lattice theoretic completion. If the ringoid or vectoid is denoted by R we shall denote its lattice theoretic completion by $\overline{R}$. With this notation a rounding $\square$ then becomes a mapping $\square : \overline{R} \to \overline{S}$, where $\overline{S}$ is a lower (resp. upper) screen of the complete lattice $\overline{R}$. ■

Chapter 3
Definition of Computer Arithmetic

In all branches of mathematics one is interested in gaining insight into the structures that can occur. We have established in the preceding chapter that the spaces listed in the leftmost element of every row in Figure 1 are ringoids and vectoids with certain order properties such as being weakly ordered or ordered or inclusion-isotonally ordered. We are now going to show that these structures recur in the subsets on the right-hand side of Figure 1 provided that the arithmetic and mapping properties are properly defined. In Section 3.1 of this chapter we give a heuristic approach to the mapping concept of a semimorphism. In Section 3.2 we derive some fundamental properties of semimorphisms and certain other mappings. In Section 3.3 we give the conventional definition of computer arithmetic and show that it produces ordered and weakly ordered ringoids and vectoids in the subsets of Figure 1. In Section 3.4 we introduce the definition of computer arithmetic by means of semimorphism. This leads to the best possible arithmetic in all spaces and in the product spaces in particular. We show expressly for all rows of Figure 1 that do not include interval sets that weakly ordered and ordered ringoids and vectoids are invariant with respect to semimorphisms. The definition of arithmetic by semimorphism will also be used in Chapter 4 to derive similar results for the rows of Figure 1 that correspond to sets of intervals.

3.1 Introduction

Let R be a set in which certain operations and relations are defined, and let R' be a set of rules or axioms given for these operations and relations. By way of example, the commutative law of addition might be one such rule. Then we call the pair $\{R, R'\}$ a structure. We shall also refer to $\{R, R'\}$ as the structure of the set R. The real or complex numbers, vectors, or matrices are well-known structures. We now assume that certain operations in $\{R, R'\}$ cannot be carried out by a computer. As an example take the representation and addition of irrational numbers. We are then obliged to replace $\{R, R'\}$ by a structure $\{S, S'\}$, the operations of which can be done by machine and which lead to good approximations to the operations in $\{R, R'\}$. We shall always take $S \subseteq R$.

In all significant applications there exist at least two operations in R: an addition $+$ and a multiplication $\cdot$, which have neutral elements or an identity operator in the case of an outer multiplication. Let us denote these elements by o and e. In addition we shall always have a minus operator. We assume therefore that the subset S has the property

(S3) $o, e \in S \quad \wedge \quad \bigwedge_{a \in S} \; -a \in S.$

We further assume that the elements of R are mapped into the subset S by a rounding with the property

(R1) $\bigwedge_{a \in S} \; \Box a = a.$

In attempting to approximate a given structure $\{R, R'\}$ by a structure $\{S, S'\}$ with $S \subseteq R$, we are motivated to employ well-established mappings having useful properties such as isomorphism or homomorphism. For the sake of clarity, we give the definition of these concepts for ordered algebraic structures.

Definition 3.1. Let $\{R, R'\}$ and $\{S, S'\}$ be ordered algebraic structures, and let a one-to-one correspondence exist between the corresponding operations and order relation(s). A mapping $\Box : R \to S$ is called a *homomorphism* if it is an *algebraic homomorphism*, i.e., if the image of the operation is equal to the operation of the images:

$$\bigwedge_{a,b \in R} (\Box a) \boxdot (\Box b) = \Box(a \circ b) \tag{3.1.1}$$

for all corresponding operations $\circ$ in R and $\boxdot$ in S and if it is an *order homomorphism*, i.e., if

$$\bigwedge_{a,b \in R} (a \leq b \Rightarrow \Box a \leq \Box b). \tag{3.1.2}$$

A homomorphism is called an *isomorphism* if $\Box : R \to S$ is a one-to-one mapping of R onto S and

$$\bigwedge_{a,b \in S} (a \leq b \Rightarrow \Box^{-1} a \leq \Box^{-1} b). \qquad \blacksquare$$

Thus an isomorphism requires a one-to-one mapping. However, in our applications S is in general a subset of R of different cardinality. There does not exist a one-to-one mapping and certainly no isomorphism between sets of different cardinality.

A homomorphism preserves the structure of a group, a ring, or a vector space. However, we show by a simple example that the approximation of $\{R, R'\}$ by $\{S, S'\}$ by means of homomorphism cannot be realized in a sensible way. A simple theorem which proves this statement is given in Appendix B.

Example 3.2. Let S be a floating-point system[1] that is defined as a set of numbers of the form $X = m \cdot b^e$. m is called the mantissa, b the base of the system and e the exponent. Let $b = 10$, let the range of the mantissa be $m = -0.9(0.1)0.9$, and let the exponent be $e \in \{-1, 0, 1\}$. If $\square : \mathbb{R} \to S$ denotes the rounding to the nearest number of S and $x = 0.34, y = 0.54 \in \mathbb{R}$, we get $\square x = 0.3$, $\square y = 0.5$ and thus $(\square x) \boxplus (\square y) = \square((\square x) + (\square y)) = \square(0.8) = 0.8$ but $\square(x + y) = \square(0.88) = 0.9$, i.e., $(\square x) \boxplus (\square y) \neq \square(x + y)$.

It seems desirable, however, to stay as close to a homomorphism as possible. We therefore derive some necessary conditions for a homomorphism. If we restrict (3.1.1) to elements of S, then because of (R1) we obtain

(RG) $\bigwedge\limits_{a,b \in S} \quad a \boxed{\circ} b = \square(a \circ b)$.

We shall use this formula to define the operation $\boxed{\circ}$ in S by means of the corresponding operation $\circ$ in R and the rounding $\square : \overline{R} \to \overline{S}$ (see Remark 2.17). From (3.1.2) we see that the rounding has to be monotone

(R2) $\bigwedge\limits_{a,b \in R} \quad (a \leq b \Rightarrow \square a \leq \square b)$.

Now in the case of multiplication in (3.1.1), replace a by the negative multiplicative unit $-e$. Then

$$\bigwedge_{b \in R} \square(-b) = \square(-e) \boxdot \square b \underset{\text{(S3),(R1)}}{=} (-e) \boxdot \square b \underset{\text{(RG)}}{=} \square(-\square b) \underset{\text{(S3),(R1)}}{=} -\square b.$$

Thus the rounding has to be antisymmetric:

(R4) $\bigwedge\limits_{a \in R} \quad \square(-a) = -\square a$.

Specifically (see Definition 3.3), we shall call a mapping a semimorphism if it has the properties (R1,2,4), and (RG).

The conditions (R1,2,4) do not define the rounding uniquely. We shall see, however, in this chapter and in Chapter 4 that in all cases of Figure 1, ordered or weakly ordered ringoids and vectoids as well as inclusion-isotonally ordered ringoids and vectoids are invariant with respect to semimorphisms. We shall see further in Part 2 of this book that semimorphisms can be implemented on computers by fast algorithms and hardware circuits in all cases of Figure 1. These properties give to the concept of semimorphism a significance that is quite independent of the present derivation of it via the necessary conditions for a homomorphism.

3.2 Preliminaries

In ringoids and vectoids there is a minus operator available. We use it to extend the concept of a screen and of a rounding. We do this in the following definition, where we also formalize the notion of a semimorphism.

[1] See Chapter 5 for a general definition of a floating-point system.

Definition 3.3. Let R be a ringoid or a vectoid and o the neutral element of addition and e the neutral element of the multiplication if it exists. Further let $\{\overline{R}, \leq\}$ be a complete lattice.

(a) A lower (resp. upper) screen $\{S, \leq\}$ is called *symmetric* if

(S3) $o, e \in S \quad \wedge \quad \bigwedge_{a \in S} - a \in S.$

(b) A rounding of R into a symmetric lower (resp. upper) screen S, $\square : R \to S$ is called *antisymmetric* if

(R4) $\bigwedge_{a \in R} \square(-a) = -\square a$ (antisymmetric).

(c) A mapping of the ringoid or vectoid R into a symmetric lower (resp. upper) screen is called a *semimorphism* if it is a monotone and antisymmetric rounding, i.e., has the properties (R1,2,4), and if all inner and outer operations in S are defined by

(RG) $\bigwedge_{a,b \in S} a \boxdot b := \square(a \circ b).$ ■

With the minus operator in ringoids and vectoids an interdependence of the special roundings $\triangledown$ and $\vartriangle$ can now be proved. It is the subject of the following theorem.

Theorem 3.4. *If $\{R, \leq\}$ is a weakly ordered ringoid or vectoid, $\{\overline{R}, \leq\}$ a complete lattice and $\{\overline{S}, \leq\}$ a symmetric screen of $\overline{R}$, and $\triangledown : \overline{R} \to \overline{S}$ (resp. $\vartriangle : \overline{R} \to \overline{S}$) the monotone downwardly (resp. upwardly) directed roundings, then*

$$\bigwedge_{a \in R} (\triangledown a = -\vartriangle(-a) \wedge \vartriangle a = -\triangledown(-a)).$$

Proof. We give the proof only for a ringoid. For a vectoid, the proof can be given analogously by using corresponding properties of Theorem 2.10.

If $\{R, +, \cdot\}$ is a ringoid, then $\{\mathbb{P}R \setminus \{\varnothing\}, +, \cdot\}$ is a ringoid. With

$$L(a) := \{b \in R \mid b \leq a\} \quad \text{and} \quad U(a) := \{b \in R \mid a \leq b\}$$

we obtain because of (OD2) that

$$-(L(a) \cap S) := \{b \in S \mid -b \leq a\} = \{b \in S \mid -a \leq b\} = U(-a) \cap S$$

and

$$-(U(a) \cap S) := \{b \in S \mid a \leq -b\} = \{b \in S \mid b \leq -a\} = L(-a) \cap S.$$

Using this and Theorem 2.3(y), we obtain

$$\sup(L(a) \cap S) = -\inf(-(L(a) \cap S)) = -\inf(U(-a) \cap S)$$

and

$$\inf(U(a) \cap S) = -\sup(-(U(a) \cap S)) = -\sup(L(-a) \cap S).$$

This and the general property $\nabla a = \sup(L(a) \cap S) \wedge \triangle a = \inf(U(a) \cap S)$ prove the theorem. ■

This theorem can easily be illustrated when R is a linearly ordered ringoid and S is a finite subset of R. See Figure 3.1. It has been proved, independently, for much more general cases such as complexifications and vector or matrix sets.

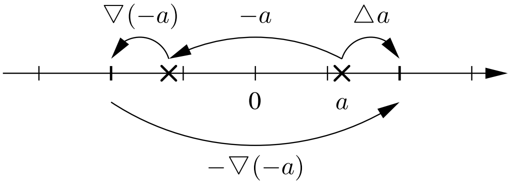


Figure 3.1. Dependence of directed roundings in a linearly ordered ringoid.

Theorem 3.4 asserts that under its hypothesis the monotone directed roundings are interdependent. Thus only one of the monotone directed roundings is necessary when both are to be implemented on computers. The other one could then be obtained by sign changes.

If $S \neq R$ and we take an $a \in R$ with $a \notin S$, then $\nabla a < a < \triangle a$. A comparison of Definition 3.3 and Theorem 3.4 shows that the monotone directed roundings are not antisymmetric under the hypothesis of the theorem. In contrast to this result, we shall consider in Chapter 4 several monotone upwardly directed roundings that are antisymmetric. In this case the screen and the monotone directed roundings are defined with respect to an order relation that is different from the one that orders the ringoid or vectoid.

Crucial for the whole development and to applications in the next sections and in Chapter 4 is the following theorem.

Theorem 3.5 (Rounding invariance of ringoid properties.). *Let R be a ringoid or division ringoid, or a weakly ordered or ordered ringoid or division ringoid with the special elements o, e, and $-e$. Further let $\{\overline{R}, \leq\}$ be a complete lattice and $\{\overline{S}, \leq\}$ a symmetric (lower resp. upper) screen of $\overline{R}$ and $\square : \overline{R} \to \overline{S}$ a semimorphism. Then besides (D6), all properties of R also hold for the resulting structure in S. In particular*

(a) the property (Di) in S is a consequence of (Di) in R, for $i = 1, 2, 3, 4, 5, 7, 8, 9$ with the same special elements o, e, and $-e$ in S,

(b) (ODi) in S is a consequence of (ODi) in R, for $i = 1, 2, 3, 4, 5$,

(c) from (RG) and (Ri) follows (RGi) in S, for $i = 1, 2, 3, 4$ and for all $\circ \in \{+, -, \cdot, /\}$ with

(RG4) $\bigwedge\limits_{a \in S} (-e) \cdot a = (-e) \boxdot a$,

(d) If $\{S, \boxplus, \boxdot\}$ is a ringoid then (RG) and (RGi), $i = 1, 2, 3$ also hold for subtraction.

Proof. The proof of the theorem is obtained in a straightforward manner. In order to gain familiarity with the theorem we recommend it be worked through in complete detail. As sample here we just prove property (D5d). As a first step we show the validity of

(RG4): $\bigwedge\limits_{a \in S} (-e) \boxdot a \underset{\text{(RG)}}{=} \square(-a) \underset{\text{(R4)}}{=} -\square a \underset{\text{(R1)}}{=} -a \underset{\text{(S3)}}{\in} S.$

(D5d): $(-e) \boxdot (a \boxdot b) \underset{\text{(RG)}}{=} \square((-e) \cdot \square(ab)) \underset{\text{(R4)}}{=} \square(\square(-(ab)))$

$$\underset{\text{(R1)}}{=} \square(-(ab)) = \underset{\text{(D5d)}_R}{=} \square((-a)b) \underset{\text{(D5d)}_R}{=} \square(a(-b))$$

$$\underset{\text{(RG)}}{=} (-a) \boxdot b \underset{\text{(RG)}}{=} a \boxdot (-b) \underset{\text{(RG4)}}{=} ((-e) \boxdot a) \boxdot b$$

$$\underset{\text{(RG4)}}{=} a \boxdot ((-e) \boxdot b).$$

(d): If $\{S, \boxplus, \boxdot\}$ is a ringoid, we have for all $a, b \in S$ that

$$a \boxminus b := a \boxplus (\boxminus b) \underset{\text{(RG4)}}{=} a \boxplus (-b) \underset{\text{(RG)}_+}{=} \square(a - b),$$

which is (RG) and (RG1) for subtraction. (RG2) (resp. (RG3)) follow from (RG) and (R2) (resp. (R3)). ■

All properties of a semimorphism (S3), (R1,2,4), and (RG) are repeatedly used within the proof of Theorem 3.5. If we change these properties or if they are not strictly implemented on a computer (even with respect to quantifiers), we get a different structure or no describable structure at all in the subset S.

Theorem 3.5 asserts that if we generate S as hypothesized, we almost obtain for S itself the structure of a ringoid (resp. a division ringoid). What is missing is the establishment of the property (D6). A universal proof of this latter property is a more difficult task, and we are obliged to establish it individually for all cases in Figure 1.

For ease of reading we formally repeat the compatibility properties (RGi), $i = 1(1)3$, between the structure in R and in S that are mentioned in Theorem 3.5(c). Under the hypothesis of Theorem 3.5 we have

(RG1) $\bigwedge\limits_{a,b \in S} (a \circ b \in S \Rightarrow a \boxed{\circ} b = a \circ b)$,

(RG2) $\bigwedge\limits_{a,b,c,d \in S} (a \circ b \leq c \circ d \Rightarrow a \boxed{\circ} b \leq c \boxed{\circ} d)$,

(RG3) $\bigwedge\limits_{a,b \in S} a \boxed{\circ} b \leq a \circ b$ resp. $\bigwedge\limits_{a,b \in S} a \circ b \leq a \boxed{\circ} b.$

Here $\circ$ denotes any operation of the set $\{+, -, \cdot, /\}$.

Since neither an isomorphism nor a homomorphism between the basic set R and the screen S is available, we define arithmetic on the screen through semimorphisms. As was done in the case of a groupoid, for brevity we characterize the compatibility properties, which we then obtain between the operations in the basic set R and in the screen S, by the following definition.

Definition 3.6. Let R be a ringoid (resp. vectoid) and S a symmetric screen of R. A ringoid (resp. vectoid) in S is called

- a *screen ringoid* (resp. *vectoid*) if (RG1) holds,
- *monotone* if (RG2) holds,
- a *lower* or *upper screen ringoid* (resp. *vectoid*) if (RG3) holds,

always for all inner (resp. inner and outer) operations. ■

A screen ringoid always has the same special elements as the basic ringoid itself since $\bigwedge\limits_{a \in S} (a + o \in S \wedge a \cdot e \in S \underset{\text{(RG1)}}{\Rightarrow} a \boxplus o = a + o = a \wedge a \boxdot e = a \cdot e = a)$ and $-e := (-e) \cdot e \in S \underset{\text{(RG1)}}{\Rightarrow} (-e) \boxdot e = \boxminus e.$

Similarly a screen vectoid always has the same neutral elements as the basic vectoid itself, and, since in Definition 3.6 S is a symmetric screen, a screen vectoid always has the same minus operator as the basic vectoid itself. For all $\mathfrak{a} \in S$ we have $-\mathfrak{a} := (-e)\mathfrak{a} \underset{\text{(S3)}}{\in} S \underset{\text{(RG1)}}{\Rightarrow} (-e) \boxdot \mathfrak{a} = \boxminus \mathfrak{a}.$

In the case of a linearly ordered ringoid we can use Theorem 3.5 and Theorem 2.4 to obtain the following theorem.

Theorem 3.7. *Let $\{R, +, \cdot, \leq\}$ be a linearly ordered ringoid, $\{\overline{R}, \leq\}$ a complete lattice, $\{\overline{S}, \leq\}$ a symmetric screen, and $\square : \overline{R} \to \overline{S}$ a semimorphism. Then $\{S, \boxplus, \boxdot, \leq\}$ is a linearly ordered monotone screen ringoid.* ■

Since the real numbers are a linearly ordered ringoid, Theorem 3.7 asserts, for example, that the floating-point numbers are a linearly ordered monotone screen ringoid if their arithmetic is defined by a semimorphism. This process can be repeated each time passing into a coarser subset and each time obtaining a linearly ordered monotone screen ringoid. Theorem 3.7 enables us to define the arithmetic in the sets D and S in Figure 1. These in turn can be used to define the arithmetic in the corresponding columns of Figure 1. This latter process is the traditional definition of computer arithmetic in the product sets in Figure 1. It will be discussed in the next section.

3.3 The Traditional Definition of Computer Arithmetic

Let us now assume that $\{R, N, +, \cdot, /, \leq\}$ is a linearly ordered division ringoid. We already know by Theorem 2.7 that the process of complexification leads to a weakly ordered division ringoid $\{\mathbb{C}R, N', +, \cdot, /, \leq\}$. As for the matrices over R and $\mathbb{C}R$, we know by Theorem 2.6 that $\{M_nR, +, \cdot, \leq\}$ is an ordered ringoid and that $\{M_n\mathbb{C}R, +, \cdot, \leq\}$ is a weakly ordered ringoid. The definition of the operations in $\mathbb{C}R$, M_nR, and $M_n\mathbb{C}R$ by those of R as given preceding Theorems 2.6 and 2.7, we call this the *traditional* or *conventional* definition of computer arithmetic.

If now D is a symmetric screen of R and S a symmetric screen of D and if we define the arithmetic in D and S by semimorphism (stepwise), then by Theorem 3.7 we once more obtain linearly ordered ringoids. Therefore, we can again define an arithmetic and structure in the derived sets M_nD, $\mathbb{C}D$, $M_n\mathbb{C}D$ and M_nS, $\mathbb{C}S$, $M_n\mathbb{C}S$ by the traditional definition of the arithmetic and get the same structures as in the corresponding sets over R. See Figure 3.2.

$$\begin{tikzcd}
R \arrow[rr, "\text{Theorem 3.7}"] \arrow[dr, "\text{Theorem 2.6}"] \arrow[dd, "\text{Theorem 2.7}"'] & & D \arrow[rr, "\text{Theorem 3.7}"] \arrow[dr, "\text{Theorem 2.6}"] \arrow[dd, "\text{Theorem 2.7}"'] & & S \arrow[dr, "\text{Theorem 2.6}"] \arrow[dd, "\text{Theorem 2.7}"'] & \\
& M_nR & & M_nD & & M_nS \\
\mathbb{C}R \arrow[d, "\text{Theorem 2.6}"'] & & \mathbb{C}D \arrow[d, "\text{Theorem 2.6}"'] & & \mathbb{C}S \arrow[d, "\text{Theorem 2.6}"'] & \\
M_n\mathbb{C}R & & M_n\mathbb{C}D & & M_n\mathbb{C}S &
\end{tikzcd}$$

Figure 3.2. The traditional definition of computer arithmetic leading to ringoids.

As an example, R may be the linearly ordered ringoid of real numbers, D the subset of double precision floating-point numbers, and S the subset of single precision floating-point numbers of a given computer.

A similar process can be used to define vectoids over R, D, and S. Figure 3.3 lists the resulting spaces and the theorems with which we proved the vectoid properties. The spaces listed immediately under R, D, and S are ordered vectoids, and those listed under $\mathbb{C}R$, $\mathbb{C}D$, and $\mathbb{C}S$ are weakly ordered vectoids.

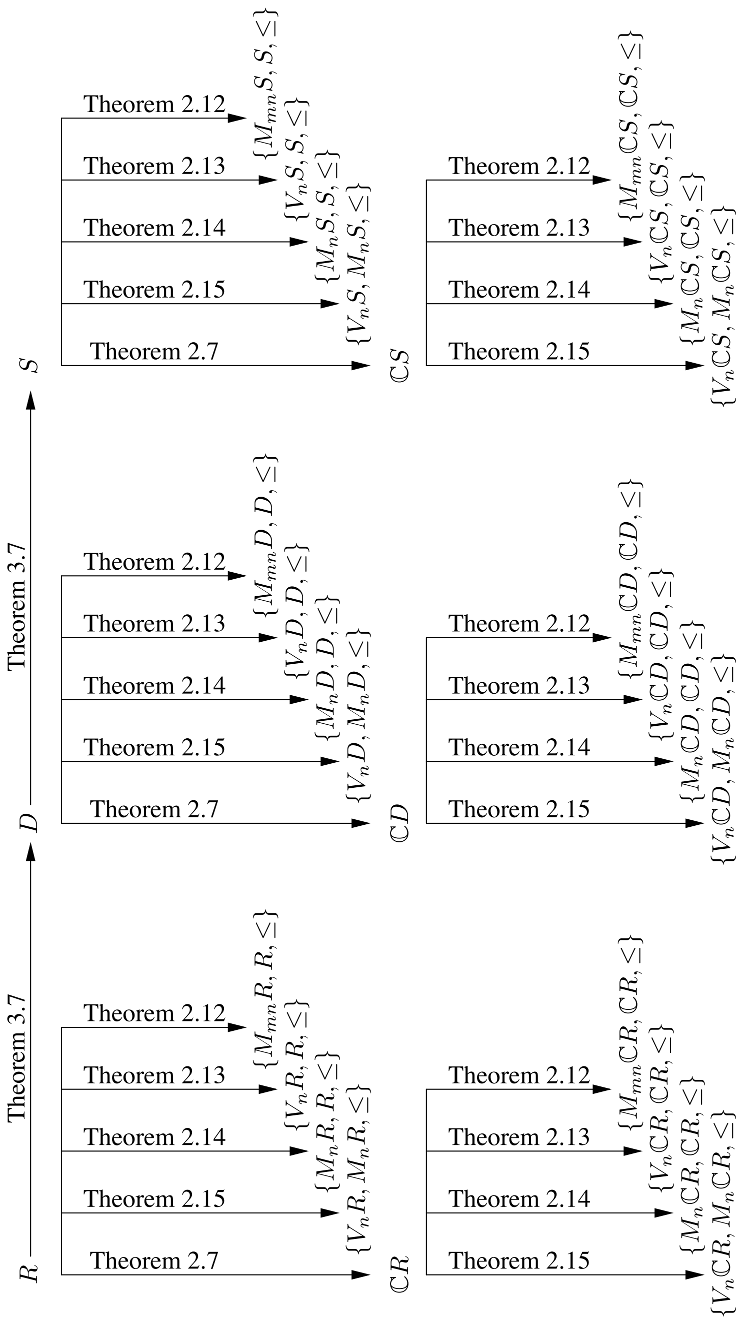

R
Theorem 3.7
D
Theorem 3.7
S
Theorem 2.7
Theorem 2.15
Theorem 2.14
Theorem 2.13
Theorem 2.12
$\mathbb{C}R$
$\{V_nR, M_nR, \leq\}$
$\{M_nR, R, \leq\}$
$\{V_nR, R, \leq\}$
$\{M_{mn}R, R, \leq\}$
$\{V_n\mathbb{C}R, M_n\mathbb{C}R, \leq\}$
$\{M_n\mathbb{C}R, \mathbb{C}R, \leq\}$
$\{V_n\mathbb{C}R, \mathbb{C}R, \leq\}$
$\{M_{mn}\mathbb{C}R, \mathbb{C}R, \leq\}$
$\mathbb{C}D$
$\{V_nD, M_nD, \leq\}$
$\{M_nD, D, \leq\}$
$\{V_nD, D, \leq\}$
$\{M_{mn}D, D, \leq\}$
$\{V_n\mathbb{C}D, M_n\mathbb{C}D, \leq\}$
$\{M_n\mathbb{C}D, \mathbb{C}D, \leq\}$
$\{V_n\mathbb{C}D, \mathbb{C}D, \leq\}$
$\{M_{mn}\mathbb{C}D, \mathbb{C}D, \leq\}$
$\mathbb{C}S$
$\{V_nS, M_nS, \leq\}$
$\{M_nS, S, \leq\}$
$\{V_nS, S, \leq\}$
$\{M_{mn}S, S, \leq\}$
$\{V_n\mathbb{C}S, M_n\mathbb{C}S, \leq\}$
$\{M_n\mathbb{C}S, \mathbb{C}S, \leq\}$
$\{V_n\mathbb{C}S, \mathbb{C}S, \leq\}$
$\{M_{mn}\mathbb{C}S, \mathbb{C}S, \leq\}$

Figure 3.3. The traditional definition of computer arithmetic leading to vectoids.

The traditional method of defining arithmetic is used on most existing computers to specify the arithmetic of floating-point vectors, matrices, complexifications, and so on. The operations in M_nS, for instance, are defined by the operations in S and the usual formulas for addition and multiplication of real matrices. We saw that the resulting structures can be described in terms of ordered or weakly ordered ringoids and vectoids.

This will also be one of the main results of the definition of the arithmetic in the subsets by means of semimorphisms, which we discuss in Section 3.4. The definition of the arithmetic by semimorphisms goes further, leading to simple compatibility properties such as (RG1,2,3,4) between the operations in the basic set and the screen. It is the connection between these two classes of operations that is of central interest.

In Chapter 5 we derive error estimates for the operations defined by both methods. It will turn out that the error formulas for the definition of the operations by semimorphisms are much simpler and more accurate than those of the traditional method and consequently allow a simpler and more accurate error analysis of numerical algorithms. The resulting structures of both methods are the same: ringoids and vectoids. But the arithmetic operations defined by the two methods are very different.

3.4 Definition of Computer Arithmetic by Semimorphisms

The first result of the definition of computer arithmetic by means of semimorphisms, of course, is contained in Theorem 3.7.

In order to define the arithmetic in all the other spaces listed in Figures 3.4 and 3.5 by means of semimorphisms, we first prove the following theorem.

Theorem 3.8. *Let $\{R, +, \cdot\}$ be a ringoid, $\{\overline{R}, \leq\}$ a complete lattice, $\{\overline{S}, \leq\}$ a symmetric lower (resp. upper) screen, and $\square : \overline{R} \to \overline{S}$ a monotone and antisymmetric rounding. Consider the product sets $\overline{V_nS} := \overline{S} \times \overline{S} \times \cdots \times \overline{S} \subseteq \overline{V_nR}$. Then $\{\overline{V_nR}, \leq\}$ is a complete lattice if the order relation is defined componentwise and $\{\overline{V_nS}, \leq\}$ is a symmetric lower (resp. upper) screen of $\{\overline{V_nR}, \leq\}$. Further the mapping $\square : \overline{V_nR} \to \overline{V_nS}$ defined by*

$$\bigwedge_{a=(a_i)\in V_nR} \square a := \begin{pmatrix} \square a_1 \\ \square a_2 \\ \vdots \\ \square a_n \end{pmatrix}$$

is a monotone and antisymmetric rounding.

Proof. Although the proof is straightforward, we give it in complete detail in order to review the concepts dealt with. As a product set of complete lattices, $\{\overline{V_nR}, \leq\}$ is itself a complete lattice. The subset $\{\overline{V_nS}, \leq\}$ is an ordered set and, as a product set of complete lattices, also a complete lattice and therefore a complete subnet of $\{\overline{V_nR}, \leq\}$. The supremum (resp. infimum) in $\overline{V_nS}$ equals the vector of suprema (resp. infima) taken in $\overline{S}$. Since $\{\overline{S}, \leq\}$ is a lower (resp. upper) screen of $\overline{R}$, the latter equals the suprema (resp. infima) taken in $\overline{R}$. Consequently, the supremum (resp. infimum) taken in $\overline{V_nS}$ equals the supremum (resp. infimum) taken in $\overline{V_nR}$. Then by Theorem 1.19, $\overline{V_nS}$ is a lower (resp. upper) screen of $\{\overline{V_nR}, \leq\}$. Since $\overline{S}$ is a symmetric screen of $\overline{R}$, then the zero vector is an element of $\overline{V_nS}$, and for all $a = (a_i) \in V_nS, -a = (-a_i) \in V_nS$, i.e., $\overline{V_nS}$ is a symmetric lower (resp. upper) screen of $\{\overline{V_nR}, \leq\}$. The mapping $\square : \overline{V_nR} \to \overline{V_nS}$ is defined by rounding componentwise. This directly implies that the properties (R1), (R2), and (R4) of the rounding $\square : \overline{R} \to \overline{S}$ also hold for the rounding $\square : \overline{V_nR} \to \overline{V_nS}$. ■

A matrix of $M_{mn}R$ may be viewed as an element of the $m \times n$-dimensional product set. Therefore, Theorem 3.8 remains valid if V_nR is replaced by $M_{mn}R$ or M_nR and V_nS by $M_{mn}S$ or M_nS. The rounding for matrices has to be defined by rounding the components. The following two theorems define the arithmetic in the sets listed in Figure 3.2 by means of semimorphisms.

Theorem 3.9. *Let $\{R, +, \cdot\}$ be a ringoid, $\{\overline{R}, \leq\}$ a complete lattice, $\{\overline{S}, \leq\}$ a symmetric screen of $\{\overline{R}, \leq\}$, and $\{S, ⊞, ⊡\}$ a ringoid defined by a semimorphism $\square : \overline{R} \to \overline{S}$. Further, let $\{M_nR, +, \cdot\}$ be the ringoid of $n \times n$-matrices over R. Operations in M_nS are specified by the semimorphism $\square : \overline{M_nR} \to \overline{M_nS}$ defined by*

$$\bigwedge_{A=(a_{ij})\in M_nR} \square A := (\square a_{ij}) \quad \textit{and}$$

(RG) $$\bigwedge_{A,B\in M_nS} A \boxed{\circ} B := \square(A \circ B), \quad \textit{for } \circ \in \{+, \cdot\}.$$

(a) Then $\{M_nS, ⊞, ⊡\}$ is a monotone screen ringoid of M_nR.

(b) If the rounding $\square : \overline{R} \to \overline{S}$ is upwardly (resp. downwardly) directed, then the ringoid $\{M_nS, ⊞, ⊡\}$ is an upper (resp. lower) screen ringoid.

(c) If $\{M_nR, +, \cdot, \leq\}$ is a weakly ordered (resp. an ordered) ringoid, then $\{M_nS, ⊞, ⊡, \leq\}$ is a weakly ordered (resp. an ordered) screen ringoid of M_nR.

Proof. (a) By Theorem 3.8 $\{\overline{M_nS}, \leq\}$ is a symmetric screen of $\{\overline{M_nR}, \leq\}$, and the rounding $\square : \overline{M_nR} \to \overline{M_nS}$ is monotone and antisymmetric. Now Theorem 3.5 implies the validity of the properties (D1,2,3,4,5), (RG1,2), and (RG4). It remains only to prove (D6).

(D6): Let O and E denote the neutral elements of $\{M_nR, +, \cdot\}$. By Theorem 3.5 we know that the element $-E$ satisfies (D5) in M_nS. We show that it is unique with respect to this property. Let $X = (x_{ij})$ be any element of M_nS that satisfies (D5). Then by (D5a),

$$X ⊞ E = O, \tag{3.4.1}$$

and by the definition of the addition and the rounding, we get for $i \neq j$:

$$x_{ij} \boxplus o = o \underset{(\text{D2})_S}{\Rightarrow} x_{ij} = o,$$

i.e., X is a diagonal matrix. Specializing to the diagonal in (3.4.1), we get

$$x_{ii} \boxplus e = o \quad \text{for all } i = 1(1)n, \tag{3.4.2}$$

and by (D5b) for X:

$$x_{ii} \boxdot x_{ii} = e \quad \text{for all } i = 1(1)n. \tag{3.4.3}$$

From (D5c) and (D5d) in M_nS we get for two diagonal matrices with the constant diagonal entries a and b

$$\bigwedge_{a,b \in S} x_{ii} \boxdot (a \boxplus b) = x_{ii} \boxdot a \boxplus x_{ii} \boxdot b, \tag{3.4.4}$$

and

$$\bigwedge_{a,b \in S} x_{ii} \boxdot (a \boxdot b) = (x_{ii} \boxdot a) \boxdot b = a \boxdot (x_{ii} \boxdot b). \tag{3.4.5}$$

Relations (3.4.2)–(3.4.5) imply that x_{ii} has to satisfy the axiom (D5) in S. Since $\{S, \boxplus, \boxdot\}$ is a screen ringoid of $\{R, +, \cdot\}$, $x_{ii} = -e$ for all $i = 1(1)n$, and therefore $X = -E \in M_nS$.

(b) and (c) are simple consequences of Theorem 3.5. ■

Theorem 3.10. *Let $\{R, +, \cdot\}$ be a ringoid, $\{\overline{R}, \leq\}$ a complete lattice, $\{\overline{S}, \leq\}$ a symmetric screen of $\{\overline{R}, \leq\}$, and $\{S, \boxplus, \boxdot\}$ a ringoid defined by a semimorphism $\square : \overline{R} \to \overline{S}$. Further, let $\{\mathbb{C}R, +, \cdot\}$ be the ringoid of pairs (a, b) over R, and let operations in $\mathbb{C}S$ be specified by the semimorphism $\square : \overline{CR} \to \overline{CS}$ defined by*

$$\bigwedge_{\alpha=(a_1,a_2)\in\mathbb{C}R} \square\alpha = (\square a_1, \square a_2), \text{ and}$$

(RG) $\displaystyle\bigwedge_{\alpha,\beta\in\mathbb{C}S} \alpha \boxcircle \beta := \square(\alpha \circ \beta), \circ \in \{+, \cdot\}.$

(a) Then $\{\mathbb{C}S, \boxplus, \boxdot\}$ is a monotone screen ringoid of $\mathbb{C}R$.

(b) If $\square : \overline{R} \to \overline{S}$ is upwardly (resp. downwardly) directed, then $\{\mathbb{C}S, \boxplus, \boxdot\}$ is an upper (resp. lower) screen ringoid of $\mathbb{C}R$.

(c) If $\{R, +, \cdot, \leq\}$ is weakly ordered, then $\{\mathbb{C}S, \boxplus, \boxdot, \leq\}$ is a weakly ordered screen ringoid of $\mathbb{C}R$.

(d) Now let $\{R, N, +, \cdot, /\}$ be a division ringoid, $\{\mathbb{C}R, N', +, \cdot, /\}$ with $N' = \{(c_1, c_2) \in \mathbb{C}R \mid c_1c_1 + c_2c_2 \in N\}$, its complexification, and $\{S, N \cap S, \boxplus, \boxdot, \boxslash\}$ a division ringoid defined by the semimorphism $\square : \overline{R} \to \overline{S}$. If division by an element $\beta \notin N' \cap \mathbb{C}S$ is defined by the semimorphism (RG), then $\{\mathbb{C}S, N' \cap \mathbb{C}S, \boxplus, \boxdot, \boxslash\}$ is a monotone screen division ringoid of $\mathbb{C}R$.

Proof. (a) By Theorem 3.8 $\{\overline{\mathbb{C}S}, \leq\}$ is a symmetric screen of $\{\overline{\mathbb{C}R}, \leq\}$, and $\square : \overline{\mathbb{C}R} \to \overline{\mathbb{C}S}$ is a monotone and antisymmetric rounding. Theorem 3.5 provides the properties (D1,2,3,4,5), (RG1,2), and (RG4). It remains to prove (D6).

(D6): Let o and e denote the neutral elements of $\{R, +, \cdot\}$. Then $\omega = (o, o)$ and $\epsilon = (e, o)$ are the neutral elements of $\{\mathbb{C}R, +, \cdot\}$. By Theorem 3.5 we know that the element $(-e, o)$ satisfies (D5) in $\mathbb{C}S$.
We show that it is unique with respect to this property. Let $\eta = (x, y)$ be any element of $\mathbb{C}S$ that satisfies (D5). Then by (D5a),

$$\eta \boxplus (e, o) = (o, o)$$

and by the definition of addition and the rounding, we get

$$y \boxplus o = o \underset{\text{(D2)}_S}{\Rightarrow} y = o,$$

i.e., η is of the form $\eta = (x, o)$.

Let us now denote by T the set of all elements of $\mathbb{C}S$ with an imaginary part of zero (second component). Then the mapping $\varphi : S \to T$ defined by

$$\bigwedge_{a \in S} \varphi(a) = (a, o)$$

is a ringoid isomorphism.

The element $\eta = (x, o)$ has to satisfy (D5) for all $\alpha, \beta \in \mathbb{C}S$, and in particular for $\alpha = (a, o)$ and $\beta = (b, o)$ with $a, b \in S$. Because of the isomorphism $T \leftrightarrow S$, the properties (D5) in $\mathbb{C}S$ for the elements (a, o) and (b, o) are equivalent to (D5) in S. Since $\{S, \boxplus, \boxdot\}$ is a screen ringoid, there exists only one such element $x = -e$. Therefore, $\eta = (-e, o)$ is the only element in $\mathbb{C}S$ that satisfies (D5).

(b), (c) and (d) are simple consequences of Theorem 3.5. ■

The operations defined in M_nS and $\mathbb{C}S$ by Theorems 3.9 and 3.10 are quite different from those defined in the same sets by the traditional definition of arithmetic in these subsets in Theorems 2.6 and 2.7.

Theorems 3.9 and 3.10 already specify the definition of the arithmetic by semimorphisms for several of the sets displayed under D in Figure 3.2. As an example, let R denote the linearly ordered set of real numbers, S and D the subsets of single and double precision floating-point numbers of a given computer. Then the arithmetic and structure are defined in M_nD by Theorem 3.9, in $\mathbb{C}D$ by Theorem 3.10, and in $M_n\mathbb{C}D$ once again by Theorem 3.9. See Figure 3.4.

These theorems, however, cannot be used directly to define the arithmetic and structure in M_nS, $\mathbb{C}S$, and $M_n\mathbb{C}S$. The reason for this is that in Theorems 3.9 and 3.10 the arithmetic in the spaces M_nR, $\mathbb{C}R$, and $M_n\mathbb{C}R$ is defined vertically in the traditional way. The arithmetics and structures of the spaces D, M_nD, $\mathbb{C}D$, and $M_n\mathbb{C}D$, however, are no longer connected vertically in the traditional way.

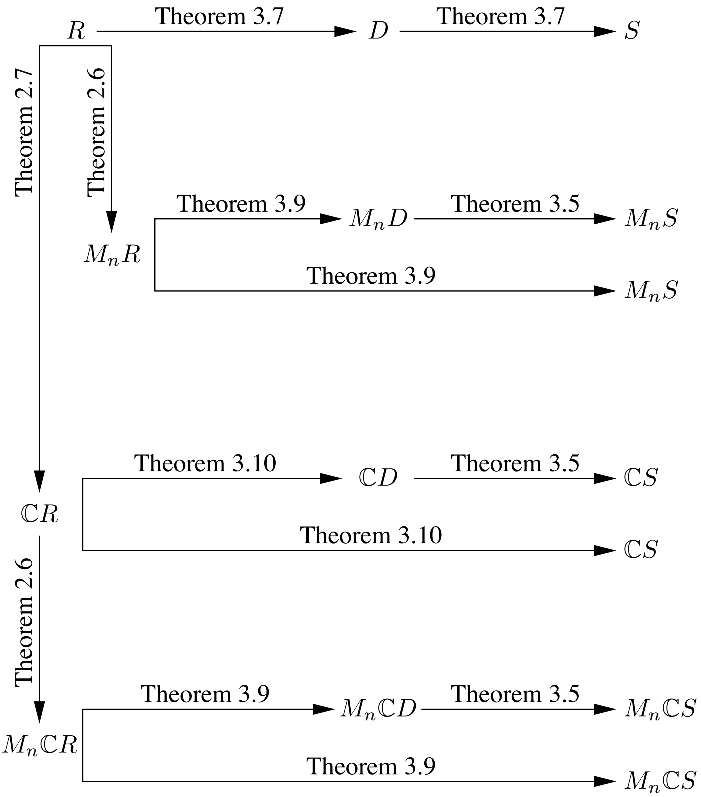


Figure 3.4. The definition of computer arithmetic by semimorphism leading to ringoids.

Nevertheless, the following consideration leads to the desired results. If $\overline{S}$ is a symmetric screen of $\overline{D}$, then so is $\overline{M_nS}$ of $\overline{M_nD}$. See Figure 3.4. For a monotone and antisymmetric rounding $\Diamond : \overline{D} \rightarrow \overline{S}$, we consider the semimorphism $\Diamond : \overline{M_nD} \rightarrow \overline{M_nS}$ defined by

$$\bigwedge_{A=(a_{ij})\in M_nD} \Diamond A := (\Diamond a_{ij}) \quad \text{and}$$

(RG) $\bigwedge_{A,B\in M_nS} A \mathbin{\Diamond\!\!\!\!\circ}\, B := \Diamond(A \mathbin{\boxed{\circ}} B)$, for $\circ \in \{+, \cdot\}$.

Then, apart from (D6), $\{M_nS, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}\}$ has all desired properties as we know from Theorem 3.5.

To see that $\{M_nS, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}, \leq\}$ is a ringoid, we consider the composition of the mappings $\Diamond\, \Box : \overline{R} \rightarrow \overline{S}$. This composition is also a monotone and antisymmetric rounding. By Theorem 3.9, therefore, the semimorphism $\Diamond\, \Box : \overline{M_nR} \rightarrow \overline{M_nS}$ with the property

(RG) $\bigwedge_{A,B\in M_nS} A \mathbin{\Diamond\!\!\!\!\circ}\, B := \Diamond(\square(A\circ B))$, for $\circ\in\{+,\cdot\}$,

leads to a ringoid in M_nS. This ringoid is identical to the one defined by the semimorphism $\Diamond : \overline{M_nD}\to\overline{M_nS}$.

$\{M_nS, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}\,, \leq\}$, therefore, is a weakly ordered (resp. an ordered) monotone screen ringoid of M_nD.

Using the composition of the mappings $\square : \overline{\mathbb{C}R}\to\overline{\mathbb{C}D}$ and $\Diamond : \overline{\mathbb{C}D}\to\overline{\mathbb{C}S}$, it can be proved analogously by Theorem 3.5 and Theorem 3.10 that $\{\mathbb{C}S, N'\cap\mathbb{C}S, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}\,, \mathbin{\Diamond\!\!\!\!/}\,, \leq\}$ is a weakly ordered monotone screen division ringoid of $\mathbb{C}D$.

Similarly, Theorem 3.5 and Theorem 3.9 lead to the result that $\{M_n\mathbb{C}S, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}\,, \leq\}$ is a weakly ordered monotone screen ringoid of $\{M_n\mathbb{C}D, \boxplus, \boxdot, \leq\}$. See Figure 3.4.

In Figure 3.4 the chain $R\supseteq D\supseteq S$ can also be extended to the right by means of coarser symmetric screens. Again it is clear by Theorems 3.5, 3.9, and 3.10 that we obtain ringoids in all lines of Figure 3.4. In this sense the ringoid structures in all lines of Figure 3.4 are invariant with respect to semimorphisms.

The following theorem enables us to define the arithmetic in the sets listed in Figure 3.3 by means of semimorphisms.

Theorem 3.11 (Rounding invariance of vectoid properties). *Let $\{V,R\}$ be a vectoid, which may be multiplicative, $\{\overline{V},\leq\}$ a complete lattice, $\{\overline{S},\leq\}$ a symmetric lower (resp. upper) screen of $\{\overline{V},\leq\}$, and $\{T, \oplus, \odot\}$ a screen ringoid of $\{R,+,\cdot\}$. Further, let $\square : \overline{V}\to\overline{S}$ be a semimorphism, i.e., a monotone and antisymmetric rounding with which operations are defined by*

(RG) $\bigwedge_{\mathfrak{a},\mathfrak{b}\in S} \mathfrak{a} \mathbin{\square\!\!\!\!\circ}\, \mathfrak{b} := \square(\mathfrak{a}\circ\mathfrak{b}), \circ\in\{+,\cdot\}, \qquad \bigwedge_{a\in T}\bigwedge_{\mathfrak{a}\in S} a \boxdot \mathfrak{a} := \square(a\cdot\mathfrak{a}).$

Then all vectoid properties of $\{V,R\}$ also hold for the resulting structure in $\{S,T\}$. In particular

(a) the property (Vi) in S is a consequence of (Vi) in V, for $i = 1,2,3,4,5$,

(b) (VDi) in S is a consequence of (VDi) in V, for $i = 1,2,3,4,5$,

(c) (OVi) in S is a consequence of (OVi) in V, for $i = 1,2,3,4,5$,

(d) from (RG) and (Ri) follows (RGi) in S, for $i = 1,2,3,4$ and for all inner and outer operations.

Proof. The proof of this theorem is obtained in a straightforward manner. We omit it here. In order to gain familiarity with the many properties we recommend it be worked through in complete detail. ■

This theorem permits the definition of arithmetic by means of semimorphisms in all rows of Figure 3.5. As an example, one may again interpret R as the linearly ordered real numbers, S and D as the subsets of single and double precision floating-point numbers of a given computer. The arithmetic and structure in the product sets listed in Figure 3.5 are defined by the theorems displayed on the connecting lines.

One may also extend the chain $R \supseteq D \supseteq S$ by means of progressively coarser symmetric screens. Theorem 3.11 then shows that one again obtains vectoids in all rows of Figure 3.5. In this sense the vectoid structures in all rows of Figure 3.5 are invariant with respect to semimorphisms.

We mention finally that the arithmetic defined by semimorphisms in the sets listed in Figure 3.5 is in principle different from that defined in the same sets by the traditional method of defining arithmetic in the product sets. In both cases the resulting structures are vectoids although the arithmetic operations differ. Within the structures in the rows beginning with $\{M_{mn}R, R, \leq\}$ and $\{V_nR, R, \leq\}$, however, both methods lead to the same operations.

3.5 A Remark About Roundings

The results of this chapter show the importance of monotone and antisymmetric roundings. When composed into a semimorphism, they leave invariant the ringoid and vectoid structure in all numerically interesting cases.

Besides the monotone and antisymmetric roundings the monotone directed roundings $\bigtriangledown$ and $\bigtriangleup$ are of particular importance. We recall Theorem 1.30, which asserts that in a linearly ordered set every monotone rounding can be expressed by means of the monotone downwardly and upwardly directed roundings $\bigtriangledown$ and $\bigtriangleup$. Employing Theorem 3.4 in the case of a linearly ordered ringoid, we see that even the rounding $\bigtriangleup$ can be expressed by $\bigtriangledown$ and conversely. This leads to the result that in a linearly ordered ringoid (e.g., the real numbers) every monotone rounding can be expressed by the monotone downwardly directed rounding $\bigtriangledown$ alone. Apart from $\bigtriangleup$, no other monotone rounding has this property.

This implies that on a computer, in principle, every monotone rounding can be performed if either the monotone downwardly or upwardly directed rounding $\bigtriangledown$ or $\bigtriangleup$ is available. Since non-monotone roundings are of little interest, this points out the polytone role of the monotone directed roundings $\bigtriangledown$ and $\bigtriangleup$ for all rounded computations.

Beyond this result, the monotone directed roundings $\bigtriangledown$ and $\bigtriangleup$ are needed for all interval computations as we shall see in Chapter 4. Interval arithmetic is a fundamental extension to floating-point arithmetic. It brings mathematics and guarantees into computing. Despite these basic facts, the monotone downwardly and upwardly directed roundings $\bigtriangledown$ and $\bigtriangleup$ and the corresponding arithmetic operations defined by (RG) are not yet made conveniently available by hardware facilities and by programming languages on computers.

We may interpolate the observation that the monotone downwardly directed rounding $\bigtriangledown$, for instance, is not at all difficult to implement. If negative numbers are rep-

$$\begin{tikzcd}
R \arrow[rr, "\text{Theorem 3.7}"] \arrow[dr, "\text{Theorem 2.12}"] \arrow[ddr, "\text{Theorem 2.13}"] \arrow[dddr, "\text{Theorem 2.14}"] \arrow[ddddr, "\text{Theorem 2.15}"] \arrow[ddddd, "\text{Theorem 2.7}"] & & D \arrow[r, "\text{Theorem 3.7}"] & S \\
& \{M_{mn}R, R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_{mn}D, D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_{mn}S, S, \leq\} \\
& \{V_nR, R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_nD, D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_nS, S, \leq\} \\
& \{M_nR, R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_nD, D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_nS, S, \leq\} \\
& \{V_nR, M_nR, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_nD, M_nD, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_nS, M_nS, \leq\} \\
\mathbb{C}R \arrow[dr, "\text{Theorem 2.12}"] \arrow[ddr, "\text{Theorem 2.13}"] \arrow[dddr, "\text{Theorem 2.14}"] \arrow[ddddr, "\text{Theorem 2.15}"] & & & \\
& \{M_{mn}\mathbb{C}R, \mathbb{C}R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_{mn}\mathbb{C}D, \mathbb{C}D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_{mn}\mathbb{C}S, \mathbb{C}S, \leq\} \\
& \{V_n\mathbb{C}R, \mathbb{C}R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_n\mathbb{C}D, \mathbb{C}D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_n\mathbb{C}S, \mathbb{C}S, \leq\} \\
& \{M_n\mathbb{C}R, \mathbb{C}R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_n\mathbb{C}D, \mathbb{C}D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{M_n\mathbb{C}S, \mathbb{C}S, \leq\} \\
& \{V_n\mathbb{C}R, M_n\mathbb{C}R, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_n\mathbb{C}D, M_n\mathbb{C}D, \leq\} \arrow[r, "\text{Theorem 3.11}"] & \{V_n\mathbb{C}S, M_n\mathbb{C}S, \leq\}
\end{tikzcd}$$

Figure 3.5. The definition of computer arithmetic by semimorphisms leading to vectoids.

resented by the b-complement within the rounding procedure, then for all $x \in R$ the rounding $\bigtriangledown$ is identical with the truncation of the representation to finite length. Truncation is the simplest and fastest rounding procedure.

3.6 Uniqueness of the Minus Operator

In Chapter 2 it was shown that passage to the power set, complexification, and to the matrices over a ringoid, leads straight back to ringoids. In Chapter 3 it was shown that all these ringoids are invariant with respect to semimorphism. In Chapter 4 we shall extend this statement to several interval structures.

The uniqueness of the minus operator is essential for the structure and properties of ringoids and vectoids. Proof that the minus operator is unique was an essential part in the establishment of all the derived ringoids mentioned in the last paragraph.

In a ringoid the minus operator is defined by the four properties (D5a,b,c,d). It is certainly an interesting question, whether the uniqueness of the minus operator can be established by the property (D5a) $(e + x = o)$ alone. We show by a simple example that in general this is not the case. For that purpose we consider the mapping of the real numbers $\mathbb{R}$ into a simple floating-point system.

A normalized floating-point number is a real number of the form

$$x = m \cdot b^e.$$

Here m is the mantissa, b is the base of the number system in use and e is the exponent. b is an integer greater than unity. The exponent is an integer between two fixed integer bounds $e1$, $e2$, and in general $e1 < 0 < e2$. The mantissa is of the form

$$m = \circ \sum_{i=1}^{r} d_i \cdot b^{-i},$$

where $\circ \in \{+, -\}$ is the sign of the number.

The d_i are the digits of the mantissa. They have the property $d_i \in \{0, 1, \ldots, b-1\}$ for all $i = 1(1)r$ and $d_1 \neq 0$. Without the condition $d_1 \neq 0$, floating-point numbers are said to be unnormalized. The set of normalized floating-point numbers does not contain zero. So zero is adjoined to S. For a unique representation of zero it is often assumed that $m = 0.00\ldots0$ and $e = 0$. A floating-point system depends on the constants b, r, $e1$, and $e2$. We denote it by $S = S(b, r, e1, e2)$.

The floating-point numbers are not equally spaced between successive powers of b and their negatives. This spacing changes at every power of b. In particular, there are relatively large gaps around zero which contain no further floating-point number. Figure 3.6 shows a simple floating-point system $S = S(2, 3, -1, 2)$ consisting of 33 elements.

e	2^e	m_1	m_2	m_3	m_4
-1	$\frac{1}{2}$	0.100	0.101	0.110	0.111
0	1	0.100	0.101	0.110	0.111
1	2	0.100	0.101	0.110	0.111
2	4	0.100	0.101	0.110	0.111

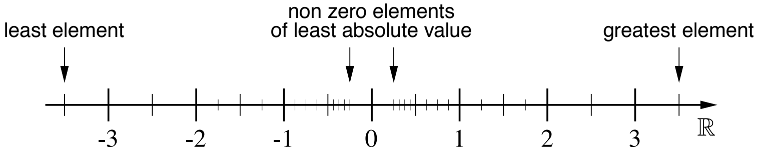


Figure 3.6. The characteristic spacing of a floating-point system.

In Figure 3.6 the mantissas are shown in binary representation.

If, for instance, the rounding towards zero is chosen, the entire interval $(-\frac{1}{4}, \frac{1}{4})$ is mapped onto zero. By Theorem 3.7 the semimorphism with this rounding generates a ringoid in S. In this ringoid we have, of course,

$$e \boxplus x = 1 \boxplus (-1) \underset{\text{(RG)}}{=} \square(1 + (-1)) = \square o \underset{\text{(R1)}}{=} o,$$

i.e., (D5a) holds. But we also obtain

$$1 \boxplus \left(-\frac{7}{8}\right) := \square \left(\frac{1}{8}\right) = 0,$$

i.e., (D5a) also holds for $-\frac{7}{8}$. In this case, however, (D5b) is not valid:

$$\left(-\frac{7}{8}\right) \boxdot \left(-\frac{7}{8}\right) = \square \left(\frac{49}{64}\right) = \square(0.765625) = \frac{3}{4} \neq 1.$$

Since $o \in S$, (R1) and (RG) yield immediately that for every $a \in S$ the element $-a$ is an additive inverse of a in S:

$$a \boxplus (-a) \underset{\text{(RG)}}{:=} \square(a + (-a)) = \square o \underset{\text{(R1)}}{=} o, \text{ for all } a \in S.$$

However, in a normalized floating-point system $-a$ is not in general the only additive inverse of a in S. This was already shown by the example above. An affirmative second example is: $\frac{1}{4} \boxplus (-\frac{3}{8}) = \square(-\frac{1}{8}) = 0$.

So $\frac{1}{4}$ and $-\frac{3}{8}$ form a pair of additive inverses. More generally, whenever the real sum of two numbers in S falls into the interval $(-\frac{1}{4}, \frac{1}{4})$ their sum $a \boxplus b$ in S is zero.

We shall derive necessary and sufficient conditions for the existence of unique additive inverses in S under semimorphism in the next section. Under these conditions the uniqueness of the minus operator in S is, of course, already a consequence of (D5a).

3.7 Rounding Near Zero

Theorem 3.12. *If S is a symmetric, discrete screen of $\mathbb{R}$, $\square : \mathbb{R} \to S$ a semimorphism, and $\epsilon > o$ the least distance between distinct elements of S, then for all $a \in S$ the element $b = -a$ is the unique additive inverse of a if and only if*

$$\square^{-1}(o) \subseteq (-\epsilon, \epsilon). \tag{3.7.1}$$

Here $\square^{-1}(o)$ denotes the inverse image of o and $(-\epsilon, \epsilon)$ is the open interval between $-\epsilon$ and ϵ.

Proof. At first we show that (3.7.1) is sufficient: We assume that $b \neq -a$ is an additive inverse of a in S. Then $a \boxplus b = o$ and by (RG) $a+b \in \square^{-1}(o)$. This means by (3.7.1)

$$-\epsilon < a + b < \epsilon. \tag{3.7.2}$$

Since $-b \neq a$ we have by definition of $\epsilon : |a - (-b)| \geq \epsilon$. This contradicts (3.7.2), so the assumption is false. Under the condition (3.7.1) there is no additive inverse of a in S other than $-a$. In other words (3.7.1) is sufficient for the uniqueness of the additive inverse $-a$ in S.

Now we show that (3.7.1) is necessary also: Since $o \in S$ by (R1) $\square(o) = o$ and $o \in \square^{-1}(o)$. Since $\square$ is monotone, $\square^{-1}(o)$ is convex. Since $\square$ is antisymmetric, $\square^{-1}(o)$ is symmetric with respect to zero. Assuming that (3.7.1) is not true, then $\square^{-1}(o) \supset (-\epsilon, \epsilon)$. We take two elements $a, b \in S$ with distance ϵ. Then $a \neq b$ and $|a - b| = \epsilon$, i.e., $a + (-b) \in \square^{-1}(o)$ or $a \boxplus (-b) = o$. This means that $-b \neq -a$ is inverse to a. Thus (3.7.1) is necessary because otherwise there would be more than one additive inverse to a in S. ■

Equation (3.7.1) holds automatically if the number ϵ itself is an element of S. Then because of (R1), $\square(\epsilon) = \epsilon$ and, because of the monotonicity of the rounding (R2), $\square^{-1}(o) \subseteq (-\epsilon, \epsilon)$. In other words: If the least distance ϵ of two elements of a discrete screen S of $\mathbb{R}$ is an element of S, then for all $a \in S$ the element $-a \in S$ is the unique additive inverse of a in S. This holds under the assumption that the mapping $\square : \mathbb{R} \to S$ is a semimorphism.

Such subsets of the real numbers do indeed occur. For instance, the integers or the so-called fixed-point numbers are subsets of $\mathbb{R}$ with this property. Sometimes the normalized floating-point numbers are extended into a set with this property. Such a set is obtained if in the case $e = e1$ unnormalized mantissas are permitted. Then ϵ itself becomes an element of S and for all $a \in S$ the element $-a$ is the unique additive inverse of a. This is the case, for instance, if IEEE arithmetic with denormalized numbers is implemented.

Figure 3.7 illustrates the behavior of typical roundings in the neighborhood of zero. (R1) means that for floating-point numbers the rounding function coincides with the identity mapping.

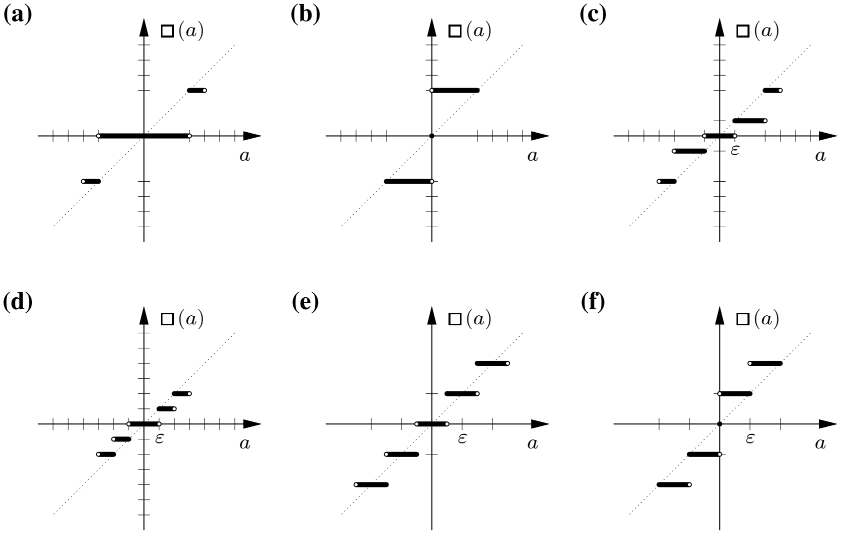


Figure 3.7. The behavior of frequently used roundings near zero.

(a) shows the conventional behavior of the rounding of normalized floating-point numbers. In this case (3.7.1) does not hold and the additive inverse is not unique.

(b) shows the rounding away from zero of normalized floating-point numbers. In this case (3.7.1) holds and we have unique additive inverses.

(c) here and in the following cases ϵ is an element of S and we have unique additive inverses.

(d) shows the rounding towards zero near zero when denormalized numbers are permitted for $e = e1$. In the IEEE arithmetic standard this is called gradual underflow.

(e) shows the rounding to the nearest number in S in the neighborhood of zero. The roundings (d) and (e) are provided by the IEEE arithmetic standard.

(f) shows the rounding away from zero in the neighborhood of zero with denormalized numbers permitted for $e = e1$. This rounding has all the required properties. It is $\square(a) = o$ if and only if $a = o$, a property which can be very important for making a clear distinction between a number that is zero and a number that is not, however small it might actually be. This rounding is not provided by the IEEE arithmetic standard.

For the developments in the next chapter it is of interest whether Theorem 3.12 can be generalized in such a way that it becomes applicable to the product spaces that occur in Figure 1 and in particular to those rows which contain intervals. In all interval spaces of Figure 1 arithmetic will be defined by semimorphism and the resulting structures will be ringoids and vectoids again. The proof that the minus operator is unique has been intricate in all cases of ringoids in Chapters 2 and 3. The following theorem gives a criterion for the uniqueness of additive inverses. It contains Theorem 3.12 as a special case.

Theorem 3.13. *Let* $\{R, +, \cdot\}$ *be a ringoid and* $\{R, d\}$ *a metric space with a distance function* $d : R \times R \to \mathbb{R}$ *which is* translation invariant*, i.e., which has the property*

$$\bigwedge_{a,b,c \in R} d(a+c, b+c) = d(a,b) \qquad \textit{(translation invariance).} \tag{3.7.3}$$

Let $\{\overline{R}, \leq\}$ *be a complete lattice,* $\{\overline{S}, \leq\}$ *a symmetric lower (resp. upper) screen of* $\overline{R}$ *which is discrete, i.e., which has the property*

$$\bigwedge_{a,b \in S} (a \neq b \Rightarrow d(a,b) \geq \epsilon > o),$$

where $\epsilon > o$ *is the least distance of distinct elements of* S*. Further, let* $\square : \overline{R} \to \overline{S}$ *be a semimorphism.*

(a) If a *is an element of* S *which has a unique additive inverse* $-a$ *in* R*, then* $-a$ *is also the unique additive inverse of* a *in* S *if*

$$\bigwedge_{x \in R} (\square x = o \Rightarrow d(x, o) < \epsilon). \tag{3.7.4}$$

(b) If in addition $\{R, +\}$ *is a group, then* (3.7.4) *is necessary also.*

Proof. From (RG) and (R1) follows that an element $a \in S$ which has an additive inverse $-a$ in R has the same additive inverse in S:

$$a \boxplus (-a) := \square(a + (-a)) = \square o = o. \tag{3.7.5}$$

(a) We show that (3.7.4) is sufficient for uniqueness of $-a$: We assume that $b \neq -a$ is an additive inverse of a in S. Then we obtain by (RG) and (3.7.4):

$$a \boxplus b = o \Rightarrow \square(a+b) = o \Rightarrow d(a+b, o) < \epsilon. \tag{3.7.6}$$

On the other hand we get by the definition of ϵ and by the translation invariance (3.7.3):

$$d(b, -a) \geq \epsilon \Rightarrow d(a+b, a+(-a)) = d(a+b, o) \geq \epsilon.$$

This contradicts (3.7.6). So the assumption that there is an additive inverse b of a in S other than $-a$ is false. In other words (3.7.4) is sufficient for the uniqueness of the additive inverse $-a$ in S.

(b) We show that under the additional assumption that $\{R, +\}$ is a group, (3.7.4) is necessary also: By (R1) we obtain $o \in \square^{-1}(o)$. As a consequence of (R2) $\square^{-1}(o)$ is convex and by (R4) $\square^{-1}(o)$ is symmetric, since for all $a \in R$ with

$$a \in \square^{-1}(o) \Rightarrow \square a = o \Rightarrow \square(-a) \underset{\text{(R4)}}{=} -\square(a) = o \Rightarrow -a \in \square^{-1}(o).$$

Thus $\square^{-1}(o)$ is a symmetric and convex set. By (3.7.4) all elements of this set have a distance to zero which is less than ϵ: $d(x, o) < \epsilon$. Should (3.7.4) not hold, there would be elements $x \in \square^{-1}(o)$ with $d(x, o) \geq \epsilon > o$. ϵ is the least distance between distinct elements of S. Now we chose two different elements $a, b \in S$ with distance ϵ. Then

$$a \neq b \wedge d(a, b) = \epsilon \underset{(3.7.3)}{\Rightarrow} d(a - b, o) = \epsilon \Rightarrow a - b \in \square^{-1}(o) \Rightarrow a \boxplus (-b) = o.$$

This means $-b$ is inverse to a and $-b \neq -a$. In other words: If (3.7.4) does not hold there are elements in S which have more than one additive inverse. This shows that (3.7.4) is necessary. ■

By Theorem 3.13(a), (3.7.4) is a sufficient criterion for the uniqueness of additive inverses in columns 3 and 4 of Figure 1 for rows 1, 3, 4, 6, 7, 9, 10, and 12. We mention explicitly that in these cases under condition (3.7.4) the resulting structure is that of a ringoid and the uniqueness of the minus operator is a consequence of (D5a) alone.

Theorem 3.13(b) shows that (3.7.4) is also necessary for the uniqueness of additive inverses in columns 3 and 4 of Figure 1 for rows 1, 3, 7, and 9. In these cases $\{R, +\}$ is a group.

To fully establish the applicability of Theorem 3.13 we still have to demonstrate that in all the basic spaces under consideration a metric does indeed exist which is translation invariant (3.7.3). We just mention the appropriate metric and leave the demonstration of (3.7.3) to the reader:

- If R is the set of real numbers $\mathbb{R}$, then $d(a, b) = |a - b|$.
- If R is the set of complex numbers $\mathbb{C}$, the distance of two complex numbers $a = a_1 + ia_2$ and $b = b_1 + ib_2$ is defined by $d(a, b) := |a_1 - b_1| + |a_2 - b_2|$.
- If R is the set of real intervals $I\mathbb{R}$, the distance of two intervals $a = [a_1, a_2]$ and $b = [b_1, b_2]$ is defined by $d(a, b) := \max(|a_1 - b_1|, |a_2 - b_2|)$.
- If R is the set of complex intervals $I\mathbb{C}$, the distance of two complex intervals $a = a_1 + ia_2$ and $b = b_1 + ib_2$ with real intervals a_1, a_2, b_1, and b_2 is defined by $d(a, b) := d(a_1, b_1) + d(a_2, b_2)$.
- For two matrices $a = (a_{ij})$ and $b = (b_{ij})$ with components a_{ij}, b_{ij} of $\mathbb{R}$, $\mathbb{C}$, $I\mathbb{R}$, or $I\mathbb{C}$, the distance is defined in each case as the maximum of the distances of corresponding matrix components: $d(a, b) := \max(d(a_{ij}, b_{ij}))$.

In all basic structures under consideration, which are the sets $\mathbb{R}$, $\mathbb{C}$, $I\mathbb{R}$, $I\mathbb{C}$, and the matrices with components of these sets, (see Figure 1), the multiplicative unit e has a unique additive inverse $-e$. Under the hypotheses of Theorem 3.12 and Theorem 3.13 (conditions (3.7.1) and (3.7.4)) respectively, therefore, $-e$ is also the unique additive inverse in any computer representable subset. This allows the definition of the minus operator and of subtraction in the computer representable subsets in the same manner as it was done in ringoids with all its consequences by (D5a) alone.

We did not follow this line in Chapter 3. The reason for this is the fact that there are reasonable semimorphisms which are frequently used in practice and for which the conditions (3.7.1) and/or (3.7.4) do not hold. An example of this is the rounding towards zero from the real numbers into a floating-point screen S and we have seen in Section 3.6 that elements of S may have more than one additive inverse. In particular there may be more than one additive inverse for the multiplicative unit e in S.

At this stage of the development a look ahead at interval spaces is particularly interesting. We shall deal with these spaces in the next chapter and we shall meet a different situation there. Again we consider the sets $\mathbb{R}$ and $\mathbb{C}$ as well as the matrices $M\mathbb{R}$ and $M\mathbb{C}$ with components of $\mathbb{R}$ and $\mathbb{C}$, respectively. All these sets are ordered with respect to the order relation $\leq$. If R denotes any one of these sets, an interval is defined by

$$A = [a_1, a_2] := \{a \in R \mid a_1, a_2 \in R, a_1 \leq a \leq a_2\}.$$

Arithmetic operations on the set *IR* of all such intervals will be defined in the next chapter. Every interval $[a, a] \in$ *IR* with $a \in R$ has a unique additive inverse $[-a, -a]$ in *IR* for all $R \in \{\mathbb{R}, \mathbb{C}, M\mathbb{R}, M\mathbb{C}\}$. However, the elements of *IR* in general are not computer representable and the arithmetic operations in *IR* are not computer executable. Therefore, subsets $S \subseteq R$ of computer representable elements have to be chosen. An interval in *IS* is defined by[2]

$$A = [a_1, a_2] := \{a \in R \mid a_1, a_2 \in S, a_1 \leq a \leq a_2\}.$$

Arithmetic operations in *IS* are defined by semimorphism, i.e., by (RG) with the monotone and antisymmetric rounding $\diamondsuit$: *IR* $\rightarrow$ *IS* which is upwardly directed with respect to set inclusion as an order relation. $\diamondsuit$ is uniquely defined by these properties (R1,2,3,4). See Chapter 1. This process leads to computer executable operations in the interval spaces *IS* where S is a computer representable subset of anyone of the sets $\mathbb{R}$, $\mathbb{C}$, $M\mathbb{R}$, and $M\mathbb{C}$.

In both theorems of this section the inverse image of zero with respect to a rounding plays a key role. Since the rounding $\diamondsuit$: *IR* $\rightarrow$ *IS* is upwardly directed with respect to set inclusion as an order relation the inverse image of zero $\diamondsuit^{-1}(o)$ can only be zero itself. Thus the criteria (3.7.4) for the existence of unique additive inverses evidently holds in *IS*. This establishes the fact, with all its consequences, that the unit interval $[e, e]$ has a unique additive inverse $[-e, -e]$ in *IS*. This holds for all interval sets IS where S is a discrete subset of $R \in \{\mathbb{R}, \mathbb{C}, M\mathbb{R}, M\mathbb{C}\}$. This will be the way we shall prove the uniqueness of the minus operator in the next chapter.

[2] Note, that although the interval limits a_1 and a_2 are elements of S, the interior points are elements of R.

Chapter 4
Interval Arithmetic

We start with the observation made in Chapter 2, that all the spaces listed in the leftmost element in every row in Figure 1 are ringoids or vectoids. In particular, the spaces that are power sets are inclusion-isotonally ordered with respect to inclusion as an order relation. We shall define the operations in all interval spaces in Figure 1 by means of semimorphisms. With these operations, each interval space becomes an inclusion-isotonally ordered monotone upper screen ringoid (resp. vectoid) of its left neighbor in Figure 1. Furthermore, these ringoids and vectoids are ordered or weakly ordered with respect to the order relation $\leq$, which we shall define below.

For all operations to be defined in the interval sets listed in Figure 1, we derive explicit formulas for the computation of the resulting interval in terms of the bounds of the interval operands.

To do this, we show that the structures M_nIR and IM_nR as well as $\{V_nIR, IR, \leq\}$ and $\{IV_nR, IR, \leq\}$ – if the operations in M_nIR and V_nIR are defined by the traditional method – are isomorphic with respect to all algebraic operations and the order relation $\leq$. The same isomorphisms are shown to exist between the corresponding spaces, wherein R is replaced by its complexification $\mathbb{C}R$.

The operations in all interval spaces in the columns listed under D and S in Figure 1 are also defined via semimorphisms. The isomorphisms referred to here are the main tool for obtaining optimal (i.e., the smallest appropriate interval) and computer executable formulas for the operations in these spaces.

4.1 Interval Sets and Arithmetic

Let $\{R, \leq\}$ be an ordered set and $\mathbb{P}R$ the power set of R. The intervals

$$[a_1, a_2] := \{x \in R \mid a_1, a_2 \in R, a_1 \leq x \leq a_2\} \text{ with } a_1 \leq a_2$$

are special elements contained in the power set $\mathbb{P}R$. Thus, denoting the set of all intervals over R by IR, we have $IR \subset \mathbb{P}R$. In IR we define equality and the order relation $\leq$ by

$$\begin{aligned} [a_1, a_2] = [b_1, b_2] \quad &:\Leftrightarrow a_1 = b_1 \wedge a_2 = b_2, \\ [a_1, a_2] \leq [b_1, b_2] \quad &:\Leftrightarrow a_1 \leq b_1 \wedge a_2 \leq b_2. \end{aligned}$$

With these definitions, we formulate the following theorem.

Theorem 4.1. *If $\{R, \le\}$ is a complete lattice, then $\{IR, \le\}$ is also a complete lattice. Moreover, with $A = [a_1, a_2]$*

$$\bigwedge_{S \subseteq \{IR, \le\}} \left(\inf S = [\inf_{A \in S} a_1, \inf_{A \in S} a_2] \wedge \sup S = [\sup_{A \in S} a_1, \sup_{A \in S} a_2] \right). \tag{4.1.1}$$

Proof. It is clear that $\{IR, \le\}$ is an ordered set. Since IR is a subset of the product set $R \times R$ and the order relation is defined componentwise, Theorem 1.10 implies that $\{IR, \le\}$ is a complete lattice and that (4.1.1) holds. ■

We remark that within $\mathbb{P}R$ the corresponding definition

$$A \le B :\Leftrightarrow \inf_{a \in A} a \le \inf_{b \in B} b \wedge \sup_{a \in A} a \le \sup_{b \in B} b,$$

does not lead to an order relation because (O3) is obviously not valid.

If $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty\} \cup \{+\infty\}$, then the interval $[-\infty, -\infty]$ is the least element and the interval $[+\infty, +\infty]$ is the greatest element of $\{I\overline{\mathbb{R}}, \le\}$.

If $\{R, \le\}$ is a complete lattice, then R is the greatest element of the power set $\{\mathbb{P}R, \subseteq\}$. Writing R in the form $[o(R), i(R)]$, we see that it is also the greatest element of $\{IR, \subseteq\}$. The empty set $\emptyset$ is the least element of $\{\mathbb{P}R, \subseteq\}$. According to the above definition, however, $\emptyset$ is not an element of IR. To have a least element available whenever necessary, we adjoin the empty set to IR and define

$$\overline{IR} := IR \cup \{\emptyset\}.$$

In the following theorem we show that $\overline{IR}$ serves as an upper screen of $\mathbb{P}R$ with respect to inclusion as an order relation.

Theorem 4.2. *If $\{R, \le\}$ is a complete lattice, then*

(a) $\{IR, \subseteq\}$ is a conditionally completely ordered set. For all $A \subseteq IR$, which are bounded below, we have with $B = [b_1, b_2] \in IR$

$$\inf_{\{IR, \subseteq\}} A = [\sup_{B \in A} b_1, \inf_{B \in A} b_2] \wedge \sup_{\{IR, \subseteq\}} A = [\inf_{B \in A} b_1, \sup_{B \in A} b_2],$$

i.e., the infimum is the intersection, and the supremum is the interval hull of the elements of A.

(b) $\{\overline{IR}, \subseteq\}$ is an upper screen of $\{\mathbb{P}R, \subseteq\}$.

Proof. (a) Since $\{R, \le\}$ is a complete lattice and the set A is bounded below (i.e., A contains an interval), the intervals

$$I := [\sup_{B \in A} b_1, \inf_{B \in A} b_2], \quad S := [\inf_{B \in A} b_1, \sup_{B \in A} b_2]$$

exist, and for $B \in A$ we have $I \subseteq B \wedge B \subseteq S$, i.e., I is a lower and S an upper bound of A. If I_1 is any lower and S_1 any upper bound of A, then $I_1 \subseteq I$ and $S \subseteq S_1$, i.e., I is the greatest lower and S the least upper bound of A.

(b) Using (a), we see that $\{\overline{IR}, \subseteq\}$ is a complete lattice and therefore a complete subnet of $\{\mathbb{P}R, \subseteq\}$. We still have to show that for every subset $A \subseteq \overline{IR}$, $\inf_{\overline{IR}} A = \inf_{\mathbb{P}R} A$.

If A is bounded below in IR, then by (a) the infimum in IR is the intersection as it is in $\mathbb{P}R$, and therefore $\inf_{IR} A = \inf_{\overline{IR}} A = \inf_{\mathbb{P}R} A$.

If A is not bounded below in IR, we consider three cases:

(1) $A = \emptyset$. Then $\inf_{\overline{IR}} A = \inf_{\mathbb{P}R} A = i(\mathbb{P}R) = R$.

(2) $A \neq \emptyset \wedge \emptyset \in A$. Then $\inf_{\overline{IR}} A = \emptyset = \inf_{\mathbb{P}R} A$.

(3) $\emptyset \notin A$. Since A is not bounded below in IR, we obtain once again $\inf_{\overline{IR}} A = \emptyset = \inf_{\mathbb{P}R} A$. ■

Now we consider the monotone upwardly directed rounding $\square : \mathbb{P}R \to \overline{IR}$, which is defined by the following properties:

(R1) $\bigwedge_{A \in \overline{IR}} \square A = A$,

(R2) $\bigwedge_{A,B \in \mathbb{P}R} (A \subseteq B \Rightarrow \square A \subseteq \square B)$,

(R3) $\bigwedge_{A \in \mathbb{P}R} A \subseteq \square A$.

Remark 4.3. At this stage we are facing a notational problem. The theory that is being developed should be applicable to realistic models. These models can usually be characterized as conditionally completely ordered algebraic structures. We have seen in Section 1.1 of Chapter 1 that every conditionally completely ordered set R can be made into a complete lattice by adjoining a least element $o(R)$ and a greatest element $i(R)$. Then $\overline{R} := R \cup \{o(R)\} \cup \{i(R)\}$ is a complete lattice. This completion will be necessary for the process of rounding. A rounding was defined as a mapping of a complete lattice into a lower (resp. upper) screen. However, the adjoined elements $o(R)$ and $i(R)$ often do not satisfy all algebraic properties of the model. ■

As an example the real numbers $\mathbb{R}$ are defined as a conditionally complete, linearly ordered field. Derived spaces such as those in the second column of Figure 1 are also only conditionally complete with respect to the order relation $\leq$. If the real numbers are completed in the customary manner by adjoining a least element $-\infty$ and a greatest element $+\infty$ the resulting set $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty\} \cup \{+\infty\}$ is a complete lattice. Arithmetic operations are frequently defined for the new elements, as for instance:

$$\begin{aligned} \infty + x &= \infty, & -\infty + x &= -\infty, \\ -\infty + (-\infty) = (-\infty) \cdot \infty &= -\infty, & \infty + \infty = \infty \cdot \infty &= \infty, \\ \infty \cdot x = \infty \text{ for } x &> 0, & \infty \cdot x = -\infty \text{ for } x &< 0, \\ \frac{x}{\infty} = \frac{x}{-\infty} &= 0, \end{aligned}$$

together with variants obtained by applying the sign rules and the law of commutativity. However, the new elements $-\infty$ and $+\infty$ fail to satisfy several of the algebraic properties of a field. For example $a + \infty = b + \infty$ even if $a < b$, so that the cancellation law is not valid. Also, values like $\infty - \infty$ or $0 \cdot \infty$ are not defined for the new elements.

A similar situation occurs in other models. We have seen that the power set $\mathbb{P}R$ of a set R is a complete lattice with respect to set inclusion as an order relation. In Theorem 2.5 it was shown that ringoid properties are preserved upon passage to the power set. The empty set is the least element in $\mathbb{P}R$. If operations for the empty set are defined the result can only be the empty set. Thus we have $\{0\} \cdot \varnothing = \varnothing \cdot \{0\} = \varnothing \neq \{0\}$. This is not in accordance with (D4), i.e., the empty set does not fulfil the ringoid properties in $\mathbb{P}R$.

In Theorem 4.2 it was shown that the completed set $\overline{IR} := IR \cup \{\varnothing\}$ of intervals serves as an upper screen of $\mathbb{P}R$. However, to obtain a ringoid in Theorem 4.5, the empty set has to be excluded again because of the property $[0,0] \cdot \varnothing = \varnothing \cdot [0,0] = \varnothing \neq [0,0]$.

A mathematically strict notation, therefore, would have to distinguish between a set R and its lattice theoretic completion $\overline{R}$. In derived sets this may lead to very complicated notations. If, for instance, R is a conditionally completely ordered ringoid and $\overline{R}$ its lattice theoretic completion, the monotone upwardly directed rounding from the power set into the intervals would be a mapping $\square : \mathbb{P}\overline{R} \to \overline{I\overline{R}}$. In order to avoid such unreadable notations we shall assume henceforth that the basic set is already completely ordered. If this set is a ringoid or a vectoid, the reader should be aware, however, that the least and/or the greatest elements possibly do not satisfy all algebraic properties of a ringoid or a vectoid. In particular for power set or interval structures we shall in general not explicitly note that the empty set (the least element) does not satisfy the algebraic properties of a ringoid or vectoid.

Following this line we now develop properties of the monotone upwardly directed rounding $\square : \mathbb{P}R \to \overline{IR}$ in the following theorem.

Theorem 4.4. *Let* $\{R, +, \cdot, \leq\}$ *be a completely and weakly ordered ringoid with the neutral elements* 0 *and* e. *Then* *(a)* $\{\overline{IR}, \subseteq\}$ *is a symmetric upper screen of* $\{\mathbb{P}R, \subseteq\}$*, i.e., we have*

$$\textbf{(S3)} \quad [0,0],[e,e] \in IR \quad \wedge \bigwedge_{A=[a_1,a_2]\in IR} -A = [-a_2,-a_1] \in IR. \qquad (4.1.2)$$

(b) The monotone upwardly directed rounding $\square : \mathbb{P}R \to \overline{IR}$ *is antisymmetric, i.e.,*

$$\textbf{(R4)} \bigwedge_{\varnothing\neq A\in\mathbb{P}R} \square(-A) = -\square A.$$

(c) We have

$$\textbf{(R)} \bigwedge_{\varnothing\neq A\in\mathbb{P}R} \square A = \inf(U(A)\cap IR) = o(U(A)\cap IR) = \Big[\inf_{a\in A} a, \sup_{a\in A} a\Big].$$

Proof. The assertions are shown in the order (a), (c), (b).

(a) With $A := [a_1,a_2] := \{x \in R \mid a_1 \leq x \leq a_2\}$, we get $-A := \{-e\}\cdot A := \{(-e)\cdot x \mid x \in A\} = \{-x \in R \mid a_1 \leq x \leq a_2\} = \{x \in R \mid a_1 \leq -x \leq a_2\} \underset{(\mathrm{OD2})_R}{=} \{x \in R \mid -a_2 \leq x \leq -a_1\} = [-a_2,-a_1] \in IR$.

(c) Theorem 1.24 implies $\square A = \inf(U(A)\cap \overline{IR})$. We show that $\square A = [\inf A, \sup A]$. For all $A \in \mathbb{P}R$ such that $A \neq \varnothing$ we have $A \subseteq [\inf A, \sup A]$. For all $B = [b_1,b_2] \in IR$ with the property $A \subseteq B$ we obtain for all $a \in A$: $b_1 \leq a \leq b_2 \Rightarrow b_1 \leq \inf A \wedge \sup A \leq b_2 \Rightarrow [\inf A, \sup A] \subseteq [b_1,b_2]$, i.e., $[\inf A, \sup A]$ is the least upper bound of A in $\{IR, \subseteq\}$, which proves (c).

(b) With $\square A = [\inf A, \sup A]$, we get $\square(-A) = [\inf(-A), \sup(-A)]$, and with Theorem 2.3$(y)$, $\square A = [-\sup A, -\inf A]$. Now with (b) and (c) we obtain $\square(-A) = -\square(A)$. ■

Now we make use of the monotone upwardly directed rounding $\square : \mathbb{P}R \to \overline{IR}$ to define operations $\boxdot$, $\circ \in \{+,-,\cdot,/\}$, in IR by the corresponding semimorphism that has the following property:

$$\textbf{(RG)} \quad \bigwedge_{A,B\in IR} A \boxdot B := \square(A\circ B) = \Big[\inf_{a\in A, b\in B}(a\circ b), \sup_{a\in A, b\in B}(a\circ b)\Big]. \quad (4.1.3)$$

Here the infimum and supremum are taken over $a \in A$, $b \in B$, while the equality on the right-hand side is a simple consequence of (R). These operations have the following properties to which by Theorem 1.32 (RG) is equivalent:

$$\textbf{(RG1)} \bigwedge_{A,B\in IR} (A\circ B \in IR \Rightarrow A\boxdot B = A\circ B),$$

$$\textbf{(RG2)} \bigwedge_{A,B,C,D\in IR} (A\circ B \subseteq C\circ D \Rightarrow A\boxdot B \subseteq C\boxdot D),$$

$$\textbf{(RG3)} \bigwedge_{A,B\in IR} (A\circ B \subseteq A\boxdot B).$$

When in these formulas the operator $\circ$ represents division, we assume additionally that $B, D \notin \widetilde{N}$. For the definition of $\widetilde{N}$, see Theorem 4.5, which we now state.

Theorem 4.5. *(a) Let $\{R, +, \cdot, \leq\}$ be a completely and weakly ordered ringoid with neutral elements* 0 *and* e*. We assume additionally that $x = -e$ is already unique in R by (D5a) alone*[1]*. If operations are defined in IR by the semimorphism $\square : \mathbb{P}R \to \overline{IR}$, then $\{IR, \boxplus, \boxdot, \leq, \subseteq\}$ becomes a completely and weakly ordered ringoid with respect to the order relation $\leq$. The special elements are $[0, 0]$, $[e, e]$, and $[-e, -e]$. With respect to $\subseteq$, IR is an inclusion-isotonally ordered monotone upper screen ringoid of $\mathbb{P}R$. Moreover, for all $A = [a_1, a_2], B = [b_1, b_2] \in IR$, we have*

(A) $A \boxplus B = [a_1 + b_1, a_2 + b_2]$, $A \boxminus B = A \boxplus (-B) = [a_1 - b_2, a_2 - b_1]$.

(b) If additionally $\{R, +, \cdot, \leq\}$ is an ordered ringoid, then $\{IR, \boxplus, \boxdot, \leq, \subseteq\}$ is also an ordered ringoid with respect to $\leq$ and

(B) $A \geq 0 \wedge B \geq 0 \Rightarrow A \boxdot B = [a_1 b_1, a_2 b_2]$,

(C) $A \leq 0 \wedge B \leq 0 \Rightarrow A \boxdot B = [a_2 b_2, a_1 b_1]$,

(D) $A \leq 0 \wedge B \geq 0 \Rightarrow A \boxdot B = [a_1 b_2, a_2 b_1]$,
$A \geq 0 \wedge B \leq 0 \Rightarrow A \boxdot B = [a_2 b_1, a_1 b_2]$.

(c) If $\{R, N, +, \cdot, /, \leq\}$ is an ordered division ringoid, then $\{IR, \widetilde{N}, \boxplus, \boxdot, \boxslash, \leq\}$ with $\widetilde{N} := \{A \in IR \mid A \cap N \neq \emptyset\}$ is also an ordered division ringoid. Moreover, for all $A = [a_1, a_2] \in IR$ and $B = [b_1, b_2] \in IR \setminus \widetilde{N}$ we have

(E) $A \geq 0 \wedge 0 < b_1 \leq b_2 \Rightarrow A \boxslash B = [a_1/b_2, a_2/b_1]$,

(F) $A \geq 0 \wedge b_1 \leq b_2 < 0 \Rightarrow A \boxslash B = [a_2/b_2, a_1/b_1]$,

(G) $A \leq 0 \wedge 0 < b_1 \leq b_2 \Rightarrow A \boxslash B = [a_1/b_1, a_2/b_2]$,

(H) $A \leq 0 \wedge b_1 \leq b_2 < 0 \Rightarrow A \boxslash B = [a_2/b_1, a_1/b_2]$.

Proof. (a) Theorem 3.5 directly implies the properties (D1,2,3,4,5,7,8,9) and (OD5). It remains to show (A), (D6), (OD1), and (OD2). Let be $A = [a_1, a_2]$, $B = [b_1, b_2]$, $C = [c_1, c_2]$.

(A): We demonstrate the relation

$$\bigwedge_{A,B \in IR} (\inf A + \inf B = \inf(A + B) \quad \wedge \quad \sup A + \sup B = \sup(A + B)) \tag{4.1.4}$$

[1] For instance, this is the case if $\{R, +\}$ is a group.

which by (4.1.3) proves (A). We have

$$\bigwedge_{a\in A}\bigwedge_{b\in B}(\inf A \le a \wedge \inf B \le b \underset{(\mathrm{OD1})_R}{\Rightarrow} \inf A + \inf B \le a + b)$$
$$\Rightarrow \inf A + \inf B \le \inf(A+B).$$

On the other hand, $\inf(A+B) \le a_1 + b_1 = \inf A + \inf B$ and therefore by (O3) $\inf A + \inf B = \inf(A+B)$. The proof of the second property in (4.1.4) is dual.

(D6): $[e,e] \boxplus [x,y] \underset{(\mathrm{A})}{=} [e+x, e+y] = [0,0]$, i.e., $e+x=0$ and $e+y=0$. By assumption there is only one element in R that solves (D5a). Thus $x=y=-e$.

(OD1):

$$A \le B :\Leftrightarrow a_1 \le b_1 \wedge a_2 \le b_2 \underset{(\mathrm{OD1})_R}{\Rightarrow} a_1 + c_1 \le b_1 + c_1 \wedge a_2 + c_2 \le b_2 + c_2$$
$$\Rightarrow \inf A + \inf C \le \inf B + \inf C \wedge \sup A + \sup C \le \sup B + \sup C$$
$$\underset{(4.1.4)}{\Rightarrow} \inf(A+C) \le \inf(B+C) \wedge \sup(A+C) \le \sup(B+C)$$
$$\underset{(4.1.3)}{\Rightarrow} (A \boxplus C) \le (B \boxplus C).$$

(OD2):

$$A \le B :\Leftrightarrow a_1 \le b_1 \wedge a_2 \le b_2 \underset{(\mathrm{OD2})_R}{\Rightarrow} -b_1 \le -a_1 \wedge -b_2 \le -a_2$$
$$\Rightarrow [-b_2, -b_1] \le [-a_2, -a_1] \underset{(4.1.2)}{\Rightarrow} -B \le -A \Rightarrow \boxminus B \le \boxminus A.$$

(b) The proof of (OD3) in IR is a direct consequence of (OD3) in R.

(B): $A \ge 0 \wedge B \ge 0 \Rightarrow \bigwedge_{a\in A}\bigwedge_{b\in B} 0 \le \inf A \le a \wedge 0 \le \inf B \le b \underset{(\mathrm{OD3})_R}{\Rightarrow}$ $\inf A \cdot \inf B \le a \cdot b \Rightarrow \inf A \cdot \inf B \le \inf(A \cdot B)$.

On the other hand, $\inf(A \cdot B) \le a_1 \cdot b_1 = \inf A \cdot \inf B \underset{(\mathrm{O3})}{\Rightarrow} \inf A \cdot \inf B = \inf(A \cdot B)$.

(C): $A \le [0,0] \wedge B \le [0,0] \Rightarrow -A = [-a_2, -a_1] \ge [0,0] \wedge -B = [-b_2, -b_1] \ge [0,0] \underset{(\mathrm{B})}{\Rightarrow} A \boxdot B = [(-a_2)(-b_2), (-a_1)(-b_1)] = [a_2 b_2, a_1 b_1]$.

(D): These cases can again be reduced to (B) by changing the sign of those intervals whose bounds are less or equal to 0.

(c) (E): $A \ge [0,0] \wedge 0 < b_1 \le b_2 \Rightarrow \bigwedge_{a\in A}\bigwedge_{b\in B} 0 \le \inf A \le a \wedge 0 < b \le \sup B \underset{(\mathrm{OD4})_R}{\Rightarrow}$ $\inf A/\sup B \le a/\sup B \le a/b \Rightarrow \inf A/\sup B \le \inf(A/B)$.

On the other hand, $\inf(A/B) \le a_1/b_2 = \inf A/\sup B \Rightarrow \inf A/\sup B = \inf(A/B)$.

The proof of the second property is dual.

Properties (F), (G) and (H) can be reduced to (E) by changing the sign of those intervals whose bounds are less or equal to 0.

It remains to prove (OD4).

(OD4):

$$
\begin{aligned}
C \in IR \setminus \widetilde{N} \wedge C > [0,0] \Rightarrow 0 \notin C \qquad & \\
& \Rightarrow 0 \leq a_1 \leq b_1 \wedge 0 \leq a_2 \leq b_2 \wedge 0 < c_1 \leq c_2 \\
& \underset{(\text{OD4a})_R}{\Rightarrow} 0 \leq a_1/c_2 \leq b_1/c_2 \wedge 0 \leq a_2/c_1 \leq b_2/c_1 \\
& \underset{(\text{E})}{\Rightarrow} [0,0] \leq A \boxslash C \leq B \boxslash C.
\end{aligned}
$$

$$
\begin{aligned}
A, B \in IR \setminus \widetilde{N} \wedge [0,0] < A \leq B \qquad & \\
& \Rightarrow 0 < a_1 \leq b_1 \wedge 0 < a_2 \leq b_2 \wedge 0 \leq c_1 \leq c_2 \\
& \underset{(\text{OD4b})_R}{\Rightarrow} c_1/a_2 \geq c_1/b_2 \geq 0 \wedge c_2/a_1 \geq c_2/b_1 \geq 0 \\
& \underset{(\text{E})}{\Rightarrow} C \boxslash A \geq C \boxslash B \geq [0,0].
\end{aligned}
$$

■

The rules (B)–(H) in Theorem 4.5 cover all cases where both of the operand intervals are comparable with zero with respect to the order relation $\leq$. In all these cases the result of an operation $A \boxcircle B$, $\circ \in \{\cdot, /\}$, can be expressed in terms of the bounds of the operand intervals. The remaining cases are those in which one or both operands are not comparable with $[0,0]$ with respect to the order relation $\leq$. We shall deal with these cases in the next section using the additional hypothesis that the set $\{R, \leq\}$ is linearly ordered.

We already know that the real numbers $\{\mathbb{R}, \{0\}, +, \cdot, /, \leq\}$ are a completely ordered division ringoid if they are completed by the least and greatest elements $-\infty$ and $+\infty$. (These two elements, however, do not obey all the algebraic rules.) By Theorem 4.5, therefore, $\{I\mathbb{R}, N, \boxplus, \boxdot, \boxslash, \leq, \subseteq\}$, $N := \{A \in I\mathbb{R} \mid 0 \in A\}$ is a completely ordered division ringoid with respect to $\leq$, while with respect to inclusion $\subseteq$ it is, moreover, an inclusion-isotonally ordered monotone upper screen ringoid of the power set $\mathbb{PR}$.

Since $\{\mathbb{C}, \{0\}, +, \cdot, /, \leq\}$ is a completely and weakly ordered division ringoid, we obtain the same properties for the interval set $\{I\mathbb{C}, N, \boxplus, \boxdot, \boxslash, \leq, \subseteq\}$, $N := \{A \in I\mathbb{C} \mid 0 \in A\}$ except that the latter is weakly ordered with respect to $\leq$.

We also know that $\{M_n\mathbb{R}, +, \cdot, \leq\}$ is a completely ordered ringoid and that $\{M_n\mathbb{C}, +, \cdot, \leq\}$ is a completely and weakly ordered ringoid. By Theorem 4.5 therefore $\{IM_n\mathbb{R}, \boxplus, \boxdot, \leq, \subseteq\}$ is a completely ordered ringoid and $\{IM_n\mathbb{C}, \boxplus, \boxdot, \leq, \subseteq\}$ is a completely and weakly ordered ringoid with respect to $\leq$. With respect to $\subseteq$, both ringoids are inclusion-isotonally ordered and monotone upper screen ringoids of $\mathbb{P}M_n\mathbb{R}$ (resp. $\mathbb{P}M_n\mathbb{C}$).

We now consider vectoids. The corresponding concepts can be developed quite similarly. Let $\{V, \leq\}$ be a complete lattice, IV the set of intervals over V, and $\overline{IV} := IV \cup \{\emptyset\}$. Then $\{\overline{IV}, \subseteq\}$ is an upper screen of $\{\mathbb{P}V, \subseteq\}$. The monotone upwardly directed rounding is defined by the following properties:

(R1) $\bigwedge\limits_{\boldsymbol{A} \in IV} \square \boldsymbol{A} = \boldsymbol{A}$,

(R2) $\bigwedge\limits_{\boldsymbol{A},\boldsymbol{B} \in \mathbb{P}V} (\boldsymbol{A} \subseteq \boldsymbol{B} \Rightarrow \square \boldsymbol{A} \subseteq \square \boldsymbol{B})$,

(R3) $\bigwedge\limits_{\boldsymbol{A} \in \mathbb{P}V} \boldsymbol{A} \subseteq \square \boldsymbol{A}$.

The following theorem characterizes the set $\overline{IV}$ and the rounding $\square : \mathbb{P}V \to \overline{IV}$ just introduced.

Theorem 4.6. *Let $\{V, R, \leq\}$ be a completely and weakly ordered vectoid with the neutral element $\boldsymbol{o}$. Then*

(a) $\{\overline{IV}, \subseteq\}$ is a symmetric upper screen of $\{\mathbb{P}V, \mathbb{P}R, \subseteq\}$, i.e., we have

(S3) $[\boldsymbol{o}, \boldsymbol{o}]$ *(and* $[\boldsymbol{e}, \boldsymbol{e}]$*)*^{2} $\in \overline{IV} \wedge \bigwedge\limits_{\boldsymbol{A}=[\boldsymbol{a}_1,\boldsymbol{a}_2] \in IV} -\boldsymbol{A} = [-\boldsymbol{a}_2, -\boldsymbol{a}_1] \in IV.$

(b) The monotone upwardly directed rounding $\square : \mathbb{P}V \to \overline{IV}$ is antisymmetric, i.e.,

(R4) $\bigwedge\limits_{\emptyset \neq \boldsymbol{A} \in \mathbb{P}V} \square(-\boldsymbol{A}) = -\square \boldsymbol{A}$.

(c) We have

(R) $\bigwedge\limits_{\emptyset \neq \boldsymbol{A} \in \mathbb{P}V} \square \boldsymbol{A} = \inf(U(\boldsymbol{A}) \cap IV) = o(U(\boldsymbol{A}) \cap IV) = [\inf_{\boldsymbol{a} \in \boldsymbol{A}} \boldsymbol{a}, \sup_{\boldsymbol{a} \in \boldsymbol{A}} \boldsymbol{a}]$.

Proof. By employing corresponding properties of the vectoid, the proof can be given in a manner similar to the proof of Theorem 4.4. ■

Now we employ the monotone upwardly directed rounding $\square : \mathbb{P}V \to \overline{IV}$ to define inner and outer operations in IV by the corresponding semimorphism. This semimorphism has the following properties:

(RG) $\bigwedge\limits_{\boldsymbol{A},\boldsymbol{B} \in IV} \boldsymbol{A} \boxdot \boldsymbol{B} := \square(\boldsymbol{A} \circ \boldsymbol{B}) = [\inf_{a \in \boldsymbol{A}, b \in \boldsymbol{B}}(a \circ b), \sup_{a \in \boldsymbol{A}, b \in \boldsymbol{B}}(a \circ b)]$.

The operations defined by (RG) have the following properties (to which, by Theorems 1.32 and 1.38, (RG) is equivalent):

(RG1) $\bigwedge\limits_{\boldsymbol{A},\boldsymbol{B} \in IV} (\boldsymbol{A} \circ \boldsymbol{B} \in IV \Rightarrow \boldsymbol{A} \boxdot \boldsymbol{B} = \boldsymbol{A} \circ \boldsymbol{B})$,

(RG2) $\bigwedge\limits_{\boldsymbol{A},\boldsymbol{B},\boldsymbol{C},\boldsymbol{D} \in IV} (\boldsymbol{A} \circ \boldsymbol{B} \subseteq \boldsymbol{C} \circ \boldsymbol{D} \Rightarrow \boldsymbol{A} \boxdot \boldsymbol{B} \subseteq \boldsymbol{C} \boxdot \boldsymbol{D})$,

(RG3) $\bigwedge\limits_{\boldsymbol{A},\boldsymbol{B} \in IV} (\boldsymbol{A} \circ \boldsymbol{B} \subseteq \boldsymbol{A} \boxdot \boldsymbol{B})$.

[2] If V is multiplicative.

In the following theorem we characterize interval vectoids and derive explicit representations for certain interval operations.

Theorem 4.7. *(a) Let $\{V, R, \leq\}$ be a completely and weakly ordered vectoid with the neutral elements* $\boldsymbol{o}$ *and* $\boldsymbol{e}$ *(the latter if a multiplication exists), and let $\{IR, \boxplus, \boxdot, \leq, \subseteq\}$ be the monotone upper screen ringoid of $\mathbb{P}R$. If operations in IV are defined by the semimorphism $\square : \mathbb{P}V \to \overline{IV}$, then $\{IV, IR, \leq, \subseteq\}$ is a completely and weakly ordered vectoid with respect to the order relation $\leq$. The neutral elements are $[\boldsymbol{o}, \boldsymbol{o}]$ and $[\boldsymbol{e}, \boldsymbol{e}]$. With respect to inclusion $\subseteq$, IV is an inclusion-isotonally ordered monotone upper screen vectoid of $\{\mathbb{P}V, \mathbb{P}R\}$. It is multiplicative if $\{V, R, \leq\}$ is. Moreover, for all $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2]$, $\boldsymbol{B} = [\boldsymbol{b}_1, \boldsymbol{b}_2] \in IV$, we have*

(A) $\boldsymbol{A} \boxplus \boldsymbol{B} = [\boldsymbol{a}_1 + \boldsymbol{b}_1, \boldsymbol{a}_2 + \boldsymbol{b}_2]$, $\boldsymbol{A} \boxminus \boldsymbol{B} = \boldsymbol{A} \boxplus (-\boldsymbol{B}) = [\boldsymbol{a}_1 - \boldsymbol{b}_2, \boldsymbol{a}_2 - \boldsymbol{b}_1]$.

(b) If in addition $\{V, R, \leq\}$ is ordered, then $\{IV, IR, \leq, \subseteq\}$ is also ordered with respect of $\leq$, and for all $A = [a_1, a_2] \in IR$ we have

(B) $A \geq [0,0] \wedge \boldsymbol{B} \geq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow A \boxdot \boldsymbol{B} = [a_1\boldsymbol{b}_1, a_2\boldsymbol{b}_2]$,

(C) $A \leq [0,0] \wedge \boldsymbol{B} \leq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow A \boxdot \boldsymbol{B} = [a_2\boldsymbol{b}_2, a_1\boldsymbol{b}_1]$,

(D) $A \leq [0,0] \wedge \boldsymbol{B} \geq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow A \boxdot \boldsymbol{B} = [a_1\boldsymbol{b}_2, a_2\boldsymbol{b}_1]$,

(E) $A \geq [0,0] \wedge \boldsymbol{B} \leq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow A \boxdot \boldsymbol{B} = [a_2\boldsymbol{b}_1, a_1\boldsymbol{b}_2]$.

If a multiplication exists in V, we also obtain for all $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2]$:

(F) $\boldsymbol{A} \geq [\boldsymbol{o}, \boldsymbol{o}] \wedge \boldsymbol{B} \geq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow \boldsymbol{A} \boxdot \boldsymbol{B} = [\boldsymbol{a}_1\boldsymbol{b}_1, \boldsymbol{a}_2\boldsymbol{b}_2]$,

(G) $\boldsymbol{A} \leq [\boldsymbol{o}, \boldsymbol{o}] \wedge \boldsymbol{B} \leq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow \boldsymbol{A} \boxdot \boldsymbol{B} = [\boldsymbol{a}_2\boldsymbol{b}_2, \boldsymbol{a}_1\boldsymbol{b}_1]$,

(H) $\boldsymbol{A} \geq [\boldsymbol{o}, \boldsymbol{o}] \wedge \boldsymbol{B} \leq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow \boldsymbol{A} \boxdot \boldsymbol{B} = [\boldsymbol{a}_2\boldsymbol{b}_1, \boldsymbol{a}_1\boldsymbol{b}_2]$,

(I) $\boldsymbol{A} \leq [\boldsymbol{o}, \boldsymbol{o}] \wedge \boldsymbol{B} \geq [\boldsymbol{o}, \boldsymbol{o}] \Rightarrow \boldsymbol{A} \boxdot \boldsymbol{B} = [\boldsymbol{a}_1\boldsymbol{b}_2, \boldsymbol{a}_2\boldsymbol{b}_1]$,

Proof. (a) Theorem 3.11 implies that $\{IV, IR, \leq, \subseteq\}$ is a monotone upper screen vectoid of $\{\mathbb{P}V, \mathbb{P}R\}$, which is multiplicative if $\{V, R, \leq\}$ is. The inclusion-isotony is a simple consequence of (OV5) in $\mathbb{P}V$, of the monotonicity of the rounding $\square : \mathbb{P}V \to \overline{IV}$, and of (RG). The proofs of the properties (OV1,2,3,4) and (A)–(I), which we omit, are analogous to the proofs of the corresponding properties of Theorem 4.5. ■

As before, the rules (A)–(I) in Theorem 4.7 cover all cases where both operand intervals are comparable with zero with respect to $\leq$. In all these cases the results of the operation $A \boxdot \boldsymbol{B}$ or $\boldsymbol{A} \boxdot \boldsymbol{B}$ can be expressed in terms of the bounds of the operand intervals. The remaining cases are those in which one or both operands are not comparable with the zero interval. In Section 4.3 of this chapter we shall describe a method that furnishes explicit formulas for the resulting interval in these cases.

We know that $\{\mathbb{R}, +, \cdot, \leq\}$ is a completely ordered ringoid and that $\{\mathbb{C}, +, \cdot, \leq\}$ is a completely and weakly ordered ringoid. By Theorem 4.7, therefore, $\{IV_n\mathbb{R}, I\mathbb{R}, \leq, \subseteq\}$ and $\{IV_n\mathbb{R}, IM_n\mathbb{R}, \leq, \subseteq\}$ are completely ordered vectoids with respect to $\leq$. $\{IM_n\mathbb{R}, I\mathbb{R}, \leq, \subseteq\}$ is a completely ordered multiplicative vectoid. Further, $\{IV_n\mathbb{C}, I\mathbb{C}, \leq, \subseteq\}$ and $\{IV_n\mathbb{C}, IM_n\mathbb{C}, \leq, \subseteq\}$ are completely and weakly ordered vectoids with respect to $\leq$, while $\{IM_n\mathbb{C}, I\mathbb{C}, \leq, \subseteq\}$ is a completely and weakly ordered multiplicative vectoid. With respect to the inclusion $\subseteq$, all these vectoids are inclusion-isotonally ordered monotone upper screen vectoids of the vectoids in the corresponding power sets.

4.2 Interval Arithmetic Over a Linearly Ordered Set

We begin with a characterization of incomparable intervals.

Lemma 4.8. *In a linearly ordered set $\{R, \leq\}$, two intervals $A = [a_1, a_2]$, $B = [b_1, b_2] \in IR$ are incomparable, $A \parallel B$, with respect to $\leq$ if and only if*

$$a_1 < b_1 \leq b_2 < a_2 \quad \vee \quad b_1 < a_1 \leq a_2 < b_2.$$

Proof. It is clear that intervals with this property are incomparable. If $a_1 = b_1$, then A and B are comparable. Therefore, $A \parallel B \Rightarrow a_1 < b_1 \vee b_1 < a_1$. If in the first case $a_2 \leq b_2$, then A and B are comparable. Therefore $b_2 < a_2$. The second case is dual. ■

Thus far we have defined all interval operations by means of the formula (RG). This definition, however, is not directly usable on computers. In certain cases in Theorem 4.5, we have expressed the result of an operation $A \boxcirc B$ in terms of the bounds of the interval operands. For multiplication and division this was only possible whenever both of the operands were comparable with $[o, o]$ with respect to $\leq$. The remaining cases are those in which one or both operands are incomparable with $[o, o]$.

By Lemma 4.8 we see that in a linearly ordered set an interval A is incomparable with $[o, o]$ with respect to $\leq$ if and only if o is an interior point of A, i.e., $o \in \mathring{A} := \{x \in R \mid a_1 < x < a_2\}$. Therefore, the remaining cases are those in which one or both interval operands have the element $o \in R$ as an interior point. Thus for multiplication $A \boxdot B$, we still have to consider the cases

(a) $A \geq [o, o] \wedge o \in \mathring{B}$,

(b) $A \leq [o, o] \wedge o \in \mathring{B}$,

(c) $o \in \mathring{A} \wedge B \geq [o, o]$,

(d) $o \in \mathring{A} \wedge B \leq [o, o]$,

(e) $o \in \mathring{A} \wedge o \in \mathring{B}$.

Further, for division $A \boxslash B$, with $A \in IR$ and $B \in IR \setminus \widetilde{N}$, the cases

(a) $o \in \overset{\circ}{A} \wedge o < b_1 \leq b_2$,

(b) $o \in \overset{\circ}{A} \wedge b_1 \leq b_2 < o$

need to be considered.

All these cases will be covered by Theorem 4.10 to follow. To prove Theorem 4.10, we require the following lemma:

Lemma 4.9. *Let $\{R, \leq\}$ be a linearly ordered complete lattice, and let $A_i \subseteq R$, $i = 1(1)n$. If $A := \bigcup_{i=1}^{n} A_i$, then*

$$\inf A = \min_{i=1(1)n} \{\inf A_i\} \quad \wedge \quad \sup A = \max_{i=1(1)n} \{\sup A_i\}.$$

Proof. Suppose A consists of two sets $A = A_1 \cup A_2$. Then either $\inf A_1 = \inf A_2$ or by relabeling if necessary $\inf A_1 < \inf A_2$. In the first case $\inf A = \inf A_1 = \inf A_2$, and in the second case $\inf A = \inf A_1 < \inf A_2$. Thus the assertion of the lemma is correct in either case. The balance of the proof is a consequence of the fact that the union of sets is associative: $A = (\cdots (A_1 \cup A_2) \cup \cdots \cup A_{n-1}) \cup A_n$. ∎

Theorem 4.10. *Let $\{R, N, +, \cdot, /, \leq\}$, $(o \in N)$, be a completely and linearly ordered division ringoid with the neutral elements o and e, and let $\{IR, \widetilde{N}, \boxplus, \boxdot, \boxslash, \leq\}$, $\widetilde{N} := \{A \in IR \mid A \cap N \neq \emptyset\}$ be the ordered division ringoid of intervals over R. Then for $A = [a_1, a_2]$, $B = [b_1, b_2] \in IR$ the following properties hold:*

(a) $A \geq [o, o] \wedge o \in \overset{\circ}{B} \Rightarrow A \boxdot B = [a_2 b_1, a_2 b_2]$,

(b) $A \leq [o, o] \wedge o \in \overset{\circ}{B} \Rightarrow A \boxdot B = [a_1 b_2, a_1 b_1]$,

(c) $o \in \overset{\circ}{A} \wedge B \geq [o, o] \Rightarrow A \boxdot B = [a_1 b_2, a_2 b_2]$,

(d) $o \in \overset{\circ}{A} \wedge B \leq [o, o] \Rightarrow A \boxdot B = [a_2 b_1, a_1 b_1]$,

(e) $o \in \overset{\circ}{A} \wedge o \in \overset{\circ}{B} \Rightarrow A \boxdot B = [\min\{a_1 b_2, a_2 b_1\}, \max\{a_1 b_1, a_2 b_2\}]$.

Moreover, for $A = [a_1, a_2] \in IR$ and $B = [b_1, b_2] \in IR \setminus \widetilde{N}$:

(f) $o \in \overset{\circ}{A} \wedge o < b_1 \leq b_2 \Rightarrow A \boxslash B = [a_1/b_1, a_2/b_1]$,

(g) $o \in \overset{\circ}{A} \wedge b_1 \leq b_2 < o \Rightarrow A \boxslash B = [a_2/b_2, a_1/b_2]$.

Proof. The operations are defined by (RG)

$$A \boxed{\circ} B := \square(A \circ B) = [\inf(A \circ B), \sup(A \circ B)].$$

If $o \in B = [b_1, b_2]$ so that $b_1 < o < b_2$ and if $B_1 := [b_1, o]$ and $B_2 := [o, b_2]$ so that $B = B_1 \cup B_2$, then $A \cdot B = \{a \cdot b \mid a \in A \wedge b \in B\} = \{a \cdot b \mid a \in A \wedge b \in B_1\} \cup \{a \cdot b \mid a \in A \wedge b \in B_2\} = A \cdot B_1 \cup A \cdot B_2$.

By Lemma 4.9 we get

$$\begin{aligned}\inf(A \cdot B) &= \min\{\inf(A \cdot B_1), \inf(A \cdot B_2)\},\\ \sup(A \cdot B) &= \max\{\sup(A \cdot B_1), \sup(A \cdot B_2)\}.\end{aligned}$$

With this relation we use Theorem 4.5 to obtain

(a) $A \geq [o, o] \wedge o \in B$

$$\Rightarrow A \boxdot B = [\min\{a_2 b_1, o\}, \max\{o, a_2 b_2\}] = [a_2 b_1, a_2 b_2].$$

(b), (c), (d) are proved in the same manner.

(e) $o \in A \wedge o \in B$

$$\begin{aligned}\Rightarrow A \boxdot B &= [\min\{o, o, a_1 b_2, a_2, b_1\}, \max\{a_1 b_1, a_2, b_2, o, o\}]\\ &= [\min\{a_1 b_2, a_2, b_1\}, \max\{a_1 b_1, a_2, b_2\}].\end{aligned}$$

(f) $o \in A \wedge o < b_1 \leq b_2$

$$\Rightarrow A \boxslash B = [\min\{a_1/b_1, o\}, \max\{o, a_2/b_1\}] = [a_1/b_1, a_2/b_1].$$

(g) is proved in the same manner. ■

As the result of Theorems 4.5 and 4.10, we can state that in a linearly ordered set even for multiplication and division, the result of an interval operation $A \boxcircle B$ can be expressed in terms of the bounds of the interval operands. To get each of these bounds, typically only one multiplication or division is necessary. Only in the case of Theorem 4.10(e), $o \in A$ and $o \in B$, do two products have to be calculated and compared. All these formulas are easily implemented on a computer.

For multiplication and division we summarize these results in Tables 4.1 and 4.2. The formulas therein hold, in particular, for computations with real intervals.

The results of Theorems 4.5 and 4.10 are summarized by the following corollary.

Corollary 4.11. *Let $\{R, \{o\}, +, \cdot, /, \leq\}$ be a completely and linearly ordered division ringoid. For intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$ of IR, the following formula holds:*

$$A \boxcircle B := \square(A \circ B) = [\min_{i,j=1,2}\{a_i \circ b_j\}, \max_{i,j=1,2}\{a_i \circ b_j\}]$$

for operations $\circ \in \{+, -, \cdot, /\}$, $0 \notin B$ in case of division. ■

Whenever in the Tables 4.1 and 4.2 both operands are comparable with the interval $[o, o]$ with respect to $\leq$, $\geq$, $<$, or $>$, the result of the interval operation $A \cdot B$ or A/B contains both bounds of A and B. If one or both of the operands A or B, however, contains zero as an interior point, then the result $A \cdot B$ and A/B is expressed by

	$A = [a_1, a_2]$	$B = [b_1, b_2]$	$A \cdot B$
1	$A \geq [o, o]$	$B \geq [o, o]$	$[a_1b_1, a_2b_2]$
2	$A \geq [o, o]$	$B \leq [o, o]$	$[a_2b_1, a_1b_2]$
3	$A \geq [o, o]$	$b_1 < o < b_2$	$[a_2b_1, a_2b_2]$
4	$A \leq [o, o]$	$B \geq [o, o]$	$[a_1b_2, a_2b_1]$
5	$A \leq [o, o]$	$B \leq [o, o]$	$[a_2b_2, a_1b_1]$
6	$A \leq [o, o]$	$b_1 < o < b_2$	$[a_1b_2, a_1b_1]$
7	$a_1 < o < a_2$	$B \geq [o, o]$	$[a_1b_2, a_2b_2]$
8	$a_1 < o < a_2$	$B \leq [o, o]$	$[a_2b_1, a_1b_1]$
9	$a_1 < o < a_2$	$b_1 < o < b_2$	$[\min\{a_1b_2, a_2b_1\}, \max\{a_1b_1, a_2b_2\}]$

Table 4.1. Execution of multiplication.

	$A = [a_1, a_2]$	$B = [b_1, b_2]$	A/B
1	$A \geq [o, o]$	$0 < b_1 \leq b_2$	$[a_1/b_2, a_2/b_1]$
2	$A \geq [o, o]$	$b_1 \leq b_2 < 0$	$[a_2/b_2, a_1/b_1]$
3	$A \leq [o, o]$	$0 < b_1 \leq b_2$	$[a_1/b_1, a_2/b_2]$
4	$A \leq [o, o]$	$b_1 \leq b_2 < 0$	$[a_2/b_1, a_1/b_2]$
5	$a_1 < o < a_2$	$0 < b_1 \leq b_2$	$[a_1/b_1, a_2/b_1]$
6	$a_1 < o < a_2$	$b_1 \leq b_2 < 0$	$[a_2/b_2, a_1/b_2]$

Table 4.2. Execution of division with B not containing 0.

only three of the four bounds of A and B. In all these cases (3,6,7,8,9) in Table 4.1, the bound which is missing in the expression for the result can be shifted towards zero without changing the result of the operation $A \cdot B$. Similarly, in cases 5 and 6 in Table 4.2, the bound of B, which is missing in the expression for the resulting interval, can be shifted toward ∞ (resp. $-\infty$) without changing the result of the operation A/B. This shows a certain lack of sensitivity of interval arithmetic whenever in multiplication and division one of the operands contains zero as an interior point.

In all these cases – 3,6,7,8,9 of Table 4.1 and 5,6 of Table 4.2 – the result of $A \cdot B$ or A$/B$ also contains zero, and the formulas show that the result tends toward the zero interval if the operands that contain zero do likewise. In the limit when the operand that contains zero has become the zero interval, no such imprecision is left. This suggests that within arithmetic expressions interval operands that contain zero as an interior point should be made as small in diameter as possible.

We conclude this section with the following observation concerning $I\mathbb{R}$.

Remark 4.12. It is an interesting fact, but not essential for the development of this theory, that the intervals $I\mathbb{R}$ over the real numbers $\mathbb{R}$ form an algebraically closed subset of the power set $\mathbb{PR}$, i.e.,

$$\bigwedge_{A,B \in I\mathbb{R}} A \boxed{\circ} B := \square(A \circ B) = A \circ B, \text{ for all } \circ \in \{+, -, \cdot, /\}.$$

Here for division we assume that $0 \notin B$.

To see this, we recall that $A \boxcirc B := \square(A \circ B)$ is the least interval that contains $A \circ B := \{a \circ b \mid a \in A, b \in B\}$. It is well known from real analysis that for all $\circ \in \{+, -, \cdot, /\}$, $a \circ b$ is a continuous function of both variables. $A \circ B$ is the range of this function over the product set $A \times B$. Since A and B are closed intervals, $A \times B$ is a simply connected, bounded and closed subset of $\mathbb{R}^2$. In such a region the continuous function $a \circ b$ takes a maximum and a minimum as well as all values in between. Therefore,

$$A \circ B := \left[\min_{a \in A, b \in B} \{a \circ b\}, \max_{a \in A, b \in B} \{a \circ b\} \right] = \square(A \circ B) = A \boxcirc B. \qquad \blacksquare$$

4.3 Interval Matrices

Let $\{R, +, \cdot, \leq\}$ be a completely ordered (resp. weakly ordered) ringoid with the neutral elements o and e, and $\{M_n R, +, \cdot, \leq\}$ the ordered (resp. weakly ordered) ringoid of matrices over R with the neutral elements

$$\boldsymbol{O} = \begin{pmatrix} o & o & \dots & o \\ o & o & \ddots & \vdots \\ \vdots & \ddots & \ddots & o \\ o & \dots & o & o \end{pmatrix}, \quad \boldsymbol{E} = \begin{pmatrix} e & o & \dots & o \\ o & e & \ddots & \vdots \\ \vdots & \ddots & \ddots & o \\ o & \dots & o & e \end{pmatrix}.$$

The power set $\{\mathbb{P}M_n R \setminus \{\varnothing\}, +, \cdot, \subseteq\}$ is also a ringoid. If $IM_n R$ denotes the set of intervals over $\{M_n R, \leq\}$, then according to Theorem 4.4, $\overline{IM_n R} := IM_n R \cup \varnothing$ is a symmetric upper screen of $\{\mathbb{P}M_n R, \subseteq\}$, and the monotone upwardly directed rounding $\square : \mathbb{P}M_n R \to \overline{IM_n R}$ is antisymmetric. We consider its semimorphism, which in $IM_n R$ defines operations $\boxcirc$, with $\circ \in \{+, \cdot\}$, by

(RG) $\bigwedge_{\boldsymbol{A}, \boldsymbol{B} \in IM_n R} \boldsymbol{A} \boxcirc \boldsymbol{B} := \square(\boldsymbol{A} \circ \boldsymbol{B}) = [\inf(\boldsymbol{A} \circ \boldsymbol{B}), \sup(\boldsymbol{A} \circ \boldsymbol{B})]$.

Then according to Theorem 4.5 and under its hypotheses, $\{IM_n R, \boxplus, \boxdot, \leq, \subseteq\}$ is also a completely ordered (resp. weakly ordered) ringoid with respect to $\leq$ and an inclusion-isotonally ordered monotone upper screen ringoid of $\mathbb{P}M_n R$ with respect to $\subseteq$.

By Theorem 4.5(a), the result of an addition or subtraction in $IM_n R$ can always be expressed in terms of the bounds of the operands. For a multiplication this is only possible if $\{R, +, \cdot, \leq\}$ is an ordered ringoid and the operands are comparable with the interval $[\boldsymbol{O}, \boldsymbol{O}] \in IM_n R$ with respect to $\leq$. We are now going to derive explicit formulas that are simple to implement for all products in $IM_n R$.

To do this, we consider the set of $n \times n$ matrices $M_n IR$. The elements of this set have components that are intervals over R. If $\{R, +, \cdot, \leq\}$ is a completely ordered (resp. weakly ordered) ringoid, then by Theorem 4.5, $\{IR, \boxplus, \boxdot, \leq\}$ is such a structure also. With the operations and order relation of the latter, we define operations $\circledcirc$,

$\circ \in \{+, \cdot\}$, and an order relation $\leq$ in $M_n IR$ by employing the conventional definition of operations for matrices:

$$\begin{aligned}
\bigwedge_{\boldsymbol{A}=(A_{ij}),\boldsymbol{B}=(B_{ij})\in M_n IR} \quad & \boldsymbol{A} \oplus \boldsymbol{B} := (A_{ij} \boxplus B_{ij}), \\
\bigwedge_{\boldsymbol{A}=(A_{ij}),\boldsymbol{B}=(B_{ij})\in M_n IR} \quad & \boldsymbol{A} \odot \boldsymbol{B} := \left(\boxed{\sum}_{\nu=1}^{n} A_{i\nu} \boxdot B_{\nu j} \right), \\
& (A_{ij}) \leq (B_{ij}) :\Leftrightarrow \bigwedge_{i=1(1)n} \bigwedge_{j=1(1)n} A_{ij} \leq B_{ij}.
\end{aligned} \tag{4.3.1}$$

Here $\boxed{\sum}$ denotes the repeated summation in IR.

By Theorem 2.6, we deduce directly that $\{M_n IR, \oplus, \odot, \leq\}$ is a completely ordered (resp. weakly ordered) ringoid. We shall see that under certain further assumptions the ringoids $\{M_n IR, \oplus, \odot, \leq\}$ and $\{IM_n R, \boxplus, \boxdot, \leq\}$ are isomorphic.

To this end, we define a mapping

$$\chi : M_n IR \to IM_n R$$

which for matrices $\boldsymbol{A} = (A_{ij}) \in M_n IR$ with $A_{ij} = [a_{ij}^{(1)}, a_{ij}^{(2)}] \in IR$, $i, j = 1(1)n$, has the property

$$\chi \boldsymbol{A} = \chi(A_{ij}) = \chi([a_{ij}^{(1)}, a_{ij}^{(2)}]) := [(a_{ij}^{(1)}), (a_{ij}^{(2)})]. \tag{4.3.2}$$

Obviously χ is a one-to-one mapping of $M_n IR$ onto $IM_n R$ and an order isomorphism with respect to $\leq$. To characterize the algebraic operations, we prove Lemmas 4.13 and 4.14. In certain cases we find that χ is also an algebraic isomorphism.

Lemma 4.13. *Let $\{R, +, \cdot, \leq\}$ be a completely and weakly ordered ringoid and consider the semimorphism defined by the monotone upwardly directed rounding $\square : \mathbb{P}R \to \overline{IR}$. If the formula*

$$\bigwedge_{A_\nu, B_\nu \in IR} \left(\sum_{\nu=1}^{n} A_\nu \boxdot B_\nu \subseteq \square \sum_{\nu=1}^{n} A_\nu \cdot B_\nu \right) \tag{4.3.3}$$

between the operations in $\{\mathbb{P}R, +, \cdot\}$ and $\{IR, \boxplus, \boxdot, \leq\}$ holds, then the mapping χ establishes an isomorphism between the completely and weakly ordered ringoids $\{M_n IR, \oplus, \odot, \leq\}$ and $\{IM_n R, \boxplus, \boxdot, \leq\}$ with respect to the algebraic operations and the order relation.

Proof. The isomorphism of the order structures was already noted. Let $\boldsymbol{A}, \boldsymbol{B} \in M_n IR$, where

$$\boldsymbol{A} := ([a_{ij}^{(1)}, a_{ij}^{(2)}]), \quad \boldsymbol{B} := ([b_{ij}^{(1)}, b_{ij}^{(2)}]).$$

Then for the addition of the images,

$$\chi\boldsymbol{A} = [(a_{ij}^{(1)}), (a_{ij}^{(2)})],\ \chi\boldsymbol{B} = [(b_{ij}^{(1)}), (b_{ij}^{(2)})] \in IM_nR$$

we obtain by Theorem 4.5(a) that

$$\begin{aligned}\chi\boldsymbol{A} \boxplus \chi\boldsymbol{B} = \square\,(\chi\boldsymbol{A} + \chi\boldsymbol{B}) &= [(a_{ij}^{(1)}) + (b_{ij}^{(1)}), (a_{ij}^{(2)}) + (b_{ij}^{(2)})] \\ &= [(a_{ij}^{(1)} + b_{ij}^{(1)}), (a_{ij}^{(2)} + b_{ij}^{(2)})].\end{aligned} \tag{4.3.4}$$

In M_nIR we obtain also by Theorem 4.5(a) that

$$\boldsymbol{A} \oplus \boldsymbol{B} := (A_{ij} \boxplus B_{ij}) = ([a_{ij}^{(1)} + b_{ij}^{(1)}, a_{ij}^{(2)} + b_{ij}^{(2)}]). \tag{4.3.5}$$

Equations (4.3.4) and (4.3.5) imply

$$\chi\boldsymbol{A} \boxplus \chi\boldsymbol{B} = \chi(\boldsymbol{A} \oplus \boldsymbol{B}),$$

which proves the isomorphism of addition. For subtraction the proof can be given analogously.

As a next step we prove the isomorphism for multiplication:

$$\chi\boldsymbol{A} \boxdot \chi\boldsymbol{B} = \chi(\boldsymbol{A} \odot \boldsymbol{B}).$$

Let be $A_i \in IR$, $i = 1(1)n$ and $A_i = [a_{i1}, a_{i2}] = [\inf A_i, \sup A_i]$. Then we have

$$\bigwedge_{a_i \in A_i} \left(\inf A_i \le a_i \underset{(\text{OD1})_R}{\Rightarrow} \sum_{i=1}^{n} \inf A_i \le \sum_{i=1}^{n} a_i \right) \Rightarrow \sum_{i=1}^{n} \inf A_i \le \inf \sum_{i=1}^{n} A_i. \tag{4.3.6}$$

On the other hand we have

$$\inf \sum_{i=1}^{n} A_i \le \sum_{i=1}^{n} a_{i1} = \sum_{i=1}^{n} \inf A_i. \tag{4.3.7}$$

From (4.3.6), (4.3.7) and (O3) follows

$$\begin{aligned} &\sum_{i=1}^{n} \inf A_i = \inf \sum_{i=1}^{n} A_i \wedge \sum_{i=1}^{n} \sup A_i = \sup \sum_{i=1}^{n} A_i \\ &\Rightarrow \boxed{\sum_{i=1}^{n}} A_i = \left[\sum_{i=1}^{n} \inf A_i,\ \sum_{i=1}^{n} \sup A_i \right] \\ &\qquad = \left[\inf \sum_{i=1}^{n} A_i,\ \sup \sum_{i=1}^{n} A_i \right] = \square \sum_{i=1}^{n} A_i. \end{aligned} \tag{4.3.8}$$

We have $A_i \cdot B_i \underset{\text{(R3)}}{\subseteq} \square(A_i \cdot B_i) \underset{\text{(RG)}}{=} A_i \boxdot B_i \underset{(\text{OD5})_R}{\Rightarrow} \sum_{i=1}^n A_i \cdot B_i \subseteq \sum_{i=1}^n A_i \boxdot B_i$. Using (R2) we get

$$\square \sum_{i=1}^{n} A_i \cdot B_i \subseteq \square \sum_{i=1}^{n} A_i \boxdot B_i, \tag{4.3.9}$$

and from (4.3.3) we get by applying (R1) and (R2)

$$\square \sum_{i=1}^{n} A_i \boxdot B_i \subseteq \square \sum_{i=1}^{n} A_i \cdot B_i. \tag{4.3.10}$$

Combining (4.3.9) and (4.3.10) yields

$$\square \sum_{i=1}^{n} A_i \boxdot B_i = \square \sum_{i=1}^{n} A_i \cdot B_i. \tag{4.3.11}$$

With $\boldsymbol{A} = (A_{ij})$ and $\boldsymbol{B} = (B_{ij}) \in M_n IR$ we obtain:

$$\begin{aligned}
\chi(\boldsymbol{A} \odot \boldsymbol{B}) := \chi\left(\boxed{\sum_{\nu=1}^{n}} A_{i\nu} \boxdot B_{\nu j}\right) &\underset{(4.3.8)}{=} \chi\left(\square \sum_{\nu=1}^{n} A_{i\nu} \boxdot B_{\nu j}\right) \\
\underset{(4.3.11)}{=} \chi\left(\square \sum_{\nu=1}^{n} A_{i\nu} \cdot B_{\nu j}\right) &= \chi\left(\left[\inf \sum_{\nu=1}^{n} A_{i\nu} B_{\nu j}, \sup \sum_{\nu=1}^{n} A_{i\nu} B_{\nu j}\right]\right) \\
&= \left[\left(\inf \sum_{\nu=1}^{n} A_{i\nu} B_{\nu j}\right), \left(\sup \sum_{\nu=1}^{n} A_{i\nu} B_{\nu j}\right)\right] \\
&= \left[\inf\left(\sum_{\nu=1}^{n} A_{i\nu} B_{\nu j}\right), \sup\left(\sum_{\nu=1}^{n} A_{i\nu} B_{\nu j}\right)\right] \\
&= \left[\inf\left(\sum_{\nu=1}^{n} a_{i\nu} b_{\nu j}\right), \sup\left(\sum_{\nu=1}^{n} a_{i\nu} b_{\nu j}\right)\right] \text{ with } (a_{ij}) \in \chi\boldsymbol{A}, (b_{ij}) \in \chi\boldsymbol{B} \\
&= \square(\chi\boldsymbol{A} \cdot \chi\boldsymbol{B}) = \chi\boldsymbol{A} \boxdot \chi\boldsymbol{B}. \qquad \blacksquare
\end{aligned}$$

Lemma 4.14. *Let* $\{R, +, \cdot, \leq\}$ *be a completely and linearly ordered ringoid and* $\square : \mathbb{P}R \to \overline{IR}$ *the monotone upwardly directed rounding. Then*

$$\bigwedge_{A_\nu, B_\nu \in IR} \left(\sum_{\nu=1}^{n} A_\nu \boxdot B_\nu \subseteq \square \sum_{\nu=1}^{n} A_\nu \cdot B_\nu\right).$$

Proof. By Corollary 4.11, we have for all $A = [a_1, a_2]$ and $B = [b_1, b_2] \in IR$

$$A \boxdot B = [\min_{i,j=1,2}(a_i \cdot b_j), \max_{i,j=1,2}(a_i \cdot b_j)]. \tag{4.3.12}$$

Let $A_i = [a_{i1}, a_{i2}]$ and $B_i = [b_{i1}, b_{i2}]$, $i = 1(1)n$. Then by the definition of addition in $\mathbb{P}R$ we know that every $x \in \sum_{i=1}^{n} A_i \boxdot B_i$ is of the form $x = \sum_{i=1}^{n} x_i$ with $x_i \in A_i \boxdot B_i$. From (4.3.12) we get

$$\min_{\mu,\nu=1,2}(a_{i\mu} \cdot b_{i\nu}) \leq x_i \leq \max_{\mu,\nu=1,2}(a_{i\mu} \cdot b_{i\nu})$$

$$\underset{\mathrm{OD1})_R}{\Rightarrow} \quad \sum_{i=1}^{n} \min_{\mu,\nu=1,2}(a_{i\mu} \cdot b_{i\nu}) \leq x \leq \sum_{i=1}^{n} \max_{\mu,\nu=1,2}(a_{i\mu} \cdot b_{i\nu}).$$

$$\inf \sum_{i=1}^{n} A_i B_i \leq \sum_{i=1}^{n} \min_{\mu,\nu=1,2}(a_{i\mu} \cdot b_{i\nu}) \leq x \leq \sum_{i=1}^{n} \max_{\mu,\nu=1,2}(a_{i\mu} \cdot b_{i\nu}) \leq \sup \sum_{i=1}^{n} A_i B_i,$$

i.e., every $x \in \sum_{i=1}^{n} A_i \boxdot B_i$ is also an element of $\square \sum_{i=1}^{n} A_i \cdot B_i$. ■

Lemmas 4.13 and 4.14 imply the following theorem, which establishes the isomorphic character of χ for a linearly ordered ringoid.

Theorem 4.15. *Let $\{R, +, \cdot, \leq\}$ be a completely and linearly ordered ringoid. Then the mapping χ establishes an isomorphism between the completely ordered ringoids $\{M_n IR, \oplus, \odot, \leq\}$ and $\{IM_n R, \boxplus, \boxdot, \leq\}$ with respect to the algebraic and the order structure.* ■

$$\begin{tikzcd}
R \arrow[rr] \arrow[d] & & M_n R \arrow[d] \\
\mathbb{P}R \arrow[d] & & \mathbb{P}M_n R \arrow[d] \\
IR \arrow[r] & M_n IR \arrow[r, leftrightarrow, "\chi"'] & IM_n R
\end{tikzcd}$$

Figure 4.1. Illustration of Theorem 4.15.

Whenever two structures $\{M_n IR, \oplus, \odot, \leq\}$ and $\{IM_n R, \boxplus, \boxdot, \leq\}$ are isomorphic, corresponding elements can be identified with each other. This allows us to define an inclusion relation even for elements $\boldsymbol{A} = (A_{ij}), \boldsymbol{B} = (B_{ij}) \in M_n IR$ by

$$\boldsymbol{A} \subseteq \boldsymbol{B} :\Leftrightarrow \bigwedge_{i,j=1(1)n} A_{ij} \subseteq B_{ij},$$

and

$$(a_{ij}) \in \boldsymbol{A} = (A_{ij}) :\Leftrightarrow \bigwedge_{i,j=1(1)n} a_{ij} \in A_{ij}.$$

This convenient definition allows for the interpretation that a matrix $\boldsymbol{A} = (A_{ij}) \in M_n IR$ also represents a set of matrices as demonstrated by the following identity:

$$\boldsymbol{A} = (A_{ij}) \equiv \{(a_{ij}) \mid a_{ij} \in A_{ij},\ i,j = 1(1)n\}.$$

Both matrices contain the same elements.

For a linearly ordered ringoid $\{R, +, \cdot, \leq\}$ for all operations in the ringoid $\{IR, \boxplus, \boxdot, \leq\}$, we have derived explicit and easily performable formulas in Section 4.2. The formulas (4.3.1) for the operations $\circledcirc$, $\circ \in \{+, \cdot\}$, in $\{M_n IR, \oplus, \odot, \leq\}$, therefore, are also easily performable. This is not so for the operations $\boxdot$ originally defined in $IM_n R$. By means of the isomorphism, the operation $\odot$ can now be used to perform the multiplication $\boxdot$.

The isomorphism shows that both the conventional definition of the operations in $M_n IR$ in terms of those in IR and the definition of the operations in $IM_n R$ by means of the semimorphism $\square : \mathbb{P}M_n R \to \overline{IM_n R}$ lead to the same result. Figure 4.1 illustrates the connection.

4.4 Interval Vectors

Similar to Section 4.3, we are now going to derive easily implementable formulas for the interval operations occurring in vectoids.

Again let $\{R, +, \cdot, \leq\}$ be a completely ordered or weakly ordered ringoid. Then $\{V_n R, R, \leq\}$, $\{M_n R, R, \leq\}$, and $\{V_n R, M_n R, \leq\}$, where the operations and the order relation are defined by the usual formulas, are completely ordered (resp. weakly ordered) vectoids. $\{M_n R, R, \leq\}$ is in particular multiplicative.

Moreover, by Theorem 2.11 the three power sets $\{\mathbb{P}V_n R, \mathbb{P}R, \subseteq\}$, $\{\mathbb{P}M_n R, \mathbb{P}R, \subseteq\}$, and $\{\mathbb{P}V_n R, \mathbb{P}M_n R, \subseteq\}$ are inclusion-isotonally ordered vectoids (with the empty set excluded from the algebraic operations).

Let us now consider the semimorphism $\square : \mathbb{P}V_n R \to \overline{IV_n R} := IV_n R \cup \emptyset$. With this mapping and by Theorem 4.7, $\{IV_n R, IR, \leq, \subseteq\}$, $\{IM_n R, IR, \leq, \subseteq\}$, and $\{IV_n R, IM_n R, \leq, \subseteq\}$ become completely ordered (resp. weakly ordered) vectoids with respect to $\leq$ and inclusion-isotonally ordered monotone upper screen vectoids of the corresponding power sets with respect to $\subseteq$. Moreover, $IM_n R$ is multiplicative.

By Theorem 4.7, the result of an addition or subtraction in $IV_n R$ and $IM_n R$ can always be expressed in terms of the bounds of the interval operands. For an outer multiplication, this is possible only if the vectoid is ordered and if the operands are comparable with the corresponding zero element with respect to $\leq$. Inner multiplication in $IM_n R$ has already been considered in Section 4.3. We are now going to derive explicit and easily implementable formulas for all outer multiplications in the vectoids $\{IV_n R, IR\}$, $\{IM_n R, IR\}$, and $\{IV_n R, IM_n R\}$.

To do this, we consider the sets $V_n IR$ and $M_n IR$. The elements of these sets have components that are intervals over R. If $\{R, +, \cdot, \leq\}$ is a completely ordered (resp.

weakly ordered) ringoid, then by Theorem 4.5, $\{IR, \boxplus, \boxdot, \leq\}$ also has these properties. Employing the operations and order relation of the latter, we define operations and an order relation $\leq$ in V_nIR and M_nIR by means of the conventional method:

$$\begin{aligned}
&\bigwedge_{\boldsymbol{a}=(A_i),\boldsymbol{b}=(B_i)\in V_nIR} \boldsymbol{a}\oplus\boldsymbol{b} := (A_i \boxplus B_i),\\
&\bigwedge_{A\in IR}\ \bigwedge_{\boldsymbol{a}=(A_i)\in V_nIR} A\odot\boldsymbol{a} := (A \boxdot A_i),\\
&\bigwedge_{A\in IR}\ \bigwedge_{\boldsymbol{A}=(A_{ij})\in M_nIR} A\odot\boldsymbol{A} := (A \boxdot A_{ij}),\\
&\bigwedge_{\boldsymbol{A}=(A_{ij})\in M_nIR}\ \bigwedge_{\boldsymbol{a}=(A_i)\in V_nIR} \boldsymbol{A}\odot\boldsymbol{a} := \left(\boxed{\sum_{\nu=1}^{n}} A_{i\nu} \boxdot A_\nu\right),\\
&(A_i) \leq (B_i) := \bigwedge_{i=1(1)n} A_i \leq B_i.
\end{aligned} \tag{4.4.1}$$

Here as before $\boxed{\sum}$ denotes the repeated summation in IR.

For M_nIR, we take the definition of the inner operations $\oplus$, $\odot$, and the order relation $\leq$ as given in (4.3.1) in Section 4.3.

Using Theorems 2.13, 2.14 and 2.15, we obtain directly that $\{V_nIR, IR, \leq\}$, $\{M_nIR, IR, \leq\}$, and $\{V_nIR, M_nIR, \leq\}$ are completely ordered (resp. weakly ordered) vectoids. Moreover, $\{M_nIR, IR, \leq\}$ is multiplicative.

Under certain further assumptions, we shall see that isomorphisms exist between the following pairs of vectoids:

$$\begin{aligned}
\{V_nIR, IR, \leq\} &\leftrightarrow \{IV_nR, IR, \leq\},\\
\{M_nIR, IR, \leq\} &\leftrightarrow \{IM_nR, IR, \leq\},\\
\{V_nIR, M_nIR, \leq\} &\leftrightarrow \{IV_nR, IM_nR, \leq\}.
\end{aligned}$$

To see this for vectors and matrices

$$\begin{aligned}
\boldsymbol{a} = (A_i) \in V_nIR \quad &\text{with} \quad A_i = [a_i^{(1)}, a_i^{(2)}] \in IR,\\
\boldsymbol{A} = (A_{ij}) \in M_nIR \quad &\text{with} \quad A_{ij} = [a_{ij}^{(1)}, a_{ij}^{(2)}] \in IR,
\end{aligned}$$

we define the mappings ψ and χ:

$$\psi : V_nIR \to IV_nR, \tag{4.4.2}$$

where $\psi\boldsymbol{a} = \psi(A_i) = \psi([a_i^{(1)}, a_i^{(2)}]) = [(a_i^{(1)}), (a_i^{(2)})]$, and

$$\chi : M_nIR \to IM_nR, \tag{4.4.3}$$

where $\chi\boldsymbol{A} = \chi(A_{ij}) = \chi([a_{ij}^{(1)}, a_{ij}^{(2)}]) = [(a_{ij}^{(1)}), (a_{ij}^{(2)})]$.

Obviously, the mappings ψ and χ are one-to-one and order isomorphisms with respect to $\leq$. The following theorem characterizes properties of ψ and χ with respect to the algebraic operations.

Theorem 4.16. *Let $\{R, +, \cdot, \leq\}$ be a completely and weakly ordered ringoid and $\{IR, \boxplus, \boxdot, \leq\}$ be the completely and weakly ordered ringoid of intervals over R. Then we have*

(a) The mapping ψ establishes an isomorphism between $\{V_nIR, IR, \leq\}$ and $\{IV_nR, IR, \leq\}$ with respect to the algebraic and the order structure.

(b) If $\square : \mathbb{P}R \to \overline{IR}$ denotes the monotone upwardly directed rounding and if for the operations in $\mathbb{P}R$ and IR the formula

$$\bigwedge_{A_\nu, B_\nu \in IR} \left(\sum_{\nu=1}^{n} A_\nu \boxdot B_\nu \subseteq \square \sum_{\nu=1}^{n} A_\nu \cdot B_\nu \right) \tag{4.4.4}$$

holds, then the mapping χ establishes an isomorphism between the two completely and weakly ordered multiplicative vectoids $\{M_nIR, IR, \leq\}$ and $\{IM_nR, IR, \leq\}$ with respect to the algebraic and the order structure.

(c) Under the assumption (4.4.4)*, the mappings establish an isomorphism between the completely and weakly ordered vectoids $\{V_nIR, M_nIR, \leq\}$ and $\{IV_nR, IM_nR, \leq\}$ with respect to the algebraic and the order structure.*

Proof. The isomorphism of the order structure has already been noted immediately preceding the statement of this theorem.

(a) The isomorphism of addition can be proved as in Lemma 4.13. For outer multiplication, we get with $A \in IR$ and $\boldsymbol{a} = (A_i) \in V_nIR$, $A_i \in IR$, $i = 1(1)n$:

$$\begin{aligned} \psi(A \odot \boldsymbol{a}) &:= \psi(A \boxdot A_i) = \psi(\square(A \cdot A_i)) \\ &= \psi(\inf(A \cdot A_i), \sup(A \cdot A_i)) = [(\inf(A \cdot A_i)), (\sup(A \cdot A_i))] \\ &= [\inf(A \cdot A_i), \sup(A \cdot A_i)] = \square(A \cdot \psi\boldsymbol{a}) = A \boxdot \psi\boldsymbol{a}. \end{aligned}$$

(b) The isomorphism of the inner operations is proved in Lemma 4.13 above. The proof for outer multiplications is analogous to (a).

(c) Lemma 4.13 demonstrated the isomorphism of the ringoids $\{M_nIR, \oplus, \odot, \leq\}$ and $\{IM_nR, \boxplus, \boxdot, \leq\}$. The isomorphism of addition in V_nIR and IV_nR was proved under (a). Therefore only the isomorphism of outer multiplication remains to be shown. Using (4.4.4) it can be shown as in the proof of Lemma 4.13 that

$$\psi(\boldsymbol{A} \odot \boldsymbol{a}) = \chi\boldsymbol{A} \boxdot \psi\boldsymbol{a}.$$

■

Combining this theorem with Lemma 4.14, we obtain the following theorem which further characterizes the isomorphisms induced by the mappings ψ and χ.

Theorem 4.17. *Let $\{R, +, \cdot, \leq\}$ be a completely and linearly ordered ringoid. Then:*

(a) The mapping χ establishes an isomorphism between the completely ordered vectoids $\{M_n IR, IR, \leq\}$ and $\{IM_n R, IR, \leq\}$ with respect to the algebraic and the order structure.

(b) The mappings ψ and χ establish an isomorphism between the two completely ordered vectoids $\{V_n IR, M_n IR, \leq\}$ and $\{IV_n R, IM_n R, \leq\}$ with respect to the algebraic and the order structure. ∎

Whenever the structures $\{V_n IR, IR, \leq\}$ and $\{IV_n R, IR, \leq\}$ (resp. $\{V_n IR, M_n IR, \leq\}$ and $\{IV_n R, IM_n R, \leq\}$) are isomorphic, corresponding elements can be identified with one another. This allows us to define an inclusion relation even for elements $\boldsymbol{a} = (A_i), \boldsymbol{b} = (B_i) \in V_n IR$ by

$$\boldsymbol{a} \subseteq \boldsymbol{b} :\Leftrightarrow \bigwedge_{i=1(1)n} A_i \subseteq B_i.$$

$$(a_i) \in \boldsymbol{a} = (A_i) :\Leftrightarrow \bigwedge_{i=1(1)n} a_i \in A_i.$$

By means of the following identity, this definition allows for the interpretation that an interval vector $\boldsymbol{a} = (A_i) \in V_n IR$ also represents a set of vectors

$$\boldsymbol{a} = (A_i) \equiv \{(a_i) \mid a_i \in A_i, i = 1(1)n\}.$$

In the case of a linearly ordered ringoid $\{R, +, \cdot, \leq\}$ for all operations in the ringoid $\{IR, \boxplus, \boxdot, \leq\}$, we have derived explicit and easily performable formulas in Section 4.2 of this chapter. Formulas (4.4.1) for the operations $\circledcirc$, therefore, are also easily performable. This is not the case for the operations $\boxed{\circ}$ originally defined, for instance, in $\{IV_n R, IM_n R, \leq\}$. Appealing to the isomorphism, we see that the operations $\circledcirc$ can be used in order to execute the operations $\boxed{\circ}$.

4.5 Interval Arithmetic on a Screen

Let $\{R, \leq\}$ be a complete lattice and $\{S, \leq\}$ a screen of $\{R, \leq\}$. Now we consider intervals over R with endpoints in S:

$$[a_1, a_2] := \{x \in R \mid a_1, a_2 \in S, a_1 \leq x \leq a_2\} \quad \text{with } a_1 \leq a_2.$$

We denote the set of all such intervals by IS. Then $IS \subseteq IR$ and $\overline{IS} := IS \cup \{\varnothing\} \subseteq \overline{IR} := IR \cup \{\varnothing\}$.

Several properties of IS are described in the following theorem.

Theorem 4.18. *Let $\{R, \leq\}$ be a complete lattice and $\{S, \leq\}$ a screen of $\{R, \leq\}$. Then:*

(a) $\{IS, \subseteq\}$ is a conditionally completely ordered set. For all nonempty subsets A of IS, which are bounded below, we have with $B = [b_1, b_2] \in IS$:

$$\inf_{IS} A = [\sup_{B \in A} b_1, \inf_{B \in A} b_2] \quad \wedge \quad \sup_{IS} A = [\inf_{B \in A} b_1, \sup_{B \in A} b_2],$$

i.e., the infimum is the intersection, and the supremum is the interval hull of the elements of A.

(b) $\{\overline{IS}, \subseteq\}$ is a screen of $\{\overline{IR}, \subseteq\}$.

Proof. (a) Since $\{S, \leq\}$ is a complete lattice and A is bounded below (i.e., A contains an interval), the intervals

$$I = [\sup_{B \in A} b_1, \inf_{B \in A} b_2] \quad \wedge \quad S = [\inf_{B \in A} b_1, \sup_{B \in A} b_2]$$

exist, and for all $B \subseteq A$, we have $I \subseteq B \wedge B \subseteq S$, i.e., I is a lower bound and S an upper bound of A. If I_1 is any lower and S_1 any upper bound of A, then $I_1 \subseteq I$ and $S \subseteq S_1$, i.e., I is the greatest lower and S the least upper bound of A.

(b) $\{\overline{IS}, \subseteq\}$ is a complete lattice and therefore a complete subnet of $\{\overline{IR}, \subseteq\}$. We still have to show that for every subset $A \subseteq \overline{IS}$, $\inf_{\overline{IS}} A = \inf_{\overline{IR}} A$ and $\sup_{\overline{IS}} A = \sup_{\overline{IR}} A$.

If A is bounded from below in $\overline{IS}$, then by (a), the infimum is the intersection and the supremum is the interval hull as in $\overline{IR}$. If A is not bounded below in $\overline{IS}$ we consider the cases

1. $A = \emptyset$. Then $\inf_{\overline{IS}} A = \inf_{\overline{IR}} A = i(\overline{IS}) = i(\overline{IR}) = R$ and $\sup_{\overline{IS}} A = \sup_{\overline{IR}} A = o(\overline{IS}) = o(\overline{IR}) = \emptyset$.

2. $A \neq \emptyset$. Then $\sup_{\overline{IS}} A = \sup_{\overline{IR}} A = [\inf_{B \in A} b_1, \sup_{B \in A} b_2]$ by (a).

2.1 $\emptyset \in A$. Then $\inf_{\overline{IS}} A = \emptyset = \inf_{\overline{IR}} A$.

2.2 $\emptyset \notin A$. Since A is not bounded below in IS, we get $\inf_{\overline{IS}} A = \emptyset = \inf_{\overline{IR}} A$. ■

In general, interval calculations are employed to determine sets that include the solution of a given problem. If $\{R, \leq\}$ is an ordered set and the operations for the intervals of IR cannot be performed precisely on a computer, one has to approximate them on a screen $\{S, \leq\}$ of $\{R, \leq\}$. This approximation will be required to have the following general properties, which we shall make precise later:

(a) The result of any computation in the subset IS always has to include the result of the corresponding calculation in IR.

(b) The result of the computation in IS should be as close as possible to the result of the corresponding calculation in IR.

We may seek to achieve these qualitative requirements by defining the operations in IS by means of the semimorphism associated with the monotone upwardly directed rounding $\Diamond : \overline{IR} \to \overline{IS}$. We are now going to describe this process.

The monotone upwardly directed rounding $\Diamond : \overline{IR} \to \overline{IS}$ is defined by the properties

(R1) $\bigwedge_{A \in \overline{IS}} \Diamond A = A,$

(R2) $\bigwedge_{A,B \in \overline{IR}} (A \subseteq B \Rightarrow \Diamond A \subseteq \Diamond B),$

(R3) $\bigwedge_{A \in \overline{IR}} A \subseteq \Diamond A.$

In the following theorem we develop some properties of this rounding.

Theorem 4.19. *Let $\{R, +, \cdot, \leq\}$ be a completely and weakly ordered ringoid with the neutral elements o and e, and $\{S, \leq\}$ a symmetric screen of $\{R, +, \cdot, \leq\}$. Further, let $\{IR, \boxplus, \boxdot\}$ be a ringoid defined by the semimorphism $\square : \mathbb{P}R \to \overline{IR}$. Then:*

(a) $\{\overline{IS}, \subseteq\}$ is a symmetric screen of $\{IR, \boxplus, \boxdot\}$, i.e., we have

$$\textbf{(S3)} \quad [o,o], [e,e] \in IS \quad \wedge \quad \bigwedge_{A=[a_1,a_2] \in IS} \boxminus A = [-a_2, -a_1] \in IS. \qquad (4.5.1)$$

(b) The monotone upwardly directed rounding $\Diamond : \overline{IR} \to \overline{IS}$ is antisymmetric, i.e.,

$$\textbf{(R4)} \quad \bigwedge_{A \in IR} \Diamond(\boxminus A) = \boxminus(\Diamond A).$$

(c) We have

$$\textbf{(R)} \quad \bigwedge_{A=[a_1,a_2] \in IR} \Diamond A = \inf(U(A) \cap IS) = [\bigtriangledown a_1, \bigtriangleup a_2]. \qquad (4.5.2)$$

Proof. (a) With $A = [a_1, a_2] \in IS$ we obtain

$$\begin{aligned} \boxminus A := [-e,-e] \boxdot A := \square(\{-e\} \cdot A) = \square(-A) \\ \underset{(\text{S3})_{IR}, (\text{R1})}{=} -A \underset{\text{Theorem 4.4(a)}}{=} [-a_2, -a_1] \underset{(\text{S3})_S}{\in} IS. \end{aligned}$$

In the next step we prove (c).

(c) With $B = [b_1, b_2] \in IS$, we get for all $A = [a_1, a_2] \in IR$:

$$\begin{aligned} \inf_{IR}(U(A) \cap IS) &\underset{\text{Theorem 4.18(b)}}{=} \inf_{IS}(U(A) \cap IS) \\ &\underset{\text{Theorem 4.18(a)}}{=} [\sup_{B \in U(A) \cap IS} b_1, \inf_{B \in U(A) \cap IS} b_2]. \end{aligned}$$

In general we have

$$[a_1, a_2] \subseteq [b_1, b_2] \Leftrightarrow b_1 \leq a_1 \wedge a_2 \leq b_2.$$

With this and Theorem 1.24, we get:

$$\begin{aligned}\Diamond A &= \inf_{IS}(U(A) \cap IS) = [\sup_{b_1 \in S \wedge b_1 \leq a_1} b_1, \inf_{b_2 \in S \wedge a_2 \leq b_2} b_2]\\ &= [\sup(L(a_1) \cap S), \inf(U(a_2) \cap S)] = [\triangledown a_1, \triangle a_2].\end{aligned}$$

(b) For all $A = [a_1, a_2] \in IR$ we have

$$\begin{aligned}\Diamond(\boxminus A) &\underset{(4.5.1)}{=} \Diamond[-a_2, -a_1] \underset{(4.5.2)}{=} [\triangledown(-a_2), \triangle(-a_1)]\\ &\underset{\text{Theorem 3.4}}{=} [-\triangle a_2, -\triangledown a_1] \underset{(4.5.1)}{=} \boxminus[\triangledown a_1, \triangle a_2]\\ &\underset{(4.5.2)}{=} \boxminus(\Diamond A).\end{aligned}$$

■

The monotone upwardly directed rounding $\Diamond : \overline{IR} \to \overline{IS}$ can be employed as a semimorphism to define operations $\mathbin{\Diamond\!\!\!\!\circ}, \circ \in \{+, \cdot, /\}$ in IS as

(RG)
$$\begin{aligned}\bigwedge_{A,B \in IS} A \mathbin{\Diamond\!\!\!\!\circ} B &:= \Diamond(A \mathbin{\boxed{\circ}} B) := \Diamond(\square(A \circ B))\\ &= \Diamond[\inf(A \circ B), \sup(A \circ B)]\\ &\underset{(4.5.2)}{=} [\triangledown \inf(A \circ B), \triangle \sup(A \circ B)].\end{aligned}$$

These operations have the following properties which by Theorem 1.32 also define them:

(RG1) $\bigwedge_{A,B \in IS} (A \mathbin{\boxed{\circ}} B \in IS \Rightarrow A \mathbin{\Diamond\!\!\!\!\circ} B = A \mathbin{\boxed{\circ}} B)$,

(RG2) $\bigwedge_{A,B,C,D \in IS} (A \mathbin{\boxed{\circ}} B \subseteq C \mathbin{\boxed{\circ}} D \Rightarrow A \mathbin{\Diamond\!\!\!\!\circ} B \subseteq C \mathbin{\Diamond\!\!\!\!\circ} D)$,

(RG3) $\bigwedge_{A,B \in IS} (A \mathbin{\boxed{\circ}} B \subseteq A \mathbin{\Diamond\!\!\!\!\circ} B)$.

In these formulas we assume additionally in the case of division, that $B, D \notin \overline{N}$. For the definition of $\overline{N}$ see Theorem 4.20, to which we now turn.

Theorem 4.20. *Let $\{R, +, \cdot, \leq\}$ be a completely and weakly ordered ringoid with the neutral elements o and e. We assume additionally that $x = -e$ is already unique in R by (D5a) alone*[3]*. Further, let $\{S, \leq\}$ be a symmetric screen of $\{R, +, \cdot, \leq\}$. Consider*

[3] This is the case, for instance, if $\{R, +\}$ is a group.

the semimorphisms $\square : \mathbb{P}R \to \overline{IR}$ *and* $\Diamond : \overline{IR} \to \overline{IS}$. *Then:*

(a) $\{IS, \diamondplus, \diamonddot, \leq\}$ *is a weakly ordered ringoid.*

(b) *If* $\{R, +, \cdot, \leq\}$ *is an ordered ringoid, then* $\{IS, \diamondplus, \diamonddot, \leq\}$ *is also an ordered ringoid.*

(c) *If* $\{R, N, +, \cdot, /, \leq\}$ *is an ordered (resp. weakly ordered) division ringoid, then* $\{IS, \overline{N}, \diamondplus, \diamonddot, \diamondslash, \leq\}$ *with* $\overline{N} := \{A \in IS \mid A \cap N \neq \emptyset\}$ *is also an ordered (resp. weakly ordered) division ringoid.*

(d) $\{IS, \overline{N}, \diamondplus, \diamonddot, \diamondslash, \subseteq\}$ *is inclusion isotonally ordered.*

Proof. By Theorem 4.5 $\{IR, \boxplus, \boxdot\}$ is a ringoid. Theorem 3.5 directly implies the properties (D1,2,3,4,5,7,8,9). (OD5) and (OD6) in IS follow immediately from (OD5) and (OD6) in IR and the monotonicity of the rounding $\Diamond$. It remains to show (D6).

(D6): Let $X = [x, y]$ be any element of IS that fulfils (D5). Then

$$[e, e] \diamondplus [x, y] = \Diamond([e, e] \boxplus [x, y]) = \Diamond([e + x, e + y]) = [o, o]. \tag{4.5.3}$$

$\Diamond : \overline{IR} \to \overline{IS}$ is the upwardly directed rounding with respect to $\subseteq$. If the image of an interval only consists of the single point $o \in R$, then this interval itself must already be this point, i.e., by (4.5.3) we obtain

$$e + x = o \wedge e + y = o.$$

By assumption, there is exactly one element in R that solves (D5a). Thus $x = y = -e$ and $X = [-e, -e]$.

It remains to show (OD1,2,3,4). By Theorem 3.5 it suffices to show that the rounding $\Diamond : \overline{IR} \to \overline{IS}$ is also monotone with respect to $\leq$.

(R2): With $A = [a_1, a_2]$ and $B = [b_1, b_2] \in IR$, we have

$$A \leq B :\Leftrightarrow a_1 \leq b_1 \wedge a_2 \leq b_2 \underset{\text{(R2)}}{\Rightarrow} \nabla a_1 \leq \nabla b_1 \wedge \Delta a_2 \leq \Delta b_2$$
$$\Rightarrow [\nabla a_1, \Delta a_2] \leq [\nabla b_1, \Delta b_2] \underset{\text{(4.5.2)}}{\Rightarrow} \Diamond A \leq \Diamond B. \qquad \blacksquare$$

We shall apply Theorem 4.10 to various interval sets. The first application, given in the following theorem, deals with intervals on a screen of the linearly ordered division ringoid of the real numbers $\mathbb{R}$.

Theorem 4.21. *Let* $\{\mathbb{R}, \{0\}, +, \cdot, /, \leq\}$ *be the completely and linearly ordered ringoid of real numbers, and* $\{S, \leq\}$ *a symmetric screen of* $\mathbb{R}$. *Consider the semimorphisms* $\square : \mathbb{P}\mathbb{R} \to \overline{I\mathbb{R}}$ *and* $\Diamond : \overline{I\mathbb{R}} \to \overline{IS}$. *Then* $\{IS, N, \diamondplus, \diamonddot, \diamondslash, \leq, \subseteq\}$ *with* $N := \{A \in IS \mid 0 \in A\}$ *is an ordered division ringoid with respect to* $\leq$ *and an inclusion-*

isotonally ordered monotone upper screen division ringoid of $I\mathbb{R}$ with respect to $\subseteq$. The neutral elements are $[0,0]$ and $[1,1]$. ■

Because of the great importance of the intervals on a screen of the real numbers $\mathbb{R}$, we derive explicit formulas for the operations in IS.

Let $\{\mathbb{R}, \{0\}, +, \cdot, /, \leq\}$ be the linearly ordered division ringoid of real numbers and $A = [a_1, a_2]$, $B = [b_1, b_2] \in I\mathbb{R}$. Corollary 4.11 summarizes the result of all operations $\boxed{\circ}$, $\circ \in \{+, -, \cdot, /\}$ in $I\mathbb{R}$ by means of the formula

$$A \boxed{\circ} B := \square(A \circ B) = \left[\min_{i,j=1,2}(a_i \circ b_j), \max_{i,j=1,2}(a_i \circ b_j)\right].$$

By the definition of the operations in IS and by (4.5.2), we obtain for all $A = [a_1, a_2]$, $B = [b_1, b_2] \in IS$ that

$$A \mathbin{\Diamond\!\!\!\!\!\circ\;} B := \Diamond(A \boxed{\circ} B) = \left[\bigtriangledown \min_{i,j=1,2}(a_i \circ b_j), \bigtriangleup \max_{i,j=1,2}(a_i \circ b_j)\right].$$

Since $\bigtriangledown : \mathbb{R} \to S$ and $\bigtriangleup : \mathbb{R} \to S$ are monotone mappings, we obtain

$$A \mathbin{\Diamond\!\!\!\!\!\circ\;} B := \Diamond(A \boxed{\circ} B) = \left[\min_{i,j=1,2}(a_i \mathbin{\bigtriangledown\!\!\!\!\!\circ\;} b_j), \max_{i,j=1,2}(a_i \mathbin{\bigtriangleup\!\!\!\!\!\circ\;} b_j)\right].$$

For execution of the operations $\mathbin{\Diamond\!\!\!\!\!\circ\;}$, $\circ \in \{+, -, \cdot, /\}$, in IS on a computer, we display the following formulas and Tables 4.3 and 4.4. These are obtained by employing the preceding equation, Theorems 4.5 and 4.10, and especially Tables 4.1 and 4.2.

	$A = [a_1, a_2]$	$B = [b_1, b_2]$	$A \mathbin{\Diamond\!\!\!\!\!\cdot\;} B = [\min_{i,j=1,2}(a_i \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_j), \max_{i,j=1,2}(a_i \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_j)]$
1	$A \geq [0,0]$	$B \geq [0,0]$	$[a_1 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_1, a_2 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_2]$
2	$A \geq [0,0]$	$B \leq [0,0]$	$[a_2 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_1, a_1 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_2]$
3	$A \geq [0,0]$	$b_1 < 0 < b_2$	$[a_2 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_1, a_2 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_2]$
4	$A \leq [0,0]$	$B \geq [0,0]$	$[a_1 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_2, a_2 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_1]$
5	$A \leq [0,0]$	$B \leq [0,0]$	$[a_2 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_2, a_1 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_1]$
6	$A \leq [0,0]$	$b_1 < 0 < b_2$	$[a_1 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_2, a_1 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_1]$
7	$a_1 < 0 < a_2$	$B \geq [0,0]$	$[a_1 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_2, a_2 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_2]$
8	$a_1 < 0 < a_2$	$B \leq [0,0]$	$[a_2 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_1, a_1 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_1]$
9	$a_1 < 0 < a_2$	$b_1 < 0 < b_2$	$[\min(a_1 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_2, a_2 \mathbin{\bigtriangledown\!\!\!\!\!\cdot\;} b_1), \max(a_1 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_1, a_2 \mathbin{\bigtriangleup\!\!\!\!\!\cdot\;} b_2)]$

Table 4.3. Execution of the multiplication in IS.

The formulas in the Tables 4.3 and 4.4 show, in particular, that the operations $\mathbin{\Diamond\!\!\!\!\!\circ\;}$, $\circ \in \{+, -, \cdot, /\}$, in IS are executable on a computer if the operations $\mathbin{\bigtriangledown\!\!\!\!\!\circ\;}$ and $\mathbin{\bigtriangleup\!\!\!\!\!\circ\;}$,

	$A=[a_1,a_2]$	$B=[b_1,b_2]$	$A \mathbin{\Diamond\kern-0.62em{\scriptscriptstyle /}\kern0.3em} B = [\min_{i,j=1,2}(a_i \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_j), \max_{i,j=1,2}(a_i \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_j)]$
1	$A \geq [0,0]$	$0 < b_1 \leq b_2$	$[a_1 \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_2, a_2 \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_1]$
2	$A \geq [0,0]$	$b_1 \leq b_2 < 0$	$[a_2 \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_2, a_1 \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_1]$
4	$A \leq [0,0]$	$0 < b_1 \leq b_2$	$[a_1 \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_1, a_2 \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_2]$
5	$A \leq [0,0]$	$b_1 \leq b_2 < 0$	$[a_2 \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_1, a_1 \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_2]$
7	$a_1 < 0 < a_2$	$0 < b_1 \leq b_2$	$[a_1 \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_1, a_2 \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_1]$
8	$a_1 < 0 < a_2$	$b_1 \leq b_2 < 0$	$[a_2 \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_2, a_1 \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b_2]$

Table 4.4. Execution of division in *IS* where B does not contain 0.

$\circ \in \{+,-,\cdot,/\}$, for elements of S are available. These latter operations are defined by the following formulas:

$$\bigwedge_{a,b\in S} a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle \circ}\kern0.25em} b := \triangledown(a \circ b) \quad \wedge \quad \bigwedge_{a,b\in S} a \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle \circ}\kern0.25em} b := \vartriangle(a \circ b).$$

Here for division we additionally assume that $b \neq 0$.

With regard to dependency and economy we mention the following: In order to perform the eight operations $\triangledown\kern-0.62em{\scriptscriptstyle \circ}\kern0.25em$ and $\vartriangle\kern-0.62em{\scriptscriptstyle \circ}\kern0.25em$, $\circ \in \{+,-,\cdot,/\}$, on a computer, it is sufficient if three of them, for instance, $\triangledown\kern-0.62em{\scriptscriptstyle +}\kern0.25em$, $\triangledown\kern-0.62em{\scriptscriptstyle \cdot}\kern0.25em$, $\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em$, or an equivalent triple, are available. This is a consequence of the fact that subtraction can be expressed by addition and the result of Theorem 3.4, which asserts that

$$\bigwedge_{a\in\mathbb{R}} \triangledown a = -\vartriangle(-a) \quad \wedge \quad \vartriangle a = -\triangledown(-a).$$

For instance, we obtain the following list of formulas:

$$\begin{aligned}
a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle +}\kern0.25em} b &= \triangledown(a+b),\\
a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle -}\kern0.25em} b &= \triangledown(a+(-b)) = a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle +}\kern0.25em}(-b),\\
a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle \cdot}\kern0.25em} b &= \triangledown(a \cdot b),\\
a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b &= \triangledown(a/b),\\
a \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle +}\kern0.25em} b &= -\triangledown(-(a+b)) = -((-a) \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle +}\kern0.25em}(-b)),\\
a \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle -}\kern0.25em} b &= -\triangledown(-(a+(-b))) = -((-a) \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle +}\kern0.25em} b),\\
a \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle \cdot}\kern0.25em} b &= -\triangledown(-(a \cdot b)) = -((-a) \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle \cdot}\kern0.25em} b) = -(a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle \cdot}\kern0.25em}(-b)),\\
a \mathbin{\vartriangle\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b &= -\triangledown(-(a/b)) = -((-a) \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em} b) = -(a \mathbin{\triangledown\kern-0.62em{\scriptscriptstyle /}\kern0.25em}(-b)).
\end{aligned}$$

We have already discussed the meaning of the monotone directed roundings $\triangledown$ and $\vartriangle$ in Section 3.5. Theorems 1.30 and 3.4 show, in particular, that in a linearly ordered ringoid every monotone rounding can be expressed in terms of either $\bigtriangledown$ or $\bigtriangleup$. The formula system that we have derived above shows additionally that to perform all interval operations in *IS* either $\bigtriangledown$ or $\bigtriangleup$ suffices.

Addition and multiplication in $\mathbb{R}$ are associative. Consequently, addition and multiplication in $\mathbb{PR}$ and, referring to Remark 4.12, those in $I\mathbb{R}$ are also associative. These associativity properties, however, are no longer valid in *IS*. We show this by means of a simple example for addition.

Without loss of generality we may assume that S is a floating-point system with the number representation $m \cdot b^e$ with $b = 10$, $m = -0.9(0.1)0.9$ and $e \in \{-1, 0, 1\}$. Throughout the example, $A, B \in IS$.

Let $A := [0.3, 0.6]$, $B := [0.3, 0.5]$, $C := [0.3, 0.4]$. Then

$$\begin{aligned} A \mathbin{\Diamond\kern-0.72em+} (B \mathbin{\Diamond\kern-0.72em+} C) &= [0.3, 0.6] \mathbin{\Diamond\kern-0.72em+} [0.6, 0.9] = [0.9, 0.2 \cdot 10], \\ (A \mathbin{\Diamond\kern-0.72em+} B) \mathbin{\Diamond\kern-0.72em+} C &= (\Diamond [0.6, 1.1]) \mathbin{\Diamond\kern-0.72em+} [0.3, 0.4] = [0.6, 0.2 \cdot 10] \mathbin{\Diamond\kern-0.72em+} [0.3, 0.4] \\ &= \Diamond [0.9, 2.4] = [0.9, 0.3 \cdot 10]. \end{aligned}$$

That is

$$A \mathbin{\Diamond\kern-0.72em+} (B \mathbin{\Diamond\kern-0.72em+} C) \neq (A \mathbin{\Diamond\kern-0.72em+} B) \mathbin{\Diamond\kern-0.72em+} C.$$

These characteristics of the operations in *IS* are simple consequences of the fact that the associative laws fail in $\{S, \mathbin{\bigtriangledown\kern-0.75em\circ}\}$ and $\{S, \mathbin{\bigtriangleup\kern-0.75em\circ}\}$, $\circ \in \{+, \cdot\}$.

All the properties that we demonstrated in Theorem 1.35 hold for the rounding $\Diamond : \overline{I\mathbb{R}} \to \overline{IS}$ and the operations in $\{IS, N, \mathbin{\Diamond\kern-0.72em+}, \mathbin{\Diamond\kern-0.55em\cdot}, \mathbin{\Diamond\kern-0.6em/}, \leq, \subseteq\}$. In particular, the inequality

$$\Diamond (A \mathbin{\Box\kern-0.6em\circ} B) \subseteq (\Diamond A) \mathbin{\Diamond\kern-0.6em\circ} (\Diamond B)$$

is valid for all $A, B \in I\mathbb{R}$. We show by means of a simple example that the strict inclusion sign can occur. Thus there exists no homomorphism between the operations $\mathbin{\Box\kern-0.6em\circ}$ in $I\mathbb{R}$ and $\mathbin{\Diamond\kern-0.6em\circ}$ in *IS* for all $\circ \in \{+, -, \cdot, /\}$. For example we use the floating-point system defined above.

Example 4.22. Let $A := [0.36, 0.54]$, $B := [0.35, 0.45] \in I\mathbb{R}$. Then $A \boxplus B = [0.71, 0, 99]$. By (4.5.2) we get $\Diamond A = [0.3, 0.6]$, $\Diamond B = [0.3, 0.5]$, and $\Diamond (A \boxplus B) = [0.7, 0.1 \cdot 10]$. Therefore, $(\Diamond A) \mathbin{\Diamond\kern-0.72em+} (\Diamond B) = \Diamond ((\Diamond A) \boxplus (\Diamond B)) = \Diamond [0.6, 1.1] = [0.6, 0.2 \cdot 10]$ and $\Diamond (A \boxplus B) = [0.7, 0.1 \cdot 10] \subset [0.6, 0.2 \cdot 10] = (\Diamond A) \mathbin{\Diamond\kern-0.72em+} (\Diamond B)$.

The different cases of interval multiplication and division represented by the Tables 4.3 and 4.4 are redundant, i.e., they are not distinct and partly overlap each other. This is inconvenient for a computer realization of these operations. Tables 4.5 and 4.6 give a representation of interval multiplication and division where the redundancy is eliminated.

	$0 \le b_1$	$b_1 < 0 \le b_2$	$b_2 < 0$
$0 \le a_1$	$[a_1 \triangledown b_1, a_2 \vartriangle b_2]$	$[a_2 \triangledown b_1, a_2 \vartriangle b_2]$	$[a_2 \triangledown b_1, a_1 \vartriangle b_2]$
$a_1 < 0 \le a_2$	$[a_1 \triangledown b_2, a_2 \vartriangle b_2]$	$[\min(a_1 \triangledown b_2, a_2 \triangledown b_1),$ $\max(a_1 \vartriangle b_1, a_2 \vartriangle b_2)]$	$[a_2 \triangledown b_1, a_1 \vartriangle b_1]$
$a_2 < 0$	$[a_1 \triangledown b_2, a_2 \vartriangle b_1]$	$[a_1 \triangledown b_2, a_1 \vartriangle b_1]$	$[a_2 \triangledown b_2, a_1 \vartriangle b_1]$

Table 4.5. The nine cases of interval multiplication with $A, B \in IS$.

	$0 < b_1$	$b_2 < 0$
$0 \le a_1$	$[a_1 \triangledown b_2, a_2 \vartriangle b_1]$	$[a_2 \triangledown b_2, a_1 \vartriangle b_1]$
$a_1 < 0 \le a_2$	$[a_1 \triangledown b_1, a_2 \vartriangle b_1]$	$[a_2 \triangledown b_2, a_1 \vartriangle b_2]$
$a_2 < 0$	$[a_1 \triangledown b_1, a_2 \vartriangle b_2]$	$[a_2 \triangledown b_1, a_1 \vartriangle b_2]$

Table 4.6. The six cases of interval division $A \diamonddiv B$ with $A, B \in IS$ and $0 \notin B$.

4.6 Interval Matrices and Interval Vectors on a Screen

We begin with the following characterization of IM_nS.

Theorem 4.23. *Let $\{\mathbb{R}, +, \cdot, \le\}$ be the completely and linearly ordered ringoid of real numbers and $\{S, \le\}$ a symmetric screen of $\mathbb{R}$. Consider the completely ordered ringoid of matrices $\{M_n\mathbb{R}, +, \cdot, \le\}$ with the neutral elements O and E and the semimorphisms $\square : \mathbb{P}M_n\mathbb{R} \to IM_n\mathbb{R}$ and $\Diamond : IM_n\mathbb{R} \to IM_nS$. Then $\{IM_nS, \diamondplus, \diamonddot, \le, \subseteq\}$ is a completely ordered ringoid with respect to $\le$. The neutral elements are $[O, O]$ and $[E, E]$. With respect to $\subseteq$, IM_nS is an inclusion-isotonally ordered monotone upper screen ringoid of $IM_n\mathbb{R}$.*

Proof. Theorem 3.8 implies that M_nS is a symmetric screen of $M_n\mathbb{R}$. By Theorem 4.5, $\{IM_n\mathbb{R}, \boxplus, \boxdot, \le\}$ is a ringoid, and by Theorem 4.20. $\{IM_nS, \diamondplus, \diamonddot, \le, \subseteq\}$ is an ordered ringoid with respect to $\le$. By Theorem 3.5 it is an inclusion-isotonally ordered monotone upper screen ringoid of $IM_n\mathbb{R}$. ■

The operations $\diamondplus$ in IM_nS are not executable on a computer. Therefore we are now going to express these operations in terms of computer executable formulas. To do this, let us once more consider the ringoid $\{M_nI\mathbb{R}, \oplus, \odot, \le, \subseteq\}$. According to Theorem 4.15, this ringoid is isomorphic to $\{IM_n\mathbb{R}, \boxplus, \boxdot, \le, \subseteq\}$. The isomorphism is expressed by the equality

$$\bigwedge_{\boldsymbol{A},\boldsymbol{B} \in M_nI\mathbb{R}} \chi\boldsymbol{A} \mathbin{\boxed{\circ}} \chi\boldsymbol{B} = \chi(\boldsymbol{A} \circledcirc \boldsymbol{B}), \circ \in \{+, \cdot\}. \tag{4.6.1}$$

Here the mapping χ: $M_n I\mathbb{R} \to IM_n\mathbb{R}$ is defined by (4.3.1) in Section 4.3.

Using the rounding $\Diamond : \overline{I\mathbb{R}} \to \overline{IS}$ and appealing to Theorem 3.9, a rounding $\Diamond : M_n I\mathbb{R} \to M_n IS$ and operations in $M_n IS$ are defined by

$$\bigwedge_{\boldsymbol{A}=(A_{ij})\in M_n I\mathbb{R}} \Diamond\boldsymbol{A} := (\Diamond A_{ij}),$$

$$\bigwedge_{\boldsymbol{A},\boldsymbol{B}\in M_n IS} \boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B} := \Diamond(\boldsymbol{A} \circledcirc \boldsymbol{B}),\ \circ \in \{+,\cdot\}.$$

For further purposes, let us denote the matrix $\boldsymbol{A} \circledcirc \boldsymbol{B} \in M_n I\mathbb{R}$ by

$$(C_{ij}) := \boldsymbol{A} \circledcirc \boldsymbol{B} \text{ with } C_{ij} = [c_{ij}^1, c_{ij}^2] \text{ and } c_{ij}^1, c_{ij}^2 \in \mathbb{R}. \tag{4.6.2}$$

We are interested in the expression $\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B})$ and obtain for it

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \chi(\Diamond C_{ij}) \underset{\text{Theorem 4.19(c)}}{=} \chi(\inf U(C_{ij}) \cap IS)\textbf{.}$$

If we exchange the componentwise infima with the infimum of a matrix, we obtain

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \chi(\inf(U(C_{ij}) \cap IS)).$$

Since the inclusion and the intersection in $M_n I\mathbb{R}$ are defined componentwise, we may exchange the matrix parenthesis with the upper bounds and obtain

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \chi(\inf U\{([c_{ij}^1, c_{ij}^2]) \cap M_n IS\}),$$

or

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \chi(\inf U\{\boldsymbol{A} \circledcirc \boldsymbol{B} \cap M_n IS\}).$$

If we now apply the mapping χ, we obtain

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \inf U\{\chi(\boldsymbol{A} \circledcirc \boldsymbol{B}) \cap IM_n S\} \underset{(4.6.1)}{=} \inf U\{(\chi\boldsymbol{A} \boxdot \chi\boldsymbol{B}) \cap IM_n S\}.$$

Since the rounding $\Diamond : IM_n\mathbb{R} \to IM_n S$ has the property

$$\bigwedge_{\boldsymbol{A}\in IM_n\mathbb{R}} \Diamond\boldsymbol{A} = \inf(U(\boldsymbol{A}) \cap IM_n S),$$

we obtain

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \Diamond(\chi\boldsymbol{A} \boxdot \chi\boldsymbol{B}).$$

By the definition of the operation $\mathbin{\Diamond\!\!\!\!\circ}$ in $IM_n S$ this finally leads to

$$\chi(\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{B}) = \chi\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\circ} \chi\boldsymbol{B}. \tag{4.6.3}$$

This states the fact that the mapping χ also represents an isomorphism between the ringoid $\{M_nIS, \diamondplus, \diamonddot, \leq, \subseteq\}$ defined by Theorem 3.9 and the ringoid $\{IM_nS, \diamondplus, \diamonddot, \leq, \subseteq\}$ studied in Theorem 4.23.

The isomorphism (4.6.3) reduces the operations in IM_nS, which are not computer executable, to the operations in M_nIS. We analyse these operations more closely.

By equation (4.3.1) in Section 4.3, operations in M_nIR have been defined by

$$\bigwedge_{\boldsymbol{A}=(A_{ij}),\boldsymbol{B}=(B_{ij})\in M_nIR} \boldsymbol{A} \oplus \boldsymbol{B} := (A_{ij} \boxplus B_{ij}) \ \wedge\ \boldsymbol{A} \odot \boldsymbol{B} := \left(\boxed{\sum}_{\nu=1}^{n} (A_{i\nu} \boxdot B_{\nu j}) \right).$$

In this section operations in M_nIS are defined by

$$\bigwedge_{\boldsymbol{A}=(A_{ij}),\boldsymbol{B}=(B_{ij})\in M_nIS} \boldsymbol{A} \diamondplus \boldsymbol{B} := \Diamond(\boldsymbol{A} \oplus \boldsymbol{B}) \quad \wedge \quad \boldsymbol{A} \diamonddot \boldsymbol{B} := \Diamond(\boldsymbol{A} \odot \boldsymbol{B})$$

with the rounding $\Diamond \boldsymbol{A} := (\Diamond A_{ij})$. This leads to the following formulas for the operations in M_nIS:

$$\boldsymbol{A} \diamondplus \boldsymbol{B} = (\Diamond(A_{ij} \boxplus B_{ij})) = (A_{ij} \diamondplus B_{ij}), \tag{4.6.4}$$

$$\boldsymbol{A} \diamonddot \boldsymbol{B} = \left(\Diamond \boxed{\sum}_{\nu=1}^{n} (A_{i\nu} \boxdot B_{\nu j}) \right). \tag{4.6.5}$$

These operations are executable on a computer. The componentwise addition in (4.6.4) can be performed by means of the addition in IS. The multiplications in (4.6.5) are to be executed using Table 4.1. Then the lower bounds and the upper bounds are to be added in $\mathbb{R}$. Finally the rounding $\Diamond : \overline{IR} \to \overline{IS}$ has to be executed.

With $A_{ij} = [a_{ij}^1, a_{ij}^2], B_{ij} = [b_{ij}^1, b_{ij}^2] \in IS$, (4.6.5) can be written in a more explicit form:

$$\boldsymbol{A} \diamonddot \boldsymbol{B} = \left(\left[\triangledown \sum_{\nu=1}^{n} \min_{r,s=1,2} (a_{i\nu}^r b_{\nu j}^s), \triangle \sum_{\nu=1}^{n} \max_{r,s=1,2} (a_{i\nu}^r b_{\nu j}^s) \right] \right).$$

Here the products $a_{i\nu}^r b_{\nu j}^s$ are elements of $\mathbb{R}$ (and in general not of S). The summands (products of double length) are to be correctly accumulated in $\mathbb{R}$ by the exact scalar product. See Chapter 8. Finally the sum of products is rounded only once by $\triangledown$ (resp. $\triangle$) from $\mathbb{R}$ into S.

Since a semimorphism of a semimorphism is a semimorphism, this result can be used not only to describe the structure on a screen but also on one or more coarser screens. Accordingly, the passage that we made from $IM_n\mathbb{R}$ to IM_nS can also be performed from $IM_n\mathbb{R}$ to IM_nD or from IM_nD to IM_nS. This results in isomorphisms between the sets $IM_n\mathbb{R}$, IM_nD, IM_nS and $M_nI\mathbb{R}$, M_nID, M_nIS, respectively.

We are now going to illustrate these results. Let $\mathbb{R}$ again denote the linearly ordered set of real numbers and S the subset of single precision floating-point numbers of a given computer. Figure 4.2 shows the spaces that were shown to be ringoids and lists the theorems that proved the required properties. In particular, we display the isomorphism between the ringoids $IM_n\mathbb{R}$, IM_nD, IM_nS and $M_nI\mathbb{R}$, M_nID, M_nIS, respectively. We have already noted that the operations in the latter ringoids defined by Theorem 3.9 are executable on a computer.

$$\begin{tikzcd}
 & \mathbb{R} \arrow[r, "3.7"] \arrow[dl, "2.5"'] & D \arrow[r, "3.7"] & S \\
\mathbb{P}\mathbb{R} \arrow[r, "4.5"] & I\mathbb{R} \arrow[r, "4.20"] \arrow[dd, "2.6"] & \underline{ID} \arrow[r, "4.20"] \arrow[d, "2.6"] & \underline{IS} \arrow[d, "2.6"] \\
 & & \underline{M_nID} \arrow[d, phantom, "\vee"] & \underline{M_nIS} \arrow[d, phantom, "\vee"] \\
 & M_nI\mathbb{R} \arrow[r, "3.9"] \arrow[d, leftrightarrow, "\chi"', "4.15"] & \underline{M_nID} \arrow[r, "3.9"] \arrow[d, leftrightarrow, "\chi"'] & \underline{M_nIS} \arrow[d, leftrightarrow, "\chi"'] \\
\mathbb{P}M_n\mathbb{R} \arrow[r, "4.5"] & IM_n\mathbb{R} \arrow[r, "4.23"] & IM_nD \arrow[r, "4.23"] & IM_nS \\
 & M_n\mathbb{R} \arrow[ul, "2.5"] \arrow[r, "3.9"] & M_nD \arrow[r, "3.9"] & M_nS
\end{tikzcd}$$

Figure 4.2. Ringoids with interval matrices.

Within the sets M_nID and M_nIS, operations and a ringoid can also be defined by the conventional method defining operations for matrices using the operations in ID (resp. IS). We show that addition in these ringoids is identical to the addition defined by Theorem 3.9. The multiplication defined by Theorem 2.6 delivers upper bounds of the result of the multiplication defined by Theorem 3.9.

Using (4.6.4) and (4.6.5), these observations can be directly verified by means of the following formulas, wherein $\boldsymbol{A} = (A_{ij}), \boldsymbol{B} = (B_{ij}) \in M_nIS$:

$$\begin{aligned}
\boldsymbol{A} \mathbin{\Diamond\kern-0.72em+\kern0.2em} \boldsymbol{B} &= (\Diamond(A_{ij} \boxplus B_{ij})) = (A_{ij} \mathbin{\Diamond\kern-0.72em+\kern0.2em} B_{ij}), \\
\boldsymbol{A} \mathbin{\Diamond\kern-0.6em\cdot\kern0.3em} \boldsymbol{B} &= \left(\Diamond \overset{n}{\underset{\nu=1}{\boxed{\sum}}} A_{i\nu} \boxdot B_{\nu j} \right) \subseteq \left(\overset{n}{\underset{\nu=1}{\Diamond\kern-1em\sum}} A_{i\nu} \mathbin{\Diamond\kern-0.6em\cdot\kern0.3em} B_{\nu j} \right).
\end{aligned} \tag{4.6.6}$$

The last inequality is a consequence of (R3) applied to the rounding $\Diamond : \overline{IR} \to \overline{IS}$ and the inclusion-isotony of all operations in IS:

$$A_{i\nu} \boxdot B_{\nu j} \subseteq \Diamond(A_{i\nu} \boxdot B_{\nu j}) = A_{i\nu} \mathbin{\Diamond\kern-0.6em\cdot\kern0.3em} B_{\nu j}, A \boxplus B \subseteq \Diamond(A \boxplus B) = A \mathbin{\Diamond\kern-0.72em+\kern0.2em} B$$

$$\Rightarrow \overset{n}{\underset{\nu=1}{\boxed{\sum}}} A_{i\nu} \boxdot B_{\nu j} \subseteq \overset{n}{\underset{\nu=1}{\Diamond\kern-1em\sum}} A_{i\nu} \mathbin{\Diamond\kern-0.6em\cdot\kern0.3em} B_{\nu j}.$$

The expressions on the right hand side of (4.6.6) represent the sum (resp. the product) defined in M_nIS by the conventional method of the definition of operations for matrices (Theorem 2.6). On computers practical calculations in M_nID and M_nIS are, for simplicity, often done by use of the operations defined by this method. The multiplication defined by semimorphism (4.6.5), however, provides optimal accuracy for each component. It delivers the least upper bound of the matrix $\boldsymbol{A} \odot \boldsymbol{B}$ in M_nIS.

In Figure 4.2 double-headed arrows indicate isomorphisms, horizontal arrows denote semimorphisms, and vertical arrows indicate a conventional definition of the arithmetic operations. A slanting arrow shows where the power set operations come from. The sign $\smile$ used in Figure 4.2 is intended to illustrate the inequality (4.6.6). Interval sets, the operations of which are executable on a computer, are underlined in Figure 4.2. The numbers on the arrows drawn in the figure indicate the theorems that provide the corresponding properties.

We are now going to consider vectoids. Let $\{V, \leq\}$ be a complete lattice, $\{S, \leq\}$ a screen of $\{V, \leq\}$, and $\{IS, \subseteq\}$ the set of intervals over V with bounds in S of the form

$$\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2] = \{\boldsymbol{x} \in V \mid \boldsymbol{a}_1, \boldsymbol{a}_2 \in S, \boldsymbol{a}_1 \leq \boldsymbol{x} \leq \boldsymbol{a}_2\} \text{ with } \boldsymbol{a}_1 \leq \boldsymbol{a}_2.$$

Then $IS \subseteq IV$ and $\overline{IS} := IS \cup \{\varnothing\} \subseteq \overline{IV} := IV \cup \{\varnothing\}$. By Theorem 4.18, $\{\overline{IS}, \subseteq\}$ is a screen of $\{\overline{IV}, \subseteq\}$. We consider the monotone upwardly directed rounding $\Diamond : \overline{IV} \to \overline{IS}$, which is defined by the properties

(R1) $\bigwedge_{\boldsymbol{A} \in \overline{IS}} \Diamond \boldsymbol{A} = \boldsymbol{A}$,

(R2) $\bigwedge_{\boldsymbol{A}, \boldsymbol{B} \in \overline{IV}} (\boldsymbol{A} \subseteq \boldsymbol{B} \Rightarrow \Diamond \boldsymbol{A} \subseteq \Diamond \boldsymbol{B})$,

(R3) $\bigwedge_{\boldsymbol{A} \in \overline{IV}} \boldsymbol{A} \subseteq \Diamond \boldsymbol{A}$.

The following theorem holds.

Theorem 4.24. *Let $\{V, R, \leq\}$ be a completely and weakly ordered vectoid, and let $\{S, \leq\}$ be a symmetric screen of $\{V, R, \leq\}$. Further, let $\{IV, IR, \leq, \subseteq\}$ be the vectoid defined by the semimorphism $\square : \mathbb{P}V \to \overline{IV}$. Then setting $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2]$, we have:*

(a) *$\{\overline{IS}, \subseteq\}$ is a symmetric screen of $\{\overline{IV}, \subseteq\}$, i.e., we have*

(S3) $[\boldsymbol{o}, \boldsymbol{o}]$, $([\boldsymbol{e}, \boldsymbol{e}]) \in IS \wedge \bigwedge_{\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2] \in IS} \boxminus \boldsymbol{A} = [-\boldsymbol{a}_2, -\boldsymbol{a}_1] \in IS.$

(b) *The monotone upwardly directed rounding $\Diamond : \overline{IV} \to \overline{IS}$ is antisymmetric, i.e.,*

(R4) $\bigwedge_{\boldsymbol{A} \in IV} \Diamond(\boxminus \boldsymbol{A}) = \boxminus(\Diamond \boldsymbol{A}).$

(c) *We have*

$$\textbf{(R)} \quad \bigwedge_{\boldsymbol{A} \in IV} \Diamond \boldsymbol{A} = \inf(U(\boldsymbol{A}) \cap IS) = [\nabla \boldsymbol{a}_1, \triangle \boldsymbol{a}_2]. \tag{4.6.7}$$

Proof. The proof of this theorem is completely analogous to that of Theorem 4.19. ■

Now we use the monotone upwardly directed rounding $\Diamond : \overline{IV} \to \overline{IS}$ to define inner and outer operations $\mathbin{\Diamond\mkern-13mu\circ}$ in IS in terms of its associated semimorphism, which by Theorem 4.6 has the following properties:

$$\textbf{(RG)} \qquad \bigwedge_{\boldsymbol{A},\boldsymbol{B}\in IS} \boldsymbol{A} \mathbin{\Diamond\mkern-13mu\circ} \boldsymbol{B} := \Diamond(\boldsymbol{A} \boxdot \boldsymbol{B}) = \Diamond(\square(\boldsymbol{A} \circ \boldsymbol{B}))$$

$$\underset{(4.6.7)}{=} [\nabla \inf(\boldsymbol{A} \circ \boldsymbol{B}), \triangle \sup(\boldsymbol{A} \circ \boldsymbol{B})], \text{ for } \circ \in \{+, \cdot\},$$

and with $IT \subseteq IR$,

$$\bigwedge_{A\in IT} \bigwedge_{\boldsymbol{A}\in IS} A \mathbin{\Diamond\mkern-13mu\cdot} \boldsymbol{A} := \Diamond(A \boxdot \boldsymbol{A}) = \Diamond(\square(A \cdot \boldsymbol{A}))$$

$$\underset{(4.6.7)}{=} [\nabla \inf(A \cdot \boldsymbol{A}), \triangle \sup(A \cdot \boldsymbol{A})].$$

The following theorem gives a characterization of $\{IS, IT, \leq, \subseteq\}$.

Theorem 4.25. *Let $\{V, R, \leq\}$ be a completely and weakly ordered vectoid with the neutral elements $\boldsymbol{o}$ and $\boldsymbol{e}$ if a multiplication exists, and let $\{S, \leq\}$ be a symmetric screen of $\{V, R, \leq\}$. Further, let $\{IT, \mathbin{\Diamond\mkern-13mu+}, \mathbin{\Diamond\mkern-13mu\cdot}\}$ be a monotone upper screen ringoid of $\{IR, \boxplus, \boxdot\}$ with respect to $\subseteq$. Consider the two semimorphisms $\square : \mathbb{P}V \to \overline{IV}$ and $\Diamond : \overline{IV} \to \overline{IS}$. Then $\{IS, IT, \leq, \subseteq\}$ is a weakly ordered vectoid with respect to $\leq$. It is multiplicative if $\{V, R, \leq\}$ is. The neutral elements are $[\boldsymbol{o}, \boldsymbol{o}]$ and $[\boldsymbol{e}, \boldsymbol{e}]$. With respect to $\subseteq$, $\{IS, IT, \leq, \subseteq\}$ is an inclusion-isotonally ordered monotone upper screen vectoid of $\{IV, IR, \leq, \subseteq\}$.*

If $\{V, R, \leq\}$ is an ordered vectoid, then $\{IS, IT, \leq, \subseteq\}$ is also an ordered vectoid with respect to $\leq$.

Proof. The proof is an immediate consequence of Theorem 3.11. ■

A special application of this theorem is the set of interval vectors on a screen of the linearly ordered ringoid of real numbers with outer multiplication by elements of the ringoids $\{IS, \mathbin{\Diamond\mkern-13mu+}, \mathbin{\Diamond\mkern-13mu\cdot}, \leq, \subseteq\}$ or $\{IM_nS, \mathbin{\Diamond\mkern-13mu+}, \mathbin{\Diamond\mkern-13mu\cdot}, \leq, \subseteq\}$. The following corollary describes this application.

Corollary 4.26. *Let $\{\mathbb{R}, +, \cdot, \leq\}$ be the linearly ordered ringoid of real numbers, and $\{S, \leq\}$ a symmetric screen of $\mathbb{R}$. Consider the ordered vectoids $\{V_n\mathbb{R}, \mathbb{R}, \leq\}$, $\{M_n\mathbb{R}, \mathbb{R}, \leq\}$, and $\{V_n\mathbb{R}, M_n\mathbb{R}, \leq\}$ as well as the semimorphisms $\square : \mathbb{P}V_n\mathbb{R} \to \overline{IV_n\mathbb{R}}$, $\Diamond : \overline{IV_n\mathbb{R}} \to \overline{IV_nS}$ and the semimorphisms $\square : \mathbb{P}M_n\mathbb{R} \to \overline{IM_n\mathbb{R}}$, $\Diamond : \overline{IM_n\mathbb{R}} \to \overline{IM_nS}$. Then $\{IV_nS, IS, \leq, \subseteq\}$, $\{IM_nS, IS, \leq, \subseteq\}$, and $\{IV_nS, IM_nS, \leq, \subseteq\}$ are ordered vectoids with respect to $\leq$. IM_nS is multiplicative. With respect to $\subseteq$, all of these vectoids are inclusion-isotonally ordered monotone upper screen vectoids of $IV_n\mathbb{R}$ resp. $IM_n\mathbb{R}$.* ■

The situation is similar to that which arose for matrix structures. The operations within the vectoids $\{IV_nS, IS\}$, $\{IM_nS, IS\}$, $\{IV_nS, IM_nS\}$ cannot in general be executed on a computer since they are based on the corresponding powerset operations.

Therefore, we are now going to express these operations in terms of computer executable formulas. By Theorem 4.17 we saw that under the assumption that R is a completely and linearly ordered ringoid, there exist isomorphisms between the following pairs of vectoids:

$$\begin{aligned}
\{V_nIR, IR, \leq\} &\leftrightarrow \{IV_nR, IR, \leq\},\\
\{M_nIR, IR, \leq\} &\leftrightarrow \{IM_nR, IR, \leq\},\\
\{V_nIR, M_nIR, \leq\} &\leftrightarrow \{IV_nR, IM_nR, \leq\}.
\end{aligned}$$

The algebraic isomorphisms are expressed by the relations

$$\begin{aligned}
\psi(A \odot \boldsymbol{a}) &= A \boxdot \psi\boldsymbol{a}, & \psi(\boldsymbol{a} \oplus \boldsymbol{b}) &= \psi\boldsymbol{a} \boxplus \psi\boldsymbol{b},\\
\chi(A \odot \boldsymbol{A}) &= A \boxdot \chi\boldsymbol{A}, & \chi(\boldsymbol{A} \circledcirc \boldsymbol{B}) &= \chi\boldsymbol{A} \mathbin{\boxed{\circ}} \chi\boldsymbol{B},\ \circ \in \{+, \cdot\},\\
\psi(\boldsymbol{A} \odot \boldsymbol{a}) &= \chi\boldsymbol{A} \boxdot \psi\boldsymbol{a}, & \psi(\boldsymbol{a} \oplus \boldsymbol{b}) &= \psi\boldsymbol{a} \boxplus \psi\boldsymbol{b}.
\end{aligned}$$

Here the mapping $\psi : V_nIR \to IV_nR$ and $\chi : M_nIR \to IM_nR$ are defined by (4.4.2) and (4.4.3) in Section 4.4, and $A \in IR, \boldsymbol{a}, \boldsymbol{b} \in V_nIR, \boldsymbol{A}, \boldsymbol{B} \in M_nIR$.

We observed above that the inner and outer operations $\mathbin{\Diamond\hspace{-0.62em}\circ}$ in IV_nS, which are defined by the semimorphism $\Diamond : \overline{IV_nR} \to \overline{IV_nS}$, are not executable on the computer. We now express these operations in terms of executable formulas. Using the monotone upwardly directed rounding $\Diamond : \overline{IR} \to \overline{IS}$, a rounding $\Diamond : V_nIR \to V_nIS$ (resp. $\Diamond : M_nIR \to M_nIS$) and operations in V_nIS (resp. M_nIS) can be defined by the formulas

$$\begin{aligned}
\bigwedge_{\boldsymbol{a}=(A_i)\in V_nIR} \quad & \Diamond\boldsymbol{a} := (\Diamond A_i),\\
\bigwedge_{\boldsymbol{A}=(A_{ij})\in M_nIR} \quad & \Diamond\boldsymbol{A} := (\Diamond A_{ij}),\\
\bigwedge_{\boldsymbol{a},\boldsymbol{b}\in V_nIS} \quad & \boldsymbol{a} \mathbin{\Diamond\hspace{-0.68em}+} \boldsymbol{b} := \Diamond(\boldsymbol{a} \oplus \boldsymbol{b}),\\
\bigwedge_{A\in IS, \boldsymbol{a}\in V_nIS} \quad & A \mathbin{\Diamond\hspace{-0.5em}\cdot\hspace{0.2em}} \boldsymbol{a} := \Diamond(A \odot \boldsymbol{a}),\\
\bigwedge_{A\in IS, \boldsymbol{A}\in M_nIS} \quad & A \mathbin{\Diamond\hspace{-0.5em}\cdot\hspace{0.2em}} \boldsymbol{A} := \Diamond(A \odot \boldsymbol{A}),\\
\bigwedge_{\boldsymbol{A}\in M_nIS, \boldsymbol{a}\in V_nIS} \quad & \boldsymbol{A} \mathbin{\Diamond\hspace{-0.5em}\cdot\hspace{0.2em}} \boldsymbol{a} := \Diamond(\boldsymbol{A} \odot \boldsymbol{a}).
\end{aligned}$$

As for the matrix case treated above, it can be shown that the mappings ψ and χ establish isomorphisms between the ordered algebraic structures

$$\begin{aligned}\{V_n IS, IS, \leq\} &\leftrightarrow \{IV_n S, IS, \leq\},\\ \{M_n IS, IS, \leq\} &\leftrightarrow \{IM_n S, IS, \leq\},\\ \{V_n IS, M_n IS, \leq\} &\leftrightarrow \{IV_n S, IM_n S, \leq\}.\end{aligned}$$

With $\boldsymbol{a} = (A_i)$, $\boldsymbol{b} = (B_i)$, and $\boldsymbol{A} = (A_{ij})$, we obtain the following relations for the operations defined above:

$$\begin{aligned}\boldsymbol{a} \mathbin{\Diamond\!\!\!\!+} \boldsymbol{b} &= \Diamond(\boldsymbol{a} \oplus \boldsymbol{b}) = \Diamond(A_i \boxplus B_i) = (A_i \mathbin{\Diamond\!\!\!\!+} B_i),\\ A \mathbin{\Diamond\!\!\!\!\cdot}\, \boldsymbol{a} &= \Diamond(A \odot \boldsymbol{a}) = \Diamond(A \boxdot A_i) = (A \mathbin{\Diamond\!\!\!\!\cdot}\, A_i),\\ A \mathbin{\Diamond\!\!\!\!\cdot}\, \boldsymbol{A} &= \Diamond(A \odot \boldsymbol{A}) = \Diamond(A \boxdot A_{ij}) = (A \mathbin{\Diamond\!\!\!\!\cdot}\, A_{ij}),\\ \boldsymbol{A} \mathbin{\Diamond\!\!\!\!\cdot}\, \boldsymbol{a} &= \Diamond(\boldsymbol{A} \odot \boldsymbol{a}) = \Diamond\left(\underset{\nu=1}{\overset{n}{\boxed{\sum}}} A_{i\nu} \boxdot A_\nu\right) = \left(\Diamond \underset{\nu=1}{\overset{n}{\boxed{\sum}}} A_{i\nu} \boxdot A_\nu\right).\end{aligned}$$

All these operations furnish the smallest appropriate interval, i.e., they are of optimal accuracy. Moreover, they all can be executed on a computer. The formulas also show that the operations $\boldsymbol{a} \mathbin{\Diamond\!\!\!\!+} \boldsymbol{b}$, $A \mathbin{\Diamond\!\!\!\!\cdot}\, \boldsymbol{a}$, and $A \mathbin{\Diamond\!\!\!\!\cdot}\, \boldsymbol{A}$ are identical, whether defined by the conventional method or by semimorphism. However, in the case of $\{V_n IS, M_n IS, \leq\}$ for the outer multiplication the two definitions lead to the inequality

$$\boldsymbol{A} \mathbin{\Diamond\!\!\!\!\cdot}\, \boldsymbol{a} = \left(\Diamond \underset{\nu=1}{\overset{n}{\boxed{\sum}}} A_{i\nu} \boxdot A_\nu\right) \subseteq \left(\underset{\nu=1}{\overset{n}{\Diamond\!\!\!\!\!\sum}} A_{i\nu} \mathbin{\Diamond\!\!\!\!\cdot}\, A_\nu\right). \tag{4.6.8}$$

Both sides of the last inequality are computer-executable formulas. The expression on the left hand side is of optimal accuracy. It can be executed using Table 4.1 and the exact scalar product. See Chapter 8. In practice however, calculations on computers are for simplicity often performed using the operations defined by the conventional definition of the operations, which occurs on the right-hand side of (4.6.8).

Since a semimorphism of a semimorphism is a semimorphism, these results can be used not only to describe the structure on a screen but also on one or more coarser screens as well.

Figure 4.3 shows a diagram displaying the interval vectoids discussed in this chapter. In Figure 4.3 arrows are used as they are in the diagram of Figure 4.2. Double arrows indicate isomorphisms. Horizontal arrows denote semimorphisms. Vertical arrows indicate a conventional definition of the arithmetic operations. The numbers on the arrows drawn in the figure indicate the theorems that provide the corresponding properties. The sign $\searrow\!\!\swarrow$ used in Figure 4.3 indicates the inequality (4.6.8). All interval structures, the operations of which are executable on a computer, are underlined in Figure 4.3.

$$\begin{tikzcd}
 & \{V_nR,R\} \arrow[r, "3.11"] \arrow[dl, "2.11"'] & \{V_nD,D\} \arrow[r, "3.11"] & \{V_nS,S\} \\
\{\mathbb{P}V_nR,\mathbb{P}R\} \arrow[r, "4.7"] & \{IV_nR,IR\} \arrow[r, "4.25"] \arrow[d, leftrightarrow, "4.15"] & \{IV_nD,ID\} \arrow[r, "4.25"] \arrow[d, leftrightarrow, "4.15"] & \{IV_nS,IS\} \arrow[d, leftrightarrow, "4.15"] \\
 & \{V_nIR,IR\} \arrow[r, "3.11"] & \underline{\{V_nID,ID\}} \arrow[r, "3.11"] & \underline{\{V_nIS,IS\}} \\
\mathbb{P}R \arrow[r, "4.5"] & IR \arrow[r, "4.20"] \arrow[u, "2.13"] \arrow[dd, "2.15"] & \underline{ID} \arrow[r, "4.20"] \arrow[u, "2.13"] \arrow[d, "2.15"] & \underline{IS} \arrow[u, "2.13"] \arrow[d, "2.15"] \\
 & & \underline{\{V_nID,M_nID\}} \arrow[d, phantom, "\vee"] & \underline{\{V_nIS,M_nIS\}} \arrow[d, phantom, "\vee"] \\
 & \{V_nIR,M_nIR\} \arrow[r, "3.11"] \arrow[d, leftrightarrow, "4.15"] & \underline{\{V_nID,M_nID\}} \arrow[r, "3.11"] \arrow[d, leftrightarrow, "4.15"] & \underline{\{V_nIS,M_nIS\}} \arrow[d, leftrightarrow, "4.15"] \\
\{\mathbb{P}V_nR,\mathbb{P}M_nR\} \arrow[r, "4.7"] & \{IV_nR,IM_nR\} \arrow[r, "4.25"] & \{IV_nD,IM_nD\} \arrow[r, "4.25"] & \{IV_nS,IM_nS\} \\
 & \{V_nR,M_nR\} \arrow[r, "3.11"] \arrow[ul, "2.11"] & \{V_nD,M_nD\} \arrow[r, "3.11"] & \{V_nS,M_nS\}
\end{tikzcd}$$

Figure 4.3. Interval vectoids.

4.7 Complex Interval Arithmetic

To complete our discussion of all spaces displayed in Figure 1, it remains to consider the complex interval spaces, which we now proceed to do.

Let $\{R, N, +, \cdot, /, \leq\}$ be a completely and linearly ordered division ringoid with the neutral elements o and e, and let $\mathbb{C}R := \{(x, y) \mid x, y \in R\}$ be the set of pairs over R. If in $\mathbb{C}R$ we define equality, the order relation $\leq$, and the operations $\circ \in \{+, -, \cdot, /\}$ by means of the usual formulas, we know by Theorem 2.7 that $\{\mathbb{C}R, \overline{N}, +, \cdot, /, \leq\}$ with $\overline{N} := \{\gamma \in \mathbb{C}R \mid \gamma = (c_1, c_2) \wedge c_1c_1 + c_2c_2 \in N\}$ is a completely and weakly ordered division ringoid with the neutral elements (o, o) and (e, o). Then by Theorem 2.5, the power set $\{\mathbb{P}\mathbb{C}R\backslash\{\varnothing\}, \widetilde{N}, +, \cdot, /\}$ with $\widetilde{N} := \{\Phi \in \mathbb{P}\mathbb{C}R \mid \Phi \cap \overline{N} \neq \varnothing\}$ is also a division ringoid.

Now let $I\mathbb{C}R$ denote the set of intervals over $\mathbb{C}R$ of the form

$$\Phi = [\varphi_1, \varphi_2] = \{\varphi \in \mathbb{C}R \mid \varphi_1, \varphi_2 \in \mathbb{C}R, \varphi_1 \leq \varphi \leq \varphi_2\} \text{ with } \varphi_1 \leq \varphi_2.$$

Then by Theorems 4.2 and 4.4, $\overline{I\mathbb{C}R} := I\mathbb{C}R \cup \{\varnothing\}$ is a symmetric upper screen of $\mathbb{P}\mathbb{C}R$ and the monotone upwardly directed rounding $\square : \mathbb{P}\mathbb{C}R \to \overline{I\mathbb{C}R}$ is antisymmetric. In $I\mathbb{C}R$ we define operations by employing the upwardly directed rounding

$\square$ as a semimorphism:

$$\textbf{(RG)} \qquad \bigwedge_{\Phi,\Psi\in I\mathbb{C}R} \Phi \boxcircle \Psi := \square(\Phi\circ\Psi) = \square\{\varphi\circ\psi \mid \varphi\in\Phi\wedge\psi\in\Psi\},$$
$$\text{with } \circ\in\{+,\cdot,/\},$$

where for division, we assume that $\Psi\cap\widetilde{N} = \varnothing$. For intervals $\Phi = [\varphi_1,\varphi_2]$ and $\Psi = [\psi_1,\psi_2]\in I\mathbb{C}R$, we further define an order relation

$$\Phi\le\Psi :\Leftrightarrow (\varphi_1\le\psi_1\wedge\varphi_2\le\psi_2).$$

Then by Theorem 4.5 and under its hypothesis, $\{I\mathbb{C}R, N^*, \boxplus, \boxdot, \boxslash, \le, \subseteq\}$ with $N^* := \{\Phi\in I\mathbb{C}R \mid \Phi\cap\widetilde{N}\ne\varnothing\}$ is a completely and weakly ordered division ringoid with respect to $\le$. With respect to $\subseteq$ it is an inclusion-isotonally ordered monotone upper screen division ringoid of $\mathbb{P}\mathbb{C}R$.

The operations in $I\mathbb{C}R$ are defined by those in $\mathbb{P}\mathbb{C}R$, and are therefore not executable in practice. To obtain executable formulas, we are going to study the connection between these operations and those in the set $\mathbb{C}IR$.

We start once more with the completely and linearly ordered division ringoid $\{R,N,+,\cdot,/,\le\}$. We assume additionally that $x=-e$ is already unique in R by (D5a) alone. Then by Theorem 4.5, $\{IR, N', \boxplus, \boxdot, \boxslash, \le, \subseteq\}$ is a completely ordered division ringoid. Theorems 4.5 and 4.10 prescribe explicit and performable formulas for all operations in IR. If $A=[a_1,a_2]$ and $B=[b_1,b_2]$, these formulas are summarized in Corollary 4.11 by the relation

$$\bigwedge_{A,B\in IR} A\boxcircle B := \square(A\circ B) = [\min_{i,j=1,2}(a_i\circ b_j), \max_{i,j=1,2}(a_i\circ b_j)], \quad \circ\in\{+,-,\cdot,/\}.$$

Now consider the set $\mathbb{C}IR := \{(X,Y)\mid X,Y\in IR\}$. For elements $\Phi=(X_1,X_2)$, $\Psi=(Y_1,Y_2)$ we define equality $=$, the order relation $\le$, and the arithmetic operations $\circledcirc, \circ\in\{+,\cdot,/\}$, as in Theorem 2.7 by

$$\begin{aligned}
&\Phi=\Psi :\Leftrightarrow X_1=Y_1\wedge X_2=Y_2,\\
&\Phi\le\Psi :\Leftrightarrow X_1\le Y_1\wedge X_2\le Y_2,\\
&\Phi\oplus\Psi := (X_1\boxplus Y_1, X_2\boxplus Y_2),\\
&\Phi\odot\Psi := (X_1\boxdot Y_1\boxminus X_2\boxdot Y_2, X_1\boxdot Y_2\boxplus X_2\boxdot Y_1),\\
&\Phi\oslash\Psi := \left(\frac{X_1\boxdot Y_1\boxplus X_2\boxdot Y_2}{Y_1\boxdot Y_1\boxplus Y_2\boxdot Y_2}, \frac{X_2\boxdot Y_1\boxminus X_1\boxdot Y_2}{Y_1\boxdot Y_1\boxplus Y_2\boxdot Y_2}\right) \text{ for } \Psi\in\mathbb{C}IR\setminus N'',
\end{aligned}$$

with $N'' := \{\Phi\in\mathbb{C}IR \mid \Phi=(X_1,X_2)\wedge X_1\boxdot X_1\boxplus X_2\boxdot X_2\in N'\}$. Division here means division in IR.

Thus all of these operations $\circledcirc$ are explicitly representable in terms of the operations $\boxdot$ in IR. By Theorem 2.7, $\{\mathbb{C}IR, N'', \oplus, \odot, \oslash, \leq\}$ is a completely and weakly ordered division ringoid.

We are interested in relating the operations in $I\mathbb{C}R$ and $\mathbb{C}IR$. Thus we consider the mapping $\tau : \mathbb{C}IR \to I\mathbb{C}R$ with $\tau([x_1, x_2], [y_1, y_2]) = [(x_1, y_1), (x_2, y_2)]$.

Here τ obviously is a one-to-one mapping of $\mathbb{C}IR$ onto $I\mathbb{C}R$ and an order isomorphism.

We show that τ also is a ringoid isomorphism. To do this, we prove that addition and multiplication in $I\mathbb{C}R$ and $\mathbb{C}IR$ are isomorphic. Let $\Phi = (X_1, X_2)$, $\Psi = (Y_1, Y_2) \in \mathbb{C}IR$. Then

$$\begin{aligned}
\tau\Phi \boxplus \tau\Psi &:= \square(\tau\Phi + \tau\Psi) \\
&= \left[\inf_{\varphi\in\tau\Phi, \psi\in\tau\Psi} (\varphi + \psi), \sup_{\varphi\in\tau\Phi, \psi\in\tau\Psi} (\varphi + \psi) \right] \\
&= [\inf(x_1 + y_1, x_2 + y_2), \sup(x_1 + y_1, x_2 + y_2)] \\
&= [(\inf(x_1 + y_1), \inf(x_2 + y_2)), (\sup(x_1 + y_1), \sup(x_2 + y_2))],
\end{aligned}$$

where in the last two lines displayed here, the infima and the suprema are taken over $x_i \in X_i$, $y_i \in Y_i$, $i = 1, 2$. On the other hand, we have

$$\begin{aligned}
\Phi \oplus \Psi &:= (X_1 \boxplus Y_1, X_2 \boxplus Y_2) \\
&= (\square(X_1 + Y_1), \square(X_2 + Y_2)) \\
&= ([\inf(x_1 + y_1), \sup(x_1 + y_1)], [\inf(x_2 + y_2), \sup(x_2 + y_2)]),
\end{aligned}$$

where in the last line displayed here, the infima and the suprema are taken over $x_1 \in X_1$, $y_1 \in Y_1$ or over $x_2 \in X_2$, $y_2 \in Y_2$, as the case may be. These formulas show that

$$\tau\Phi \boxplus \tau\Psi = \tau(\Phi \oplus \Psi),$$

which demonstrates the isomorphism for addition.

For multiplication we have

$$\begin{aligned}
\bigwedge_{\varphi=(x_1,x_2)\in\tau\Phi} \bigwedge_{\psi=(y_1,y_2)\in\tau\Psi} \varphi \cdot \psi &= (x_1 y_1 - x_2 y_2, x_1 y_2 + x_2 y_1) \\
&\in \tau(X_1 \boxdot Y_1 \boxminus X_2 \boxdot Y_2, X_1 \boxdot Y_2 \boxplus X_2 \boxdot Y_1) \\
&= \tau(\Phi \odot \Psi),
\end{aligned}$$

and therefore $\tau\Phi \cdot \tau\Psi \subseteq \tau(\Phi \odot \Psi)$. If we apply the monotone upwardly directed rounding $\square : \mathbb{P}\mathbb{C}R \to \overline{I\mathbb{C}R}$ to this inequality, we obtain

$$\tau\Phi \cdot \tau\Psi \underset{\text{(R3)}}{\subseteq} \square(\tau\Phi \cdot \tau\Psi) \underset{\text{(RG)}}{=} \tau\Phi \boxdot \tau\Psi \underset{\text{(R1,2)}}{\subseteq} \tau(\Phi \odot \Psi).$$

On the other hand, by employing the explicit formulas for the operations in the ringoid *IR*, we obtain with $X_i = [x_i^1, x_i^2]$, $Y_i = [y_i^1, y_i^2]$, $i = 1, 2$:

$$\begin{aligned}
X_1 \boxdot Y_1 \boxminus X_2 \boxdot Y_2 &= \left[\min_{i,j=1,2}(x_1^i y_1^j), \max_{i,j=1,2}(x_1^i y_1^j)\right] \boxminus \left[\min_{i,j=1,2}(x_2^i y_2^j), \max_{i,j=1,2}(x_2^i y_2^j)\right] \\
&= \left[\min_{i,j=1,2}(x_1^i y_1^j) - \max_{i,j=1,2}(x_2^i y_2^j), \max_{i,j=1,2}(x_1^i y_1^j) - \min_{i,j=1,2}(x_2^i y_2^j)\right], \\
X_1 \boxdot Y_2 \boxplus X_2 \boxdot Y_1 &= \left[\min_{i,j=1,2}(x_1^i y_2^j), \max_{i,j=1,2}(x_1^i y_2^j)\right] \boxplus \left[\min_{i,j=1,2}(x_2^i y_1^j), \max_{i,j=1,2}(x_2^i y_1^j)\right] \\
&= \left[\min_{i,j=1,2}(x_1^i y_2^j) + \min_{i,j=1,2}(x_2^i y_1^j), \max_{i,j=1,2}(x_1^i y_2^j) + \max_{i,j=1,2}(x_2^i y_1^j)\right],
\end{aligned}$$

i.e., the bounds of $\tau(\Phi \odot \Psi)$ consist of inner points of

$$\square\,(\tau\Phi \cdot \tau\Psi) = \tau\Phi \boxdot \tau\Psi$$

or

$$\tau(\Phi \odot \Psi) \subseteq \square\,(\tau\Phi \cdot \tau\Psi) = \tau\Phi \boxdot \tau\Psi.$$

If we take both inequalities together, we obtain

$$\tau(\Phi \odot \Psi) = \tau\Phi \boxdot \tau\Psi,$$

which demonstrates the ringoid isomorphism.

For division, we similarly obtain

$$\tau\Phi / \tau\Psi \subseteq \square\,(\tau\Phi/\tau\Psi) = \tau\Phi \boxslash \tau\Psi \subseteq \tau(\Phi \oslash \Psi).$$

Equality, however, is not provable. We show this by means of a simple example. Let $\{\mathbb{R}, \{0\}, +, \cdot, /, \leq\}$ be the division ringoid of real numbers. Then with $\Phi = (X_1, X_2) = ([0,0],[2,4])$ and $\Psi = (Y_1, Y_2) = ([1,2],[0,0])$ we obtain with $x_i \in X_i$, $y_i \in Y_i$, $i = 1, 2$,

$$\begin{aligned}
\tau\Phi \boxslash \tau\Psi &= \square\,(\tau\Phi/\tau\Psi) \\
&= \left[\left(\inf \frac{x_1 y_1 + x_2 y_2}{y_1 y_1 + y_2 y_2}, \inf \frac{x_2 y_1 - x_1 y_2}{y_1 y_1 + y_2 y_2}\right),\right. \\
&\qquad \left.\left(\sup \frac{x_1 y_1 + x_2 y_2}{y_1 y_1 + y_2 y_2}, \sup \frac{x_2 y_1 - x_1 y_2}{y_1 y_1 + y_2 y_2}\right)\right] \\
&= [(0, \frac{1}{2}), (0, 4)],
\end{aligned}$$

$$\begin{aligned}
\Phi \oslash \Psi &:= \left(\frac{X_1 \boxdot Y_1 \boxplus X_2 \boxdot Y_2}{Y_1 \boxdot Y_1 \boxplus Y_2 \boxdot Y_2}, \frac{X_2 \boxdot Y_1 \boxminus X_1 \boxdot Y_2}{Y_1 \boxdot Y_1 \boxplus Y_2 \boxdot Y_2}\right) \\
&= ([0,0] \boxslash [1,4], [2,8] \boxslash [1,4]) = ([0,0],[1/2,8]).
\end{aligned}$$

This result is not at all surprising. It is well known from interval analysis. In the complex quotient, the quantities y_1 and y_2 occur more than once. The expression $\tau(\Phi \oslash \Psi)$, therefore, only delivers an overestimate of the range of the complex function φ/ψ. In interval analysis, therefore, other formulas are occasionally used to compute the complex quotient [152].

We illustrate the ringoid isomorphism derived above in a diagram. See Figure 4.4.

$$\begin{tikzcd}
R \arrow[rr] \arrow[d] & & \mathbb{C}R \arrow[d] \\
\mathbb{P}R \arrow[d] & & \mathbb{P}\mathbb{C}R \arrow[d] \\
IR \arrow[r] & \mathbb{C}IR \arrow[r, leftrightarrow, "\tau"] & I\mathbb{C}R
\end{tikzcd}$$

Figure 4.4. Illustration of the ringoid isomorphism τ.

Since the ringoids $\mathbb{C}IR$ and $I\mathbb{C}R$ are isomorphic, we may identify corresponding elements with each other. This allows us to define an inclusion for elements $\Phi = (X_1, X_2), \Psi = (Y_1, Y_2) \in \mathbb{C}IR$ as

$$\Phi \subseteq \Psi :\Leftrightarrow X_1 \subseteq Y_1 \wedge X_2 \subseteq Y_2$$

and

$$\varphi = (x_1, x_2) \in \Phi = (X_1, X_2) :\Leftrightarrow x_1 \in X_1 \wedge x_2 \in X_2.$$

These definitions permit the interpretation that an element $\Phi = (X_1, X_2) \in \mathbb{C}IR$ represents a set of pairs of elements of $\mathbb{C}R$ by means of the identity

$$\Phi = (X_1, X_2) \equiv \{(x_1, x_2) \mid x_1 \in X_1, x_2 \in X_2\}.$$

Both sides contain the same elements.

We now consider the complex interval spaces on a screen. The following theorem describes the structure of such a screen.

Theorem 4.27. *Let $\mathbb{R}$ be the linearly ordered ringoid of real numbers with the neutral elements* 0 *and* 1*, and let $\{S, \leq\}$ be a symmetric screen of $\mathbb{R}$. Consider the complex division ringoid $\{\mathbb{C}, \{(0,0)\}, +, \cdot, /, \leq\}$ as well as the semimorphisms $\square : \mathbb{P}\mathbb{C} \to \overline{I\mathbb{C}}$ and $\Diamond : \overline{I\mathbb{C}} \to \overline{I\mathbb{C}S}$. Then $\{I\mathbb{C}S, N, \diamondplus, \diamondcdot, \diamondslash, \leq, \subseteq\}$ with $N := \{A \in I\mathbb{C}S \mid (0,0) \in A\}$ is a weakly ordered division ringoid with respect to $\leq$. The neutral elements are* $[(0,0),(0,0)]$ *and* $[(1,0),(1,0)]$*. With respect to $\subseteq$, $I\mathbb{C}S$ is an inclusion-isotonally ordered monotone upper screen division ringoid of $I\mathbb{C}$.*

Proof. We know by Theorem 3.8 that $\mathbb{C}S$ is a symmetric screen of $\mathbb{C}$. By Theorem 4.5 $\{I\mathbb{C}, \boxplus, \boxdot\}$ is a ringoid, and by Theorem 4.20 $\{I\mathbb{C}S, N, \diamondplus, \diamondcdot, \diamondslash, \leq, \subseteq\}$ is a weakly ordered as well as an inclusion-isotonally ordered division ringoid. ■

The operations $\mathbin{\Diamond\!\!\!\!\circ}$ in $I\mathbb{C}S$ are not directly implementable on a computer. Therefore, we express them in terms of implementable formulas. Consider once more the isomorphism $\tau : \mathbb{C}I\mathbb{R} \to I\mathbb{C}$. It is expressed by the formula

$$\bigwedge_{\Phi,\Psi \in \mathbb{C}I\mathbb{R}} \tau\Phi \mathbin{\boxed{\circ}} \tau\Psi = \tau(\Phi \circledcirc \Psi), \ \circ \in \{+,\cdot\}. \tag{4.7.1}$$

Using the rounding $\Diamond : \overline{I\mathbb{R}} \to \overline{IS}$ and using Theorem 3.10, a rounding $\Diamond : \mathbb{C}I\mathbb{R} \to \mathbb{C}IS$ and operations in $\mathbb{C}IS$ are defined by the semimorphism

$$\bigwedge_{\Phi=(X_1,X_2)\in\mathbb{C}I\mathbb{R}} \Diamond\Phi := (\Diamond X_1, \Diamond X_2),$$
$$\bigwedge_{\Phi,\Psi\in\mathbb{C}IS} \Phi \mathbin{\Diamond\!\!\!\!\circ} \Psi := \Diamond(\Phi \circledcirc \Psi), \ \circ \in \{+,\cdot\}.$$

Now we show in a complete analogy to the matrix case, which we considered above, that the operations in $I\mathbb{C}S$ and $\mathbb{C}IS$ are isomorphic. Denoting $(Z_1, Z_2) := \Phi \circledcirc \Psi$, we consider the expression $\tau(\Phi \mathbin{\Diamond\!\!\!\!\circ} \Psi)$, and we obtain

$$\begin{aligned}\tau(\Phi \mathbin{\Diamond\!\!\!\!\circ} \Psi) &= \tau(\Diamond(Z_1, Z_2)) = \tau(\Diamond Z_1, \Diamond Z_2)\\ &= \tau(\inf(U(Z_1) \cap IS), \inf(U(Z_2) \cap IS))\\ &= \tau(\inf(U(Z_1) \cap IS, U(Z_2) \cap IS)).\end{aligned}$$

Since the inclusion and intersection in $\mathbb{C}IS$ are defined componentwise, we may exchange the upper bound and the pair brackets to obtain

$$\tau(\Phi \mathbin{\Diamond\!\!\!\!\circ} \Psi) = \tau(\inf(U((Z_1, Z_2) \cap \mathbb{C}IS))).$$

If we now effect the mapping τ, we obtain

$$\begin{aligned}\tau(\Phi \mathbin{\Diamond\!\!\!\!\circ} \Psi) &= \inf(U(\tau(\Phi \circledcirc \Psi) \cap I\mathbb{C}S))\\ &\underset{(4.7.1)}{=} \inf(U((\tau\Phi \mathbin{\boxed{\circ}} \tau\Psi) \cap I\mathbb{C}S))\\ &\underset{\text{Theorem 4.19(c)}}{=} \Diamond(\tau\Phi \mathbin{\boxed{\circ}} \tau\Psi).\end{aligned}$$

By the definition of the operations $\mathbin{\Diamond\!\!\!\!\circ}$ in $I\mathbb{C}S$, this finally leads to

$$\tau(\Phi \mathbin{\Diamond\!\!\!\!\circ} \Psi) = \tau\Phi \mathbin{\Diamond\!\!\!\!\circ} \tau\Psi. \tag{4.7.2}$$

Thus the mapping τ represents an isomorphism between the ringoids $\{\mathbb{C}IS, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}, \leq, \subseteq\}$ defined by Theorem 3.10 and $\{I\mathbb{C}S, \mathbin{\Diamond\!\!\!\!+}, \mathbin{\Diamond\!\!\!\!\cdot}, \leq, \subseteq\}$ studied in Theorem 4.27.

The isomorphism reduces the operations in $I\mathbb{C}S$, which are not executable in practice, to the operations in $\mathbb{C}IS$ that have the properties

$$\begin{aligned}
\mathbf{\Phi} \mathbin{\Diamond\mkern-13mu{\scriptstyle +}\mkern4mu} \mathbf{\Psi} &= (X_1 \mathbin{\Diamond\mkern-13mu{\scriptstyle +}\mkern4mu} Y_1, X_2 \mathbin{\Diamond\mkern-13mu{\scriptstyle +}\mkern4mu} Y_2),\\
\mathbf{\Phi} \mathbin{\Diamond\mkern-10mu\cdot\mkern4mu} \mathbf{\Psi} &= (\Diamond(X_1 \boxdot Y_1 \boxminus X_2 \boxdot Y_2), \Diamond(X_1 \boxdot Y_2 \boxplus X_2 \boxdot Y_1)).
\end{aligned}$$

These operations are implementable on a computer. Addition can be performed componentwise in terms of addition in IS. Multiplication can be performed using Table 4.1 and the exact scalar product. See Chapter 8.

The result of these considerations concerning complex intervals is illustrated in Figure 4.5. The latter is completely analogous to Figure 4.2. We emphasize, however, that the isomorphism shown in Figure 4.5 does not include division.

$$\begin{tikzcd}
 & \mathbb{R} \arrow[r, "3.7"] \arrow[dl, "2.5"'] & D \arrow[r, "3.7"] & S \\
\mathbb{P}\mathbb{R} \arrow[r, "4.5"] & I\mathbb{R} \arrow[r, "4.20"] \arrow[dd, "2.7"] & \underline{ID} \arrow[r, "4.20"] \arrow[d, "2.7"] & \underline{IS} \arrow[d, "2.7"] \\
 & & \underline{\mathbb{C}ID} \arrow[d, phantom, "\vee"] & \underline{\mathbb{C}IS} \arrow[d, phantom, "\vee"] \\
 & \mathbb{C}I\mathbb{R} \arrow[r, "3.10"] \arrow[d, leftrightarrow, "\tau"'] & \underline{\mathbb{C}ID} \arrow[r, "3.10"] \arrow[d, leftrightarrow, "\tau"'] & \underline{\mathbb{C}IS} \arrow[d, leftrightarrow, "\tau"'] \\
\mathbb{P}\mathbb{C}\mathbb{R} \arrow[r, "4.5"] & I\mathbb{C}\mathbb{R} \arrow[r, "4.27"] & I\mathbb{C}D \arrow[r, "4.27"] & I\mathbb{C}S \\
 & \mathbb{C}\mathbb{R} \arrow[r, "3.10"] \arrow[ul, "2.5"] & \mathbb{C}D \arrow[r, "3.10"] & \mathbb{C}S
\end{tikzcd}$$

Figure 4.5. Complex interval arithmetic.

For reasons of simplicity, calculations on a computer are often performed by the conventional method of the definition of the arithmetic in the ringoids $\mathbb{C}ID$ and $\mathbb{C}IS$ (Theorem 2.7). Multiplication in these ringoids leads to upper bounds for multiplication defined in the same sets by means of Theorem 3.10. This can be seen by means of the formula

$$\begin{aligned}
\mathbf{\Phi} \mathbin{\Diamond\mkern-10mu\cdot\mkern4mu} \mathbf{\Psi} &= (\Diamond(X_1 \boxdot Y_1 \boxminus X_2 \boxdot Y_2), \Diamond(X_1 \boxdot Y_2 \boxplus X_2 \boxdot Y_1))\\
&\subseteq ((X_1 \mathbin{\Diamond\mkern-10mu\cdot\mkern4mu} Y_1 \mathbin{\Diamond\mkern-13mu{\scriptstyle -}\mkern4mu} X_2 \mathbin{\Diamond\mkern-10mu\cdot\mkern4mu} Y_2), (X_1 \mathbin{\Diamond\mkern-10mu\cdot\mkern4mu} Y_2 \mathbin{\Diamond\mkern-13mu{\scriptstyle +}\mkern4mu} X_2 \mathbin{\Diamond\mkern-10mu\cdot\mkern4mu} Y_1)).
\end{aligned}$$

The last inequality is a consequence of (R3) for the rounding $\Diamond : \overline{I\mathbb{R}} \to \overline{IS}$ and the inclusion-isotonality of addition and subtraction in IS.

4.8 Complex Interval Matrices and Interval Vectors

We now consider complex interval matrices. Again, let $\mathbb{R}$ be the linearly ordered ringoid of real numbers, $\mathbb{C}$ its complexification, and $\{I\mathbb{C}, \boxplus, \boxdot, \leq\}$ the completely and weakly ordered ringoid of intervals over $\mathbb{C}$ (Theorem 4.5). By Theorem 2.6, the set of $n \times n$-matrices $\{M_n\mathbb{C}, +, \cdot, \leq\}$ also forms a completely and weakly ordered ringoid.

We consider the ringoid $\{\mathbb{P}M_n\mathbb{C}, +, \cdot\}$ corresponding to the power set of $M_n\mathbb{C}$ (Theorem 2.5). In the subset of intervals $IM_n\mathbb{C} \subset \mathbb{P}M_n\mathbb{C}$ we define operations $\boxed{\circ}$, $\circ \in \{+, \cdot\}$, by employing the semimorphism

$$\square : \mathbb{P}M_n\mathbb{C} \to \overline{IM_n\mathbb{C}} := IM_n\mathbb{C} \cup \{\varnothing\},$$

$$\bigwedge_{\boldsymbol{A},\boldsymbol{B} \in IM_n\mathbb{C}} \boldsymbol{A} \boxed{\circ} \boldsymbol{B} := \square(\boldsymbol{A} \circ \boldsymbol{B}) = [\inf(\boldsymbol{A} \circ \boldsymbol{B}), \sup(\boldsymbol{A} \circ \boldsymbol{B})].$$

Then using Theorem 4.5, we see that $\{IM_n\mathbb{C}, \boxplus, \boxdot, \leq\}$ is also a completely and weakly ordered ringoid.

Independently of the set $IM_n\mathbb{C}$, we now consider the set $M_nI\mathbb{C}$, i.e., the set of $n \times n$-matrices the components of which are intervals over $\mathbb{C}$. Employing the operations in the ringoid $\{I\mathbb{C}, \boxplus, \boxdot, \leq, \subseteq\}$, we define operations and an order relation in $M_nI\mathbb{C}$ by means of Theorem 2.6. Then $\{M_nI\mathbb{C}, \oplus, \odot, \leq\}$ is seen to be a completely and weakly ordered ringoid.

We consider the question of whether there exists any relation between the completely and weakly ordered ringoids $\{M_nI\mathbb{C}, \oplus, \odot, \leq\}$ and $\{IM_n\mathbb{C}, \boxplus, \boxdot, \leq\}$. The assertion of Lemma 4.13 is that both of these ringoids are isomorphic with respect to the algebraic and the order structure if the following formula relating the operations in $\mathbb{P}\mathbb{C}$ and $I\mathbb{C}$ holds:

$$\bigwedge_{A_\nu, B_\nu \in I\mathbb{C}} \left(\sum_{\nu=1}^{n} A_\nu \boxdot B_\nu \subseteq \square \left(\sum_{\nu=1}^{n} A_\nu \cdot B_\nu \right) \right). \tag{4.8.1}$$

Here $\square : \mathbb{P}\mathbb{C} \to \overline{I\mathbb{C}} := I\mathbb{C} \cup \{\varnothing\}$ denotes the monotone upwardly directed rounding. We are now going to show that this formula is valid in the present setting.

Because of the isomorphism between the ringoids $\mathbb{C}I\mathbb{R}$ and $I\mathbb{C}$, which we noted above (see Figure 4.5), we simply identify certain elements of $\mathbb{C}I\mathbb{R}$ and $I\mathbb{C}$ within the following proof. These elements are the ones that correspond to each other through the mapping

$$\tau : \mathbb{C}I\mathbb{R} \to I\mathbb{C} \text{ with } \tau([x_1, x_2], [y_1, y_2]) = [(x_1, y_1), (x_2, y_2)].$$

Let $A_i, B_i \in I\mathbb{C}$. Using the isomorphism just referred to, $A_i \equiv (A_{i1}, A_{i2})$ and $B_i \equiv (B_{i1}, B_{i2})$, where $A_{i1}, A_{i2}, B_{i1}, B_{i2} \in I\mathbb{R}$. Then by the definition of the addition in $\mathbb{P}\mathbb{C}$ we know that every $x \in \sum_{i=1}^{n} A_i \boxdot B_i$ is of the form $x = \sum_{i=1}^{n} x_i$ with

$x_i \in A_i \boxdot B_i$ and

$$A_i \boxdot B_i \equiv (A_{i1} \boxdot B_{i1} \boxminus A_{i2} \boxdot B_{i2}, A_{i1} \boxdot B_{i2} \boxplus A_{i2} \boxdot B_{i1}).$$

If we now also separate $x_i = (x_{i1}, x_{i2})$ into real and imaginary parts, we obtain

$$\bigwedge_{i=1(1)n} (x_{i1} \in A_{i1} \boxdot B_{i1} \boxminus A_{i2} \boxdot B_{i2} \quad \wedge \quad x_{i2} \in A_{i1} \boxdot B_{i2} \boxplus A_{i2} \boxdot B_{i1})$$

and

$$x = \left(\sum_{i=1}^{n} x_{i1}, \sum_{i=1}^{n} x_{i2} \right).$$

Because of Corollary 4.11, we now get with $A_{ij} = [a_{ij}^1, a_{ij}^2]$ and $B_{ij} = [b_{ij}^1, b_{ij}^2]$, $i = 1(1)n$, $j = 1, 2$, and using the following abbreviations $\alpha_i, \beta_i, \gamma_i, \delta_i$ for all $i = 1(1)n$, that

$$\begin{aligned} \alpha_i := \min_{r,s=1,2}(a_{i1}^r \cdot b_{i1}^s) - \max_{r,s=1,2}(a_{i2}^r \cdot b_{i2}^s) \\ \leq x_{i1} \leq \\ \max_{r,s=1,2}(a_{i1}^r \cdot b_{i1}^s) - \min_{r,s=1,2}(a_{i2}^r \cdot b_{i2}^s) =: \beta_i, \end{aligned}$$

$$\begin{aligned} \gamma_i := \min_{r,s=1,2}(a_{i1}^r \cdot b_{i2}^s) + \min_{r,s=1,2}(a_{i2}^r \cdot b_{i1}^s) \\ \leq x_{i2} \leq \\ \max_{r,s=1,2}(a_{i1}^r \cdot b_{i2}^s) + \max_{r,s=1,2}(a_{i2}^r \cdot b_{i1}^s) =: \delta_i \end{aligned}$$

$$\underset{(\mathrm{OD1})_{\mathbb{R}}}{\Rightarrow} \quad \sum_{i=1}^{n} \alpha_i \leq \sum_{i=1}^{n} x_{i1} \leq \sum_{i=1}^{n} \beta_i \quad \wedge \quad \sum_{i=1}^{n} \gamma_i \leq \sum_{i=1}^{n} x_{i2} \leq \sum_{i=1}^{n} \delta_i,$$

and

$$x = \left(\sum_{i=1}^{n} x_{i1}, \sum_{i=1}^{n} x_{i2} \right),$$

i.e., the bounds that we have just established for x consist of points in the set represented by the sum appearing in the expression

$$\square \left(\sum_{i=1}^{n} A_i \cdot B_i \right).$$

This proves (4.8.1).

Then by Lemma 4.13, the two completely and weakly ordered ringoids $\{IM_n\mathbb{C}, \boxplus, \boxdot, \leq\}$ and $\{M_nI\mathbb{C}, \oplus, \odot, \leq\}$ are isomorphic with respect to the algebraic and the order structure. Figure 4.6 illustrates this isomorphism by means of a scheme. By this isomorphism the operations $\boxdot$, $\circ \in \{+, -, \cdot\}$, in $IM_n\mathbb{C}$, which are not implementable in terms of their definition, are reduced to the operations in $M_nI\mathbb{C}$. Because of the isomorphism $\tau : I\mathbb{C} \to \mathbb{C}I\mathbb{R}$, these latter operations are identical to those in the ringoid $M_n\mathbb{C}I\mathbb{R}$. These operations in turn are defined and can easily be implemented in terms of those in $\mathbb{C}I\mathbb{R}$.

$$\begin{tikzcd}
\mathbb{C} \arrow[rr] \arrow[d] & & M_n\mathbb{C} \arrow[d] \\
\mathbb{P}\mathbb{C} \arrow[d] & & \mathbb{P}M_n\mathbb{C} \arrow[d] \\
I\mathbb{C} \arrow[r] & M_nI\mathbb{C} \arrow[r, leftrightarrow] & IM_n\mathbb{C}
\end{tikzcd}$$

Figure 4.6. Illustration of the isomorphism $M_nI\mathbb{C} \leftrightarrow IM_n\mathbb{C}$.

We now consider complex interval matrices on a screen, and we begin with the following theorem that characterizes such a structure.

Theorem 4.28. *Let $\mathbb{R}$ be the linearly ordered ringoid of real numbers and let $\{S, \leq\}$ be a symmetric screen of $\mathbb{R}$. Consider the complex ringoids $\{\mathbb{C}, +, \cdot, \leq\}$ and $\{M_n\mathbb{C}, +, \cdot, \leq\}$ and the semimorphisms $\square : \mathbb{P}M_n\mathbb{C} \to \overline{IM_n\mathbb{C}}$ and $\Diamond : \overline{IM_n\mathbb{C}} \to \overline{IM_n\mathbb{C}S}$. Then $\{IM_n\mathbb{C}S, \diamondplus, \diamonddot, \leq, \subseteq\}$ is a completely and weakly ordered ringoid with respect to $\leq$. With respect to $\subseteq$, $IM_n\mathbb{C}S$ is an inclusion-isotonally ordered monotone upper screen ringoid of $IM_n\mathbb{C}$.*

Proof. The proof of this theorem is completely analogous to the proofs of Theorems 4.23 and 4.27. ■

Now in a manner similar to the discussion following Theorems 4.23 and 4.27, the usual isomorphisms can be studied. Figure 4.7 gives the resulting diagram.

The case of complex vectoids can be presented as the following corollary of Theorem 4.25.

Corollary 4.29. *Let $\mathbb{R}$ be the linearly ordered ringoid of real numbers and $\{S, \leq\}$ a symmetric screen of $\mathbb{R}$. Consider the complex ringoids $\{\mathbb{C}, +, \cdot, \leq\}$, $\{M_n\mathbb{C}, +, \cdot, \leq\}$ and the weakly ordered vectoids $\{V_n\mathbb{C}, \mathbb{C}, \leq\}$, $\{M_n\mathbb{C}, \mathbb{C}, \leq\}$, and $\{V_n\mathbb{C}, M_n\mathbb{C}, \leq\}$ as well as the semimorphisms $\square : \mathbb{P}V_n\mathbb{C} \to \overline{IV_n\mathbb{C}}$, $\Diamond : \overline{IV_n\mathbb{C}} \to \overline{IV_n\mathbb{C}S}$, and $\square : \mathbb{P}M_n\mathbb{C} \to \overline{IM_n\mathbb{C}}$, $\Diamond : \overline{IM_n\mathbb{C}} \to \overline{IM_n\mathbb{C}S}$. Then the structures $\{IV_n\mathbb{C}S, I\mathbb{C}S, \leq, \subseteq\}$, $\{IM_n\mathbb{C}S, I\mathbb{C}S, \leq, \subseteq\}$, and $\{IV_n\mathbb{C}S, IM_n\mathbb{C}S, \leq, \subseteq\}$ are completely and weakly ordered vectoids with respect to $\leq$. $\{IM_n\mathbb{C}S, I\mathbb{C}S\}$ is multiplicative. With respect to $\subseteq$, all of these vectoids are inclusion-isotonally ordered and monotone*

$$\begin{tikzcd}
 & \mathbb{C} \arrow[dl, "2.5"'] \arrow[rr, "3.10"] & & \mathbb{C}D \arrow[r, "3.10"] & \mathbb{C}S \\
\mathbb{P}\mathbb{C} \arrow[r, "4.5"] & I\mathbb{C} \arrow[rr, "4.27"] \arrow[dr, leftrightarrow, "\tau"] \arrow[dddd] & & I\mathbb{C}D \arrow[r, "4.27"] \arrow[d, leftrightarrow, "\tau"] & I\mathbb{C}S \arrow[d, leftrightarrow, "\tau"] \\
 & & \mathbb{C}IR \arrow[r, "3.10"] \arrow[dd, "2.6"] & \underline{\mathbb{C}ID} \arrow[r, "3.10"] \arrow[d, "2.6"] & \underline{\mathbb{C}IS} \arrow[d, "2.6"] \\
 & & & \underline{M_n\mathbb{C}ID} \arrow[d, phantom, "\vee"] & \underline{M_n\mathbb{C}IS} \arrow[d, phantom, "\vee"] \\
 & & M_n\mathbb{C}IR \arrow[r, "3.9"] \arrow[dl, leftrightarrow] & \underline{M_n\mathbb{C}ID} \arrow[r, "3.9"] \arrow[d, leftrightarrow] & \underline{M_n\mathbb{C}IS} \arrow[d, leftrightarrow] \\
 & M_nI\mathbb{C} \arrow[rr, "3.9"] \arrow[d, leftrightarrow] & & M_nI\mathbb{C}D \arrow[r, "3.9"] \arrow[d, leftrightarrow] & M_nI\mathbb{C}S \arrow[d, leftrightarrow] \\
\mathbb{P}M_n\mathbb{C} \arrow[r, "4.5"] & IM_n\mathbb{C} \arrow[rr, "4.28"] & & IM_n\mathbb{C}D \arrow[r, "4.28"] & IM_n\mathbb{C}S \\
 & M_n\mathbb{C} \arrow[ul, "2.5"] \arrow[rr, "3.9"] & & M_n\mathbb{C}D \arrow[r, "3.9"] & M_n\mathbb{C}S
\end{tikzcd}$$

Figure 4.7. Complex interval matrices.

upper screen vectoids of $\{IV_n\mathbb{C}, I\mathbb{C}\}$, $\{IM_n\mathbb{C}, I\mathbb{C}\}$, *and* $\{IV_n\mathbb{C}, IM_n\mathbb{C}\}$, *respectively.* ■

The operations in the interval vectoids occurring in this corollary are not implementable by using their definition. Under our assumptions, however, we may apply the isomorphisms expressed by Theorem 4.16, and in addition, we make use of the isomorphisms $I\mathbb{C} \leftrightarrow \mathbb{C}I\mathbb{R}$ and $IM_n\mathbb{C} \leftrightarrow M_n\mathbb{C}I\mathbb{R}$. Summarizing, we obtain the following isomorphisms

$$\begin{aligned}
\{IV_n\mathbb{C}, I\mathbb{C}, \leq, \subseteq\} &\leftrightarrow \{V_nI\mathbb{C}, I\mathbb{C}, \leq, \subseteq\} \\
&\leftrightarrow \{V_n\mathbb{C}I\mathbb{R}, \mathbb{C}I\mathbb{R}, \leq, \subseteq\}, \\
\{IM_n\mathbb{C}, I\mathbb{C}, \leq, \subseteq\} &\leftrightarrow \{M_nI\mathbb{C}, I\mathbb{C}, \leq, \subseteq\} \\
&\leftrightarrow \{M_n\mathbb{C}I\mathbb{R}, \mathbb{C}I\mathbb{R}, \leq, \subseteq\}, \\
\{IV_n\mathbb{C}, IM_n\mathbb{C}, \leq, \subseteq\} &\leftrightarrow \{V_nI\mathbb{C}, M_nI\mathbb{C}, \leq, \subseteq\} \\
&\leftrightarrow \{V_n\mathbb{C}I\mathbb{R}, M_n\mathbb{C}I\mathbb{R}, \leq, \subseteq\}.
\end{aligned}$$

Now it can be shown in complete analogy to the considerations following Theorems 4.23 and 4.27 that these isomorphisms also induce isomorphisms between the corresponding spaces over D and S, for instance

$$\begin{aligned}
\{IV_n\mathbb{C}S, I\mathbb{C}S, \leq, \subseteq\} &\leftrightarrow \{V_nI\mathbb{C}S, I\mathbb{C}S, \leq, \subseteq\}\\
&\leftrightarrow \{V_n\mathbb{C}IS, \mathbb{C}IS, \leq, \subseteq\},\\
\{IM_n\mathbb{C}S, I\mathbb{C}S, \leq, \subseteq\} &\leftrightarrow \{M_nI\mathbb{C}S, I\mathbb{C}S, \leq, \subseteq\}\\
&\leftrightarrow \{M_n\mathbb{C}IS, \mathbb{C}IS, \leq, \subseteq\},\\
\{IV_n\mathbb{C}S, IM_n\mathbb{C}S, \leq, \subseteq\} &\leftrightarrow \{V_nI\mathbb{C}S, M_nI\mathbb{C}S, \leq, \subseteq\}\\
&\leftrightarrow \{V_n\mathbb{C}IS, M_n\mathbb{C}IS, \leq, \subseteq\}.
\end{aligned}$$

Figure 4.8 summarizes all these results. As in the similar figures displayed before, double arrows indicate isomorphisms, horizontal arrows denote semimorphisms, and vertical arrows indicate a conventional definition of the arithmetic operations. All operations defined by semimorphisms within vectoids ending with $\mathbb{C}ID$ or $\mathbb{C}IS$ can be implemented on a computer by the exact scalar product (Chapter 8). Upper bounds for these operations can be obtained if within the same sets operations are also defined by the conventional method (Figure 4.8).

$$\begin{tikzcd}
& \{V_n\mathbb{C},\mathbb{C}\} \arrow[rr,"3.7"] \arrow[dl,"2.7"'] & & \{V_n\mathbb{C}D,\mathbb{C}D\} \arrow[r,"3.7"] & \{V_n\mathbb{C}S,\mathbb{C}S\} \\
\{\mathbb{P}V_n\mathbb{C},\mathbb{P}\mathbb{C}\} \arrow[r,"4.5"] & \{IV_n\mathbb{C},I\mathbb{C}\} \arrow[rr,"4.19"] \arrow[d,leftrightarrow,"4.8"] & & \{IV_n\mathbb{C}D,I\mathbb{C}D\} \arrow[r,"4.19"] \arrow[d,leftrightarrow] & \{IV_n\mathbb{C}S,I\mathbb{C}S\} \arrow[d,leftrightarrow] \\
& \{V_nI\mathbb{C},I\mathbb{C}\} \arrow[rr] \arrow[dr,leftrightarrow] & & \{V_nI\mathbb{C}D,I\mathbb{C}D\} \arrow[r] \arrow[d,leftrightarrow] & \{V_nI\mathbb{C}S,I\mathbb{C}S\} \arrow[d,leftrightarrow] \\
& & \{V_n\mathbb{C}IR,\mathbb{C}IR\} \arrow[r,"3.7"] & \begin{array}{c}\underline{\{V_n\mathbb{C}ID,\mathbb{C}ID\}}\\ \wedge\\ \underline{\{V_n\mathbb{C}ID,\mathbb{C}ID\}}\end{array} \arrow[r,"3.7"] & \begin{array}{c}\underline{\{V_n\mathbb{C}IS,\mathbb{C}IS\}}\\ \wedge\\ \underline{\{V_n\mathbb{C}IS,\mathbb{C}IS\}}\end{array} \\
\mathbb{P}\mathbb{C} \arrow[r,"4.3"] & I\mathbb{C} \arrow[r] \arrow[uu,"2.9"] \arrow[dd,"2.11"] & \mathbb{C}I \arrow[r,"3.6"] \arrow[u,"2.9"'] \arrow[d,"2.11"] & \underline{\mathbb{C}ID} \arrow[r,"3.6"] \arrow[u,"2.9"'] \arrow[d,"2.11"'] & \underline{\mathbb{C}IS} \arrow[u,"2.9"'] \arrow[d,"2.11"'] \\
& & \{V_n\mathbb{C}IR,M_n\mathbb{C}IR\} \arrow[r,"3.7"] \arrow[dl,leftrightarrow] & \begin{array}{c}\underline{\{V_n\mathbb{C}ID,M_n\mathbb{C}ID\}}\\ \vee\\ \underline{\{V_n\mathbb{C}ID,M_n\mathbb{C}ID\}}\end{array} \arrow[r,"3.7"] \arrow[d,leftrightarrow] & \begin{array}{c}\underline{\{V_n\mathbb{C}IS,M_n\mathbb{C}IS\}}\\ \vee\\ \underline{\{V_n\mathbb{C}IS,M_n\mathbb{C}IS\}}\end{array} \arrow[d,leftrightarrow] \\
& \{V_nI\mathbb{C},M_nI\mathbb{C}\} \arrow[rr] \arrow[d,leftrightarrow,"4.9"] & & \underline{\{V_nI\mathbb{C}D,M_nI\mathbb{C}D\}} \arrow[r] \arrow[d,leftrightarrow] & \underline{\{V_nI\mathbb{C}S,M_nI\mathbb{C}S\}} \arrow[d,leftrightarrow] \\
\{\mathbb{P}V_n\mathbb{C},\mathbb{P}M_n\mathbb{C}\} \arrow[r,"4.5"] & \{IV_n\mathbb{C},IM_n\mathbb{C}\} \arrow[rr,"4.19"] & & \{IV_n\mathbb{C}D,IM_n\mathbb{C}D\} \arrow[r,"4.19"] & \{IV_n\mathbb{C}S,IM_n\mathbb{C}S\} \\
& \{V_n\mathbb{C},M_n\mathbb{C}\} \arrow[rr,"3.7"] \arrow[ul,"2.7"] & & \{V_n\mathbb{C}D,M_n\mathbb{C}D\} \arrow[r,"3.7"] & \{V_n\mathbb{C}S,M_n\mathbb{C}S\}
\end{tikzcd}$$

Figure 4.8. Complex interval vectoids.

4.9 Extended Interval Arithmetic

In an ordered or weakly ordered division ringoid R interval operations have been defined by (RG) (compare Section 4.1):

$$\textbf{(RG)} \quad \bigwedge_{A,B\in IR} A \boxed{\circ} B := \square(A\circ B) = [\inf\nolimits_{a\in A,b\in B}(a\circ b), \sup\nolimits_{a\in A,b\in B}(a\circ b)].$$

Here $\square : \mathbb{P}R \to \overline{IR}$ is the monotone upwardly directed rounding from the power set into the subset of intervals IR. In the case of division it was explicitly assumed that $0 \notin B$. Under this assumption the result is always an interval of IR. In particular, for real numbers $\mathbb{R}$ the rounding has no effect. Remark 4.12 states that for intervals $A, B \in I\mathbb{R}$ the result of the power set operation $A \circ B$ is already an interval of $I\mathbb{R}$.

We now retract the assumption $0 \notin B$ for division. This is motivated by fundamental applications of interval mathematics. Zero finding is a central task of mathematics. In conventional numerical analysis Newton's method is the key method for computing zeros of nonlinear functions. For a brief sketch of the method see Chapter 9. It is well known that under certain natural assumptions on the function the method converges quadratically to the solution if the initial value of the iteration is already close enough to the zero. However, Newton's method may well fail to find the solution in finite as well as in infinite precision arithmetic. This may happen, for instance, if the initial value of the iteration is not close enough to the solution.

In contrast to this the interval version of Newton's method never fails, not even in rounded arithmetic. It is globally convergent.

It is one of the main achievements of interval mathematics that the interval version of Newton's method has been extended so that it can be used to enclose all (single) zeros of a function in a given domain. Newton's method reaches its final elegance and strength in the form of the *extended interval Newton method* (Chapter 9). The method is locally quadratically convergent and it never fails, not even in rounded arithmetic.

The key operation to achieve these fascinating properties of Newton's method is division by an interval which contains zero. We will now develop this operation, which the extended interval Newton method uses to separate different solutions.

The power set $\mathbb{P}\mathbb{R}$ over the real numbers $\mathbb{R}$ is defined as the set of all subsets of real numbers. With respect to set inclusion as an order relation $\{\mathbb{P}\mathbb{R}, \subseteq\}$ is an ordered set and a complete lattice. The least element of $\{\mathbb{P}\mathbb{R}, \subseteq\}$ is the empty set. The greatest element of $\{\mathbb{P}\mathbb{R}, \subseteq\}$ is the set $\mathbb{R}$. The infimum of two elements of $\mathbb{P}\mathbb{R}$ is the intersection and the supremum is the union.

The operations of the real numbers $\mathbb{R}$ have been extended to the power set $\mathbb{P}\mathbb{R}$ by

$$\bigwedge_{A,B\in\mathbb{P}\mathbb{R}} A\circ B := \{a\circ b \mid a\in A \wedge b\in B\}, \quad \text{for all } \circ\in\{+,-,\cdot,/\}. \tag{4.9.1}$$

The following properties are obvious and immediate consequences of this definition:

$$A \subseteq B \wedge C \subseteq D \Rightarrow A \circ C \subseteq B \circ D, \quad \text{for all } A, B, C, D \in \mathbb{PR}, \tag{4.9.2}$$

and in particular

$$a \in A \wedge b \in B \Rightarrow a \circ b \in A \circ B, \quad \text{for all } A, B \in \mathbb{PR}. \tag{4.9.3}$$

Property (4.9.2) is called *inclusion-isotony* (or *inclusion-monotonicity*). Property (4.9.3) is called the *inclusion property*.

By use of parentheses these rules can immediately be extended to expressions with more than one arithmetic operation, e.g.,

$$A \subseteq B \wedge C \subseteq D \wedge E \subseteq F \Rightarrow A \circ C \subseteq B \circ D \Rightarrow (A \circ C) \circ E \subseteq (B \circ D) \circ F,$$

and so on. Moreover, if more than one operation is defined then this chain of conclusions also remains valid for expressions containing several different operations. In particular all these properties, i.e., the inclusion-monotonicity and inclusion properties, also hold if in (4.9.1) $\mathbb{PR}$ is restricted to operands of $I\mathbb{R}$.

The set $I\mathbb{R}$ of closed and bounded intervals over $\mathbb{R}$ is a particular subset of $\mathbb{PR}$. With respect to set inclusion as an order relation $I\mathbb{R}$ again is an ordered set. Lattice operations are also available in the set $\overline{I\mathbb{R}} := I\mathbb{R} \cup \{\varnothing\}$. The infimum of two elements of $I\mathbb{R}$ is the intersection and the supremum is the interval (convex) hull. In $I\mathbb{R}$ arithmetic operations are defined by (4.9.1). If division by an interval that includes zero is excluded the result is again an element of $I\mathbb{R}$. It can be computed using simple formulas. Under the assumption $0 \notin B$ for division, the intervals of $I\mathbb{R}$ are an algebraically closed subset[4] of the power set $\mathbb{PR}$, i.e., an operation for intervals of $I\mathbb{R}$ performed in $\mathbb{PR}$ always delivers an interval of $I\mathbb{R}$.

In accordance with (4.9.1) division in $I\mathbb{R}$ is defined by

$$\bigwedge_{A,B \in I\mathbb{R}} A/B := \{a/b \mid a \in A \wedge b \in B\}. \tag{4.9.4}$$

The quotient a/b is defined as the inverse operation of multiplication, i.e., as the solution of the equation $b \cdot x = a$. Thus (4.9.4) can be written in the form

$$\bigwedge_{A,B \in I\mathbb{R}} A/B := \{x \mid bx = a \wedge a \in A \wedge b \in B\}. \tag{4.9.5}$$

For $0 \notin B$ (4.9.4) and (4.9.5) are equivalent. While in $\mathbb{R}$ division by zero is not defined the representation of A/B by (4.9.5) allows definition of the operation and interpretation of the result for $0 \in B$ also.

[4] as the integers are of the real numbers

By way of interpreting (4.9.5) for $A = [a_1, a_2]$ and $B = [b_1, b_2] \in I\mathbb{R}$ with $0 \in B$ the following eight distinct cases can be set out:

case	$A = [a_1, a_2]$	$B = [b_1, b_2]$
1	$0 \in A,$	$0 \in B.$
2	$0 \notin A,$	$B = [0, 0].$
3	$a_1 \leq a_2 < 0,$	$b_1 < b_2 = 0.$
4	$a_1 \leq a_2 < 0,$	$b_1 < 0 < b_2.$
5	$a_1 \leq a_2 < 0,$	$0 = b_1 < b_2.$
6	$0 < a_1 \leq a_2,$	$b_1 < b_2 = 0.$
7	$0 < a_1 \leq a_2,$	$b_1 < 0 < b_2.$
8	$0 < a_1 \leq a_2,$	$0 = b_1 < b_2.$

Table 4.7. The eight cases of interval division A/B, with $A, B \in I\mathbb{R}$, and $0 \in B$.

The list distinguishes the cases $0 \in A$ (case 1) and $0 \notin A$ (cases 2 to 8). Since it is generally assumed that $0 \in B$, these eight cases indeed cover all possibilities.

We now derive simple formulas for the result of the interval division A/B for these eight cases:

(a) $0 \in A \wedge 0 \in B$. Since every $x \in \mathbb{R}$ fulfils the equation $0 \cdot x = 0$, we have $A/B = (-\infty, +\infty)$. Here $(-\infty, +\infty)$ denotes the open interval between $-\infty$ and $+\infty$ which just consists of all real numbers $\mathbb{R}$, i.e., $A/B = \mathbb{R}$.

(b) In the case $0 \notin A \wedge B = [0, 0]$ the set defined by (4.9.5) consists of all elements which fulfil the equation $0 \cdot x = a$ for $a \in A$. Since $0 \notin A$, there is no real number which fulfils this equation. Thus A/B is the empty set, i.e., $A/B = \emptyset$.

case	$A = [a_1, a_2]$	$B = [b_1, b_2]$	B'	A/B'	A/B
1	$0 \in A$	$0 \in B$			$(-\infty, +\infty)$
2	$0 \notin A$	$B = [0, 0]$			$\emptyset$
3	$a_2 < 0$	$b_1 < b_2 = 0$	$[b_1, (-\epsilon)]$	$[a_2/b_1, a_1/(-\epsilon)]$	$[a_2/b_1, +\infty)$
4	$a_2 < 0$	$b_1 < 0 < b_2$	$[b_1, (-\epsilon)]$	$[a_2/b_1, a_1/(-\epsilon)]$	$(-\infty, a_2/b_2]$
			$\cup\, [\epsilon, b_2]$	$\cup\, [a_1/\epsilon, a_2/b_2]$	$\cup\, [a_2/b_1, +\infty)$
5	$a_2 < 0$	$0 = b_1 < b_2$	$[\epsilon, b_2]$	$[a_1/\epsilon, a_2/b_2]$	$(-\infty, a_2/b_2]$
6	$a_1 > 0$	$b_1 < b_2 = 0$	$[b_1, (-\epsilon)]$	$[a_2/(-\epsilon), a_1/b_1]$	$(-\infty, a_1/b_1]$
7	$a_1 > 0$	$b_1 < 0 < b_2$	$[b_1, (-\epsilon)]$	$[a_2/(-\epsilon), a_1/b_1]$	$(-\infty, a_1/b_1]$
			$\cup\, [\epsilon, b_2]$	$\cup\, [a_1/b_2, a_2/\epsilon]$	$\cup\, [a_1/b_2, +\infty)$
8	$a_1 > 0$	$0 = b_1 < b_2$	$[\epsilon, b_2]$	$[a_1/b_2, a_2/\epsilon]$	$[a_1/b_2, +\infty)$

Table 4.8. The eight cases of interval division A/B, with $A, B \in I\mathbb{R}$, and $0 \in B$.

In all other cases $0 \notin A$ also. We have already observed under (b) that the element 0 in B does not contribute to the solution set. So it can be excluded without changing the set A/B.

So the general rule for computing the set A/B by (4.9.5) is to remove its zero from the interval B and replace it by a small positive or negative number ϵ as the case may be. The resulting set is denoted by B' and represented in column 4 of Table 4.8. With this B' the solution set A/B' can now easily be computed by applying the rules of Table 4.2. The results are shown in column 5 of Table 4.8. Now the desired result A/B as defined by (4.9.5) is obtained if in column 5 ϵ tends to zero.

Thus in the cases 3 to 8 the results are obtained by the limit process $A/B = \lim_{\epsilon \to 0} A/B'$. The solution set A/B is shown in the last column of Table 4.8 for all the eight cases. There, as usual in mathematics round brackets indicate that the bound is not included in the set. In contrast to this square brackets denote closed interval ends, i.e., the bound is included.

The operands A and B of the division A/B in Table 4.8 are intervals of $I\mathbb{R}$. The results of the division shown in the last column, however, are no longer intervals of $I\mathbb{R}$. The result is now an element of the power set $\mathbb{PR}$. With the exception of case 2 the result is now a set which stretches continuously to $-\infty$ or $+\infty$ or both.

In two cases (rows 4 and 7 in Table 4.8) the result consists of the union of two distinct sets of the form $(-\infty, c_2] \cup [c_1, +\infty)$. These cases can easily be identified by the signs of the bounds of the divisor. Within the given framework of existing processors only one interval can be delivered as the result of an interval operation. In the cases 4 and 7 of Table 4.8 the result, yet, can be returned as an improper interval $[c_1, c_2]$ where the left hand bound is higher than the right hand bound. Motivated by the extended interval Newton method[5] it is reasonable to separate these results into the two distinct sets: $(-\infty, c_2]$ and $[c_1, +\infty)$. The fact that an arithmetic operation delivers two distinct results seems to be a totally new situation in computing. Evaluation of the square root, however, also delivers two results and we have learned to live with it. Computing certainly is able to deal with this situation.

A principle solution of the problem would be for the computer to provide a flag for *distinct intervals*. The situation occurs if the divisor is an interval that contains zero as an interior point. In cases 4 and 7 of Table 4.8 the flag would be raised and signaled to the user. The user may then apply a routine of his choice to deal with the situation as is appropriate for his application.[6]

[5] Newton's method reaches its ultimate elegance and strength in the extended interval Newton method. If division by an interval that contains zero delivers two distinct sets the computation is continued along two separate paths, one for each interval. This is how the extended interval Newton method separates different zeros from each other and finally computes all zeros in a given domain. If the interval Newton method delivers the empty set, the method has proved that there is no zero in the initial interval.

[6] This routine could be: modify the operands and recompute, or continue the computation with one of the sets and ignore the other one, or put one of the sets on a list and continue the computation with the other one, or return the entire set of real numbers $(-\infty, +\infty)$ as result and continue the computation, or

If during a computation in the real number field zero appears as a divisor the computation should be stopped immediately. In floating-point arithmetic the situation is different. Zero may be the result of an underflow. In such a case a corresponding interval computation would not deliver zero but a small interval with zero as one bound and a tiny positive or negative number as the other bound. In this case division is well defined by Table 4.8. The result is a closed interval which stretches continuously to $-\infty$ or $+\infty$ as the case may be. In the real number field zero as a divisor is an accident. So in interval arithmetic division by an interval that contains zero as an interior point certainly will be a rare appearance. An exception is the interval Newton method. Here, however, it is clear how the situation has to be handled. See Chapter 9 of this book.

In the literature an improper interval $[c_1, c_2]$ with $c_1 > c_2$ occasionally is called an 'exterior interval'. On the number circle an 'exterior interval' is interpreted as an interval with infinity as an interior point. We do not follow this line here. Interval arithmetic is defined as an arithmetic for sets of real numbers. Operations for real numbers which deliver ∞ (or $-\infty$ or $+\infty$) as their result do not exist. Here and in the following the symbols $-\infty$ and $+\infty$ are only used to describe sets of real numbers.

After the splitting of improper intervals into two distinct sets only four kinds of result come from division by an interval of $I\mathbb{R}$ which contains zero:

$$\emptyset, \quad (-\infty, a], \quad [b, +\infty), \quad \text{and} \quad (-\infty, +\infty). \tag{4.9.6}$$

We call such elements extended intervals. The union of the set of intervals of $I\mathbb{R}$ with the set of extended intervals is denoted by $(I\mathbb{R})$. The elements of the set $(I\mathbb{R})$ are themselves simply called intervals. $(I\mathbb{R})$ is the set of closed intervals of $\mathbb{R}$.[7] $I\mathbb{R}$ is the set of closed and bounded intervals of $\mathbb{R}$. Intervals of $I\mathbb{R}$ and of $(I\mathbb{R})$ are sets of real numbers. $-\infty$ and $+\infty$ are not elements of these intervals. It is fascinating that arithmetic operations can be introduced for all elements of the set $(I\mathbb{R})$ in an exception-free manner. This will be shown in the next section.

Under the assumption that the result is split into two distinct intervals after division by an interval that contains zero as an interior point, the set $(I\mathbb{R})$ can be seen as another algebraically closed subset of the power set $\mathbb{PR}$.

On a computer only subsets of the real numbers are representable. We assume now that S is the set of floating-point numbers of a given computer. An interval between two floating-point bounds represents the continuous set of real numbers between these bounds. Similarly, except for the empty set, also extended intervals represent continuous sets of real numbers.

To transform the eight cases of division by an interval which contains zero into computer executable operations we assume now that the operands A and B are

stop computing, or ignore the flag, or any other action.

[7] A subset of $\mathbb{R}$ is called closed if the complement is open.

floating-point intervals of IS. To obtain a computer representable result we round the result shown in the last column of Table 4.8 into the least computer representable superset. That is, the lower bound of the result has to be computed with rounding downwards and the upper bound with rounding upwards. Thus on the computer the eight cases of division by an interval which contains zero have to be performed as shown in Table 4.9.

case	$A = [a_1, a_2]$	$B = [b_1, b_2]$	$A \mathbin{\Diamond\!\!\!/} B$
1	$0 \in A$	$0 \in B$	$(-\infty, +\infty)$
2	$0 \notin A$	$B = [0, 0]$	$\emptyset$
3	$a_2 < 0$	$b_1 < b_2 = 0$	$[a_2 \mathbin{\nabla\!\!\!/} b_1, +\infty)$
4	$a_2 < 0$	$b_1 < 0 < b_2$	$(-\infty, a_2 \mathbin{\triangle\!\!\!/} b_2] \cup [a_2 \mathbin{\nabla\!\!\!/} b_1, +\infty)$
5	$a_2 < 0$	$0 = b_1 < b_2$	$(-\infty, a_2 \mathbin{\triangle\!\!\!/} b_2]$
6	$a_1 > 0$	$b_1 < b_2 = 0$	$(-\infty, a_1 \mathbin{\triangle\!\!\!/} b_1]$
7	$a_1 > 0$	$b_1 < 0 < b_2$	$(-\infty, a_1 \mathbin{\triangle\!\!\!/} b_1] \cup [a_1 \mathbin{\nabla\!\!\!/} b_2, +\infty)$
8	$a_1 > 0$	$0 = b_1 < b_2$	$[a_1 \mathbin{\nabla\!\!\!/} b_2, +\infty)$

Table 4.9. The eight cases of interval division with $A, B \in IS$, and $0 \in B$.

Table 4.10 shows the same cases as Table 4.9 in another layout.

	$B = [0, 0]$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$
$a_2 < 0$	$\emptyset$	$[a_2 \mathbin{\nabla\!\!\!/} b_1, +\infty)$	$(-\infty, a_2 \mathbin{\triangle\!\!\!/} b_2]$ $\cup\, [a_2 \mathbin{\nabla\!\!\!/} b_1, +\infty)$	$(-\infty, a_2 \mathbin{\triangle\!\!\!/} b_2]$
$a_1 \leq 0 \leq a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$0 < a_1$	$\emptyset$	$(-\infty, a_1 \mathbin{\triangle\!\!\!/} b_1]$	$(-\infty, a_1 \mathbin{\triangle\!\!\!/} b_1]$ $\cup\, [a_1 \mathbin{\nabla\!\!\!/} b_2, +\infty)$	$[a_1 \mathbin{\nabla\!\!\!/} b_2, +\infty)$

Table 4.10. The result of the interval division with $A, B \in IS$, and $0 \in B$.

Table 4.9 and Table 4.10 display the eight distinct cases of interval division $A \mathbin{\Diamond\!\!\!/} B$ with $A, B \in IS$ and $0 \in B$. On the computer the empty interval $\emptyset$ needs a particular encoding. (`+NaN`, `-NaN`) may be such an encoding. We explicitly stress that the symbols $-\infty$ and $+\infty$ are used here only to represent the resulting sets. These symbols are not elements of these sets.

4.10 Exception-free Arithmetic for Extended Intervals

Extending the remark of the last paragraph we begin this section with a basic statement. The formulas for the result of interval operations derived in this and the previous section will ultimately be executed by computer. There the bounds are floating-point numbers. Floating-point numbers and floating-point intervals are objects of different quality. A floating-point number is an approximate representation of a real number, while an interval is a precisely defined object. An operation mixing the two, which ought to be an interval, may not be precisely specified. It is thus not reasonable to define operations between floating-point numbers and intervals. If a user does indeed need to perform an operation between a floating-point number and a floating-point interval, he may do so by employing a transfer function – which may be predefined – which transforms its floating-point operand into a floating-point interval. In doing so, the user is made aware of the possible loss of meaning of the interval as a precise object. Prohibiting an automatic type transfer from floating-point numbers to floating-point intervals prevents exceptions of the IEEE floating-point arithmetic standard from being introduced into interval arithmetic. Here the symbols $-\infty$ and $+\infty$ are only used for the description of particular sets of real numbers.

Table 4.11 and Table 4.12 show the results of addition and subtraction of intervals of $(I\mathbb{R})$. Any operation with the empty set always delivers the empty set.

Addition	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, a_2 + b_2]$	$(-\infty, a_2 + b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2]$	$(-\infty, a_2 + b_2]$	$[a_1 + b_1, a_2 + b_2]$	$[a_1 + b_1, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$(-\infty, +\infty)$	$[a_1 + b_1, +\infty)$	$[a_1 + b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.11. Addition for intervals of $(I\mathbb{R})$.

Subtraction	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, +\infty)$	$(-\infty, a_2 - b_1]$	$(-\infty, a_2 - b_1]$	$(-\infty, +\infty)$
$[a_1, a_2]$	$[a_1 - b_2, +\infty)$	$[a_1 - b_2, a_2 - b_1]$	$(-\infty, a_2 - b_1]$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$[a_1 - b_2, +\infty)$	$[a_1 - b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.12. Subtraction for intervals of $(I\mathbb{R})$.

To obtain these results in the operands A and B bounds like $-\infty$ and $+\infty$ are replaced by a very large negative and a very large positive number respectively. Then the basic rules for addition and subtraction for regular intervals $[a_1, a_2] + [b_1, b_2] =$

$[a_1+b_1, a_2+b_2]$ and $[a_1, a_2]-[b_1, b_2] = [a_1-b_2, a_2-b_1]$ are applied. In the resulting formulas then the very large negative number is shifted to $-\infty$ and the very large positive number to $+\infty$. Finally the rules $\infty+x = \infty, -\infty+x = -\infty, \infty+\infty = \infty$, and $-\infty + (-\infty) = -\infty$ with $x \in \mathbb{R}$ are applied. These rules are well established in real analysis.

The rules for division by an interval which contains zero developed in the last section together with the rules for addition and subtraction for intervals of $(I\mathbb{R})$ shown in Table 4.11 and Table 4.12 allow an exception-free execution of the extended interval Newton method of interval mathematics.

The following three tables (Table 4.13, Table 4.14, and Table 4.15) show the results of multiplication and division for intervals of $(I\mathbb{R})$. In the case of division again the two cases $0 \notin B$ and $0 \in B$ are studied separately.

The general procedure to obtain these results is very similar to the cases of addition and subtraction. Bounds like $-\infty$ and $+\infty$ in the operands for A and B are replaced by a very large negative and a very large positive number respectively. Then the basic rules for multiplication and for division with $0 \notin B$ for regular intervals of $I\mathbb{R}$ are applied. In Table 4.13 and Table 4.14 these rules are repeated in the rows and columns in the left upper corner. In the resulting formulas the very large negative number is then shifted to $-\infty$ and the very large positive number to $+\infty$. Finally, very simple and well-established rules of real analysis like $\infty * x = \infty$ for $x > 0$, $\infty * x = -\infty$ for $x < 0$, $x/\infty = x/-\infty = 0$, $\infty*\infty = \infty$, $(-\infty)*\infty = -\infty$ are applied together with variants obtained by applying the sign rules and the law of commutativity.

Table 4.15 shows the results of division for intervals of $(I\mathbb{R})$ with $0 \in B$. The basic procedure to obtain the results of the operations is similar to those in the former cases. However, two situations have to be treated separately. These are the cases shown in rows 1 and 2 of Table 4.8. Zero is now always an element of the divisor, i.e., $0 \in B$.

If $0 \in A$ and $0 \in B$ (row 1 of Table 4.8), the result consists of all real numbers, i.e., $A/B = (-\infty, +\infty)$. This applies to rows 2, 5, 6, and 8 of Table 4.15.

If $0 \notin A$ and $B = [0, 0]$ (row 2 of Table 4.8), the result of the division is the empty set, i.e., $A/B = \emptyset$. This applies to rows 1, 3, 4, and 7 of column 1 of Table 4.15.

The remaining cases of Table 4.15 can be treated very similarly to the former cases. Bounds $-\infty$ and $+\infty$ in the extended interval operands are first replaced by a very large negative number and a very large positive number respectively. Then the rules for division of regular intervals with $0 \in B$ are applied. These rules are shown in Table 4.8. In Table 4.15 these rules are repeated in rows and columns in the left upper corner. After shifting the very large negative number to $-\infty$ and the very large positive number to $+\infty$ and application of well-established rules of real analysis the rules shown in Table 4.15 are easily obtained.

In summary it can be said that after a possible splitting of an improper interval into two separate intervals the result of arithmetic operations for intervals of $(I\mathbb{R})$ always leads to intervals of $(I\mathbb{R})$ again. No exceptions do occur performing these operations.

	$[b_1, b_2]$	$[b_1, b_2]$	$[b_1, b_2]$		$(-\infty, b_2]$	$(-\infty, b_2]$	$[b_1, +\infty)$	$[b_1, +\infty)$	
Multiplication	$b_2 \leq 0$	$b_1 < 0 < b_2$	$b_1 \geq 0$	$[0, 0]$	$b_2 \leq 0$	$b_2 \geq 0$	$b_1 \leq 0$	$b_1 \geq 0$	$(-\infty, +\infty)$
$[a_1, a_2], a_2 \leq 0$	$[a_2b_2, a_1b_1]$	$[a_1b_2, a_1b_1]$	$[a_1b_2, a_2b_1]$	$[0, 0]$	$[a_2b_2, +\infty)$	$[a_1b_2, +\infty)$	$(-\infty, a_1b_1]$	$(-\infty, a_2b_1]$	$(-\infty, +\infty)$
$a_1 < 0 < a_2$	$[a_2b_1, a_1b_1]$	$[\min(a_1b_2, a_2b_1), \max(a_1b_1, a_2b_2)]$	$[a_1b_2, a_2b_2]$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 \geq 0$	$[a_2b_1, a_1b_2]$	$[a_2b_1, a_2b_2]$	$[a_1b_1, a_2b_2]$	$[0, 0]$	$(-\infty, a_1b_2]$	$(-\infty, a_2b_2]$	$[a_2b_1, +\infty)$	$[a_1b_1, +\infty)$	$(-\infty, +\infty)$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2], a_2 \leq 0$	$[a_2b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2b_1]$	$[0, 0]$	$[a_2b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2b_1]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 \geq 0$	$[a_2b_1, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2b_2]$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1b_1]$	$(-\infty, +\infty)$	$[a_1b_2, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1b_2]$	$(-\infty, +\infty)$	$[a_1b_1, +\infty)$	$[0, 0]$	$(-\infty, a_1b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.13. Multiplication for intervals of $(I\mathbb{R})$.

Division	$[b_1, b_2]$	$[b_1, b_2]$	$(-\infty, b_2]$	$[b_1, +\infty)$
$0 \notin \boldsymbol{B}$	$b_2 < 0$	$b_1 > 0$	$b_2 < 0$	$b_1 > 0$
$[a_1, a_2], a_2 \leq 0$	$[a_2/b_1, a_1/b_2]$	$[a_1/b_1, a_2/b_2]$	$[0, a_1/b_2]$	$[a_1/b_1, 0]$
$[a_1, a_2], a_1 < 0 < a_2$	$[a_2/b_2, a_1/b_2]$	$[a_1/b_1, a_2/b_1]$	$[a_2/b_2, a_1/b_2]$	$[a_1/b_1, a_2/b_1]$
$[a_1, a_2], a_1 \geq 0$	$[a_2/b_2, a_1/b_1]$	$[a_1/b_2, a_2/b_1]$	$[a_2/b_2, 0]$	$[0, a_2/b_1]$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2], a_2 \leq 0$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, 0]$
$(-\infty, a_2], a_2 \geq 0$	$[a_2/b_2, +\infty)$	$(-\infty, a_2/b_1]$	$[a_2/b_2, +\infty)$	$(-\infty, a_2/b_1]$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1/b_2]$	$[a_1/b_1, +\infty)$	$(-\infty, a_1/b_2]$	$[a_1/b_1, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1/b_1]$	$[a_1/b_2, +\infty)$	$(-\infty, 0]$	$[0, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.14. Division for intervals of $(I\mathbb{R})$ with $0 \notin B$.

Division	$B =$	$[b_1, b_2]$	$[b_1, b_2]$	$[b_1, b_2]$	$(-\infty, b_2]$	$(-\infty, b_2]$	$[b_1, +\infty)$	$[b_1, +\infty)$	
$\mathbf{0} \in \boldsymbol{B}$	$[0,0]$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$	$b_2 = 0$	$b_2 > 0$	$b_1 < 0$	$b_1 = 0$	$(-\infty, +\infty)$
$[a_1, a_2]$, $a_2 < 0$	$\varnothing$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$ $\cup\, [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, a_2/b_2]$ $\cup\, [0, +\infty)$	$(-\infty, 0]$ $\cup\, [a_2/b_1, +\infty)$	$(-\infty, 0]$	$(-\infty, +\infty)$
$[a_1, a_2]$, $a_1 \leq 0 \leq a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2]$, $a_1 > 0$	$\varnothing$	$(-\infty, a_1/b_1]$	$(-\infty, a_1/b_1]$ $\cup\, [a_1/b_2, +\infty)$	$[a_1/b_2, +\infty)$	$(-\infty, 0]$	$(-\infty, 0]$ $\cup\, [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $\cup\, [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$, $a_2 < 0$	$\varnothing$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$ $\cup\, [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, a_2/b_2]$ $\cup\, [0, +\infty)$	$(-\infty, 0]$ $\cup\, [a_2/b_1, +\infty)$	$(-\infty, 0]$	$(-\infty, +\infty)$
$(-\infty, a_2]$, $a_2 > 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$, $a_1 < 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$, $a_1 > 0$	$\varnothing$	$(-\infty, a_1/b_1]$	$(-\infty, a_1/b_1]$ $\cup\, [a_1/b_2, +\infty)$	$[a_1/b_2, +\infty)$	$(-\infty, 0]$	$(-\infty, 0]$ $\cup\, [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $\cup\, [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.15. Division for intervals of $(I\mathbb{R})$ with $0 \in B$.

For the development in the preceding sections it was essential to distinguish between open and closed interval bounds. Thus besides of the empty set intervals of $(I\mathbb{R})$ are enclosed in square and/or round brackets. If the bracket adjacent to a bound is round, the bound is not included in the interval; if it is square, the bound is included in the interval.

On the computer, of course, the lower bound of the result has to be rounded downwards and the upper bound rounded upwards. The bounds $-\infty$ and $+\infty$ are not changed by these roundings.

We summarize the complete set of arithmetic operations for interval arithmetic in (IS) that should be provided on the computer in the next section. (IS) is the set of closed real intervals with bounds of S.

4.11 Extended Interval Arithmetic on the Computer

Addition	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, a_2 \overset{\triangle}{+} b_2]$	$(-\infty, a_2 \overset{\triangle}{+} b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2]$	$(-\infty, a_2 \overset{\triangle}{+} b_2]$	$[\boldsymbol{a_1 \overset{\triangledown}{+} b_1, a_2 \overset{\triangle}{+} b_2}]$	$[a_1 \overset{\triangledown}{+} b_1, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$(-\infty, +\infty)$	$[a_1 \overset{\triangledown}{+} b_1, +\infty)$	$[a_1 \overset{\triangledown}{+} b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.16. Addition of extended intervals on the computer.

Subtraction	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, +\infty)$	$(-\infty, a_2 \overset{\triangle}{-} b_1]$	$(-\infty, a_2 \overset{\triangle}{-} b_1]$	$(-\infty, +\infty)$
$[a_1, a_2]$	$[a_1 \overset{\triangledown}{-} b_2, +\infty)$	$[\boldsymbol{a_1 \overset{\triangledown}{-} b_2, a_2 \overset{\triangle}{-} b_1}]$	$(-\infty, a_2 \overset{\triangle}{-} b_1]$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$[a_1 \overset{\triangledown}{-} b_2, +\infty)$	$[a_1 \overset{\triangledown}{-} b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.17. Subtraction of extended intervals on the computer.

Multiplication	$[b_1, b_2]$ $b_2 \leq 0$	$[b_1, b_2]$ $b_1 < 0 < b_2$	$[b_1, b_2]$ $b_1 \geq 0$	$[0, 0]$	$(-\infty, b_2]$ $b_2 \leq 0$	$(-\infty, b_2]$ $b_2 \geq 0$	$[b_1, +\infty)$ $b_1 \leq 0$	$[b_1, +\infty)$ $b_1 \geq 0$	$(-\infty, +\infty)$
$[a_1, a_2], a_2 \leq 0$	$[\boldsymbol{a_2 \triangledown b_2, a_1 \triangle b_1}]$	$[\boldsymbol{a_1 \triangledown b_2, a_1 \triangle b_1}]$	$[\boldsymbol{a_1 \triangledown b_2, a_2 \triangle b_1}]$	$[0, 0]$	$[a_2 \triangledown b_2, +\infty)$	$[a_1 \triangledown b_2, +\infty)$	$(-\infty, a_1 \triangle b_1]$	$(-\infty, a_2 \triangle b_1]$	$(-\infty, +\infty)$
$a_1 < 0 < a_2$	$[\boldsymbol{a_2 \triangledown b_1, a_1 \triangle b_1}]$	$[\mathbf{min}(\boldsymbol{a_1 \triangledown b_2, a_2 \triangledown b_1}),$ $\mathbf{max}(\boldsymbol{a_1 \triangle b_1, a_2 \triangle b_2})]$	$[\boldsymbol{a_1 \triangledown b_2, a_2 \triangle b_2}]$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 \geq 0$	$[\boldsymbol{a_2 \triangledown b_1, a_1 \triangle b_2}]$	$[\boldsymbol{a_2 \triangledown b_1, a_2 \triangle b_2}]$	$[\boldsymbol{a_1 \triangledown b_1, a_2 \triangle b_2}]$	$[0, 0]$	$(-\infty, a_1 \triangle b_2]$	$(-\infty, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, +\infty)$	$[a_1 \triangledown b_1, +\infty)$	$(-\infty, +\infty)$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2], a_2 \leq 0$	$[a_2 \triangledown b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \triangle b_1]$	$[0, 0]$	$[a_2 \triangledown b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \triangle b_1]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 \geq 0$	$[a_2 \triangledown b_1, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \triangle b_2]$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1 \triangle b_1]$	$(-\infty, +\infty)$	$[a_1 \triangledown b_2, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1 \triangle b_2]$	$(-\infty, +\infty)$	$[a_1 \triangledown b_1, +\infty)$	$[0, 0]$	$(-\infty, a_1 \triangle b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1 \triangledown b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.18. Multiplication of extended intervals on the computer.

Division $\mathbf{0} \notin \boldsymbol{B}$	$[b_1, b_2]$ $b_2 < 0$	$[b_1, b_2]$ $b_1 > 0$	$(-\infty, b_2]$ $b_2 < 0$	$[b_1, +\infty)$ $b_1 > 0$
$[a_1, a_2], a_2 \leq 0$	$[\boldsymbol{a_2} \bigtriangledown \boldsymbol{b_1}, \boldsymbol{a_1} \bigtriangleup \boldsymbol{b_2}]$	$[\boldsymbol{a_1} \bigtriangledown \boldsymbol{b_1}, \boldsymbol{a_2} \bigtriangleup \boldsymbol{b_2}]$	$[0, a_1 \bigtriangleup b_2]$	$[a_1 \bigtriangledown b_1, 0]$
$[a_1, a_2], a_1 < 0 < a_2$	$[\boldsymbol{a_2} \bigtriangledown \boldsymbol{b_2}, \boldsymbol{a_1} \bigtriangleup \boldsymbol{b_2}]$	$[\boldsymbol{a_1} \bigtriangledown \boldsymbol{b_1}, \boldsymbol{a_2} \bigtriangleup \boldsymbol{b_1}]$	$[a_2 \bigtriangledown b_2, a_1 \bigtriangleup b_2]$	$[a_1 \bigtriangledown b_1, a_2 \bigtriangleup b_1]$
$[a_1, a_2], a_1 \geq 0$	$[\boldsymbol{a_2} \bigtriangledown \boldsymbol{b_2}, \boldsymbol{a_1} \bigtriangleup \boldsymbol{b_1}]$	$[\boldsymbol{a_1} \bigtriangledown \boldsymbol{b_2}, \boldsymbol{a_2} \bigtriangleup \boldsymbol{b_1}]$	$[a_2 \bigtriangledown b_2, 0]$	$[0, a_2 \bigtriangleup b_1]$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2], a_2 \leq 0$	$[a_2 \bigtriangledown b_1, +\infty)$	$(-\infty, a_2 \bigtriangleup b_2]$	$[0, +\infty)$	$(-\infty, 0]$
$(-\infty, a_2], a_2 \geq 0$	$[a_2 \bigtriangledown b_2, +\infty)$	$(-\infty, a_2 \bigtriangleup b_1]$	$[a_2 \bigtriangledown b_2, +\infty)$	$(-\infty, a_2 \bigtriangleup b_1]$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1 \bigtriangleup b_2]$	$[a_1 \bigtriangledown b_1, +\infty)$	$(-\infty, a_1 \bigtriangleup b_2]$	$[a_1 \bigtriangledown b_1, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1 \bigtriangleup b_1]$	$[a_1 \bigtriangledown b_2, +\infty)$	$(-\infty, 0]$	$[0, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.19. Division of extended intervals with $0 \notin B$ on the computer.

Division	$B =$	$[b_1, b_2]$	$[b_1, b_2]$	$[b_1, b_2]$	$(-\infty, b_2]$	$(-\infty, b_2]$	$[b_1, +\infty)$	$[b_1, +\infty)$	
$\mathbf{0} \in \boldsymbol{B}$	$[0, 0]$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$	$b_2 = 0$	$b_2 > 0$	$b_1 < 0$	$b_1 = 0$	$(-\infty, +\infty)$
$[a_1, a_2], a_2 < 0$	$\varnothing$	$[\boldsymbol{a_2} \triangledown \boldsymbol{b_1}, +\infty)$	$(-\infty, \boldsymbol{a_2} \triangle \boldsymbol{b_2}]$ $\cup\, [\boldsymbol{a_2} \triangledown \boldsymbol{b_1}, +\infty)$	$(-\infty, \boldsymbol{a_2} \triangle \boldsymbol{b_2}]$	$[0, +\infty)$	$(-\infty, a_2 \triangle b_2]$ $\cup\, [0, +\infty)$	$(-\infty, 0]$ $\cup\, [a_2 \triangledown b_1, +\infty)$	$(-\infty, 0]$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 \leq 0 \leq a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 > 0$	$\varnothing$	$(-\infty, \boldsymbol{a_1} \triangle \boldsymbol{b_1}]$	$(-\infty, \boldsymbol{a_1} \triangle \boldsymbol{b_1}]$ $\cup\, [\boldsymbol{a_1} \triangledown \boldsymbol{b_2}, +\infty)$	$[\boldsymbol{a_1} \triangledown \boldsymbol{b_2}, +\infty)$	$(-\infty, 0]$	$(-\infty, 0]$ $\cup\, [a_1 \triangledown b_2, +\infty)$	$(-\infty, a_1 \triangle b_1]$ $\cup\, [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 < 0$	$\varnothing$	$[a_2 \triangledown b_1, +\infty)$	$(-\infty, a_2 \triangle b_2]$ $\cup\, [a_2 \triangledown b_1, +\infty)$	$(-\infty, a_2 \triangle b_2]$	$[0, +\infty)$	$(-\infty, a_2 \triangle b_2]$ $\cup\, [0, +\infty)$	$(-\infty, 0]$ $\cup\, [a_2 \triangledown b_1, +\infty)$	$(-\infty, 0]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 > 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 < 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 > 0$	$\varnothing$	$(-\infty, a_1 \triangle b_1]$	$(-\infty, a_1 \triangle b_1]$ $\cup\, [a_1 \triangledown b_2, +\infty)$	$[a_1 \triangledown b_2, +\infty)$	$(-\infty, 0]$	$(-\infty, 0]$ $\cup\, [a_1 \triangledown b_2, +\infty)$	$(-\infty, a_1 \triangle b_1]$ $\cup\, [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 4.20. Division of extended intervals with $0 \in B$ on the computer.

4.12 Implementation of Extended Interval Arithmetic

The rules for the operations of extended intervals on the computer in Tables 4.16–4.20 look rather complicated. Their implementation seems to require a major number of case distinctions. The situation, however, can be greatly simplified by the following hints.

We first consider the operations addition, subtraction, multiplication, and division by an interval which does not contain zero. On the computer actually only the basic rules for addition

$$[a_1, a_2] + [b_1, b_2] = [a_1 \mathbin{\triangledown\!\!\!\!\!+} b_1, a_2 \mathbin{\triangle\!\!\!\!\!+} b_2], \tag{4.12.1}$$

for subtraction

$$[a_1, a_2] - [b_1, b_2] = [a_1 \mathbin{\triangledown\!\!\!\!-} b_2, a_2 \mathbin{\triangle\!\!\!\!-} b_1], \tag{4.12.2}$$

for multiplication the rules of Table 4.1, and for division those of Table 4.2 are to be provided. In the Tables 4.16–4.18 these rules are printed in bold letters.

The remaining rules shown in the Tables 4.16–4.18 can automatically be produced out of these basic rules by the computer itself if a few well-established rules for computing with $-\infty$ and $+\infty$ are formally applied. With $x \in \mathbb{R}$ these rules are

$$\begin{array}{cc}
\infty + x = \infty, & -\infty + x = -\infty, \\
-\infty + (-\infty) = (-\infty) \cdot \infty = -\infty, & \infty + \infty = \infty \cdot \infty = \infty, \\
\infty \cdot x = \infty \ \text{ for } x > 0, & \infty \cdot x = -\infty \ \text{ for } x < 0, \\
\dfrac{x}{\infty} = \dfrac{x}{-\infty} = 0, &
\end{array}$$

together with variants obtained by applying the sign rules and the law of commutativity. If in an interval operand a bound is $-\infty$ or $+\infty$ the multiplication with 0 is performed as if the following rules would hold

$$0 \cdot (-\infty) = 0 \cdot (+\infty) = (-\infty) \cdot 0 = (+\infty) \cdot 0 = 0.$$

These rules have no meaning otherwise.

The case of division of an extended interval by an interval that contains zero, Table 4.20 is a little more complicated. Basically it can be treated very similarly than the other cases. However, two situations have to be dealt with separately. Zero can now be an element of A and in one case we have $B = [0, 0]$.

If $0 \in A$ and $0 \in B$ then by row 1 of Table 4.9 the result consists of the set of all real numbers, i.e., $A/B = (-\infty, +\infty)$. This applies to rows 2, 5, 6, and 8 of Table 4.20.

If $0 \notin A$ and $B = [0, 0]$ then by row 2 of Table 4.9 the result is the empty set, i.e., $A/B = \varnothing$. This applies to rows 1, 3, 4, and 7 of column 1 of Table 4.20.

Realization of the remaining cases of Table 4.20 then can automatically be produced by the computer itself out of the few rules printed in bold letters in Table 4.20 by applying the established rules for $-\infty$ and $+\infty$ shown above.

4.13 Comparison Relations and Lattice Operations

Three comparison relations are important for intervals of $I\mathbb{R}$ and $(I\mathbb{R})$:

$$\textit{equality}, \quad \textit{less than or equal}, \quad \text{and} \quad \textit{set inclusion}. \tag{4.13.1}$$

Since bounds for intervals of $(I\mathbb{R})$ may be $-\infty$ or $+\infty$ these comparison relations are defined as if performed in the complete lattice $\{\mathbb{R}^*, \leq\}$ with $\mathbb{R}^* := \mathbb{R} \cup \{-\infty\} \cup \{+\infty\}$.

Let A and B be intervals of $(I\mathbb{R})$ with bounds $a_1 \leq a_2$ and $b_1 \leq b_2$ respectively. Then the relations *equality* and *less than or equal* in $(I\mathbb{R})$ are defined by:

$$\begin{aligned} A = B \quad &:\Leftrightarrow \quad a_1 = b_1 \wedge a_2 = b_2, \\ A \leq B \quad &:\Leftrightarrow \quad a_1 \leq b_1 \wedge a_2 \leq b_2. \end{aligned}$$

With the order relation $\leq$, $\{(I\mathbb{R}), \leq\}$ is a lattice. If either A or B is the empty interval the result is false. The *greatest lower bound* (glb) and the *least upper bound* (lub) of $A, B \in (I\mathbb{R})$ are the intervals

$$\begin{aligned} \text{glb}(A, B) \quad &:= \quad [\min(a_1, b_1), \min(a_2, b_2)], \\ \text{lub}(A, B) \quad &:= \quad [\max(a_1, b_1), \max(a_2, b_2)]. \end{aligned}$$

The inclusion relation in $(I\mathbb{R})$ is defined by

$$A \subseteq B :\Leftrightarrow b_1 \leq a_1 \wedge a_2 \leq b_2. \tag{4.13.2}$$

With the relation $\subseteq$, $\{(I\mathbb{R}), \subseteq\}$ is also a lattice. If A is the empty interval the result is true. If B is the empty interval the result is false. The least element in $\{(I\mathbb{R}), \subseteq\}$ is the empty set $\emptyset$ and the greatest element is the set $\mathbb{R}$, i.e., the interval $(-\infty, +\infty)$. The infimum of two elements $A, B \in (I\mathbb{R})$ is the intersection and the supremem is the interval hull (convex hull):

$$\begin{aligned} \inf(A, B) \quad &:= \quad [\max(a_1, b_1), \min(a_2, b_2)] \quad \text{or the empty set } \emptyset, \\ \sup(A, B) \quad &:= \quad [\min(a_1, b_1), \max(a_2, b_2)]. \end{aligned}$$

The intersection of an interval with the empty set is the empty set. The interval hull with the empty set is the other operand.

If in the formulas for glb(A, B), lub(A, B), inf(A, B), sup(A, B), a bound is $-\infty$ or $+\infty$ a round bracket should be used at this interval bound to denote the resulting interval. This bound is not an element of the interval.

If in any of the comparison relations defined here both operands are the empty set, the result is true. If in (4.13.2) A is the empty set the result is true. Otherwise the result is false if in any of the three comparison relations only one operand is the empty set.

A particular case of inclusion is the relation *element of*. It is defined by

$$a \in B :\Leftrightarrow b_1 \leq a \wedge a \leq b_2. \tag{4.13.3}$$

Another useful operation is to check whether an interval $[a_1, a_2]$ is a proper interval. It computes the result $a_1 \leq a_2$. It is used to test whether the result of an interval division consists of a proper or an improper interval.

4.14 Algorithmic Implementation of Interval Multiplication and Division

In this section we give an algorithmic description of interval multiplication and interval division. In the case of interval division by an interval that contains zero we restrict the algorithm to the original solution discussed in Section 4.9. We leave the development of an algorithm for the implementation of the general case developed in Sections 4.10 and 4.11 as an exercise to the user. An algorithm for interval multiplication has to realize Table 4.5 and an algorithm for interval division has to realize Tables 4.10 and 4.6. In the following algorithms *lb* denotes the lower bound and *ub* the upper bound of the result interval.

We begin with the multiplication.

```
if $(a_1 < 0 \wedge a_2 > 0 \wedge b_1 < 0 \wedge b_2 > 0)$ then
    $p := a_1 \triangledown b_2; q := a_2 \triangledown b_1$
    $lb := \min(p, q)$
    $r := a_1 \triangle b_1; s := a_2 \triangle b_2$
    $ub := \max(r, s)$
else
    if $(b_2 \leq 0 \vee (b_1 < 0 \wedge a_1 \geq 0))$ then $p := a_2$ else $p := a_1$
    if $(a_2 \leq 0 \vee (a_1 < 0 \wedge b_2 \geq 0))$ then $q := b_2$ else $q := b_1$
    $lb := p \triangledown q$
    if $(b_1 \geq 0 \vee (a_1 \geq 0 \wedge b_2 > 0))$ then $r := a_2$ else $r := a_1$
    if $(a_1 \geq 0 \vee (a_2 > 0 \wedge b_1 \geq 0))$ then $s := b_2$ else $s := b_1$
    $ub := r \triangle s$
```

The 14 cases of interval division are realized by the following algorithm.

```
if $(b_2 < 0 \vee b_1 > 0)$ then
    if $(b_2 < 0))$ then $p := a_2$ else $p := a_1$
    if $(a_1 \geq 0 \vee (a_2 > 0 \wedge b_1 < 0))$ then $q := b_2$ else $q := b_1$
    $lb := p \triangledown q$
    if $(b_1 > 0))$ then $r := a_2$ else $r := a_1$
    if $(a_2 \leq 0 \vee (a_1 < 0 \wedge b_1 < 0))$ then $s := b_2$ else $s := b_1$
```

```
        $ub := r \triangle s$
else if $(a_1 \leq 0 \wedge a_2 \geq 0 \wedge b_1 \leq 0 \wedge b_2 \geq 0)$ then $[lb, ub] := [-\infty, +\infty]$
else if $((a_2 < 0 \vee a_1 > 0) \wedge b_1 = 0 \wedge b_2 = 0)$ then $[lb, ub] := [,]$
else if $(a_2 < 0 \wedge b_2 = 0)$ then $[lb, ub] := [a_2 \triangledown b_1, +\infty]$
else if $(a_2 < 0 \wedge b_1 = 0)$ then $[lb, ub] := [-\infty, a_2 \triangle b_2]$
else if $(a_1 > 0 \wedge b_2 = 0)$ then $[lb, ub] := [-\infty, a_1 \triangle b_1]$
else if $(a_1 > 0 \wedge b_1 = 0)$ then $[lb, ub] := [a_1 \triangledown b_2, +\infty]$
else if $(a_2 < 0 \wedge b_1 < 0 \wedge b_2 > 0)$ then $[lb, ub] := [a_2 \triangledown b_1, a_2 \triangle b_2]$
else if $(a_1 > 0 \wedge b_1 < 0 \wedge b_2 > 0)$ then $[lb, ub] := [a_1 \triangledown b_2, a_1 \triangle b_1]$
```

In this algorithm $[,]$ denotes the empty set. The algorithm is organized in such a way that the more complicated cases, where the result consists of two separate intervals, appear at the end. Even in these cases the results are represented as a single interval which overlaps the point infinity (i.e., $lb > ub$). Thus the result of an interval division always just consists of two bounds.

Hardware support for interval arithmetic will be discussed in Chapter 7.

Part II

Implementation of Arithmetic on Computers

Chapter 5

Floating-Point Arithmetic

Thus far our considerations have proceeded under the general assumption that the set R in Figure 1 is a linearly ordered ringoid. We are now going to be more specific and assume that R is the linearly ordered field of real numbers and that the subsets D and S are special screens, which are called floating-point systems. In Section 5.1 we briefly review the definition of the real numbers and their representation by b-adic expansions. Section 5.2 deals with floating-point numbers and systems. We also discuss several basic roundings of the real numbers into a floating-point system, and we derive error bounds for these roundings.

In Section 5.3 we consider floating-point operations defined by semimorphisms, and we derive error bounds for these operations. Then we consider the operations defined in the product sets listed under S and D in Figure 1, and we derive error formulas and bounds for these operations also. All this is done under the assumption that the operations are defined by semimorphisms.

For scalar product and matrix multiplication, we also derive error formulas and bounds for the conventional definition of the operations. The error formulas are simpler if the operations are defined by semimorphisms, and the error bounds are smaller by a factor of at least n compared to the bounds obtained by the conventional definition of the operations. These two properties are reproduced in the error analysis of many algorithms.

Finally we review the so-called IEEE floating-point arithmetic standard.

5.1 Definition and Properties of the Real Numbers

Two methods of defining the real numbers are most commonly used. These may be called the constructive method and the axiomatic method. We have mentioned these concepts already in the introduction of this book. For clarity we briefly consider the axiomatic method here.

The real numbers $\{\mathbb{R}, +, \cdot, \leq\}$ can be defined as a conditionally complete linearly ordered field, i.e.,

I $\{\mathbb{R}, +, \cdot\}$ is a field.

II $\{\mathbb{R}, \leq\}$ is a conditionally complete, linearly ordered set.

III The following compatibility properties hold between the algebraic and the order structure in $\mathbb{R}$:

(a) $\bigwedge\limits_{a,b,c\in\mathbb{R}} (a \le b \Rightarrow a + c \le b + c).$

(b) $\bigwedge\limits_{a,b,c\in\mathbb{R}} (a \le b \wedge c \ge 0 \Rightarrow a \cdot c \le b \cdot c).$

In the field $\{\mathbb{R}, +, \cdot\}$ both, addition and multiplication have the structure of a commutative group. 0 and 1 are the neutral elements of addition and multiplication in $\mathbb{R}$ respectively, with $0 \neq 1$. Both groups are connected by the distributive law: $a \cdot (b + c) = a \cdot b + a \cdot c$. The order structure in $\{\mathbb{R}, +, \cdot, \le\}$ fulfills our axioms $(O1), (O2), (O3), (O4)$, and $(O6)$.

These few properties are the basis for the huge construct that is real analysis. All further properties of the real numbers are derived from them. In this connection the following theorem is of fundamental interest.

Theorem 5.1. *Two conditionally complete, linearly ordered fields R and R' are isomorphic.* ■

A proof of this theorem can be found in many places in the literature, for instance, in [416].

Since corresponding elements of isomorphic structures can be identified with each other, Theorem 5.1 asserts that there exists at most one conditionally complete, linearly ordered field. Such a structure does in fact exist since the constructive method mentioned in the introduction of this book actually produces one. In this sense the real numbers defined by that method represent the only realization of a conditionally complete, linearly ordered field.

Theorem 5.1 also implies that the real numbers cannot be extended by additional elements without giving up or weakening one of the above properties. By partly abandoning the order structure, the complex numbers may be obtained. Sometimes the two elements $-\infty$ and $+\infty$ with the property $-\infty < a < +\infty$, for all $a \in \mathbb{R}$, are adjoined to the real numbers. With these elements $\{\mathbb{R} \cup \{-\infty\} \cup \{+\infty\}, \le\}$ is a complete lattice. However, it is no longer a linearly ordered field. For instance, $a + \infty = b + \infty$ even if $a < b$.

For real numbers the *absolute value* is defined as a mapping $|\cdot| : \mathbb{R} \to \mathbb{R}_+ := \{x \in \mathbb{R} \mid x \ge 0\}$ with the property

$$\bigwedge_{a\in\mathbb{R}} |a| := \sup(a, -a) = \begin{cases} a & \text{for } a \ge 0, \\ -a & \text{for } a \le 0. \end{cases}$$

Using the absolute value, a *distance* $\rho(a, b) := |a - b|$ can be defined which has all the properties of a *metric*. With the latter, the concept of the limit of a *sequence* and the limit of a *series* can be defined:

A sequence[1] $\{a_n\} \in \mathbb{R}^{\mathbb{N}} := \{\alpha \mid \alpha : \mathbb{N} \to \mathbb{R}\}$ of elements $a_n \in \mathbb{R}$ is called *convergent* to a number $a \in \mathbb{R}$, in which event we write $\lim a_n = a$ if

$$\bigwedge_{\epsilon>0} \; \bigvee_{N(\epsilon)\in\mathbb{N}} \; \bigwedge_{n>N(\epsilon)} \; |a_n - a| < \epsilon.$$

Otherwise the sequence is called *divergent*.

Using a sequence $\{a_n\}$, an associated sequence $\{s_n\}$ can be defined, where

$$s_n := \sum_{\nu=1}^{n} a_\nu.$$

If $\{s_n\}$ is convergent to a limit s, we say that the *infinite series*

$$\sum_{\nu=1}^{\infty} a_\nu = a_1 + a_2 + a_3 + \ldots$$

is convergent and write

$$s = \lim_{n\to\infty} s_n = \sum_{\nu=1}^{\infty} a_\nu.$$

Here s is called the *sum* of the series. The numbers s_n are called the *partial sums*. If $\{s_n\}$ is divergent, the series is called divergent.

Beyond the great usefulness of an abstract theory it must be remembered that analysis has been developed for its application in nature. In applied mathematics, apart from the abstract theory and symbolic representation of real numbers, their value is of great importance. The value of a real number is represented in special number systems. Applied mathematics, therefore, also has to consider particular representations of real numbers, algorithms for their computation, and algorithms for computing the result of operations on real numbers.

We begin our study of the representation of real numbers with the following well-known theorem on *b-adic systems*. This theorem may be shown to follow from the axioms I–III, see [463].

Theorem 5.2. *Every real number x is uniquely represented by a b-adic expansion of the form*

$$x = \circ d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} d_{-3} \cdots = \circ \sum_{\nu=n}^{-\infty} d_\nu b^\nu \tag{5.1.1}$$

with $\circ \in \{+,-\}$ and $b \in \mathbb{N}$, $b > 1$. Here the $d_i, i = n(-1) - \infty$, are integers that satisfy the inequalities

$$0 \le d_i \le b-1 \textit{ for all } i = n(-1) - \infty, \tag{5.1.2}$$

$$d_i \le b-2 \textit{ for infinitely many } i. \tag{5.1.3}$$

[1] $\mathbb{N}$ denotes the set of natural numbers.

Every b-adic expansion (5.1.1)–(5.1.3) *represents exactly one real number.*

If we approximate (5.1.1) *by the finite sum*

$$s_m := \circ \sum_{\nu=n}^{-m} d_\nu b^\nu,$$

the following error bound holds:

$$0 \leq |x - s_m| < b^{-m}. \tag{5.1.4}$$

■

In (5.1.1) b is called the *base* of the number system. The $d_i, i = n(-1) - \infty$, are called the *digits*.

Theorem 5.2 asserts that there exists a one-to-one correspondence between the real numbers and the b-adic expansions of the form (5.1.1)–(5.1.3). Condition (5.1.3) needs some interpretation. Without it the representation of a real number by a b-adic expansion (5.1.1) need not be unique. Consider, for instance, the two decimal expansions with the partial sums

$$s_n = 3 \cdot 10^{-1} + 0 \cdot 10^{-2} + 0 \cdot 10^{-3} + \cdots + 0 \cdot 10^{-n}$$

and

$$s'_n = 2 \cdot 10^{-1} + 9 \cdot 10^{-2} + 9 \cdot 10^{-3} + \cdots + 9 \cdot 10^{-n}.$$

Both sequences $\{s_n\}$ and $\{s'_n\}$ converge[2] to and therefore represent the same real number $s = 0.3$.

Although b-adic expansions are familiar objects, and especially so within the decimal system, it will be useful to review several additional facts concerning them. If the digits of a b-adic expansion are all zero for $i < -m$, it is called finite. Such zeros can be omitted,

$$x = \circ d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-m} = \circ \sum_{\nu=n}^{-m} d_\nu b^\nu. \tag{5.1.5}$$

The significance of b-adic expansions lies in the fact that all real numbers, in particular the irrational numbers, can be approximated by finite expansions. If we terminate a b-adic expansion with the digit d_{-m}, then according to (5.1.4) the resulting error is less than b^{-m}. By taking m large enough, the error can be made arbitrarily small.

Multiplication by b^m transforms each finite b-adic expansion (5.1.5) into a whole number. Thus each such expansion represents a rational number. On the other hand, not every rational number has a finite b-adic expansion. For instance, in the decimal system, we have $1/3 = 0.333\ldots$. We say that the decimal expansion of $1/3$ has a period of length 1. In general in a b-adic system the rational numbers are characterized by the fact that they have periodic b-adic expansions.

[2] A monotone increasing sequence of real numbers that is bounded above converges to its supremum.

If we define a b-adic expansion by selecting its digits randomly out of the set $\{0, 1, 2, \ldots, b-1\}$, the chance of obtaining a periodic expansion is negligible. This suggests that most of these expansions represent irrational numbers. In what follows we approximate real numbers by finite b-adic expansions, i.e., by a subset of the rational numbers. In the decimal system, for instance, the number $1/3$ is not a member of this approximating subset.

It is possible that the one-to-one correspondence between the numbers and the b-adic expansions permits the definition of the real numbers through their b-adic expansions. Two b-adic expansions would then be called equal if they are identical. We can also easily decide which of two numbers is the smaller. Consider the two expansions

$$c = \circ c_p c_{p-1} \cdots c_1 c_0 . c_{-1} c_{-2} c_{-3} \cdots \tag{5.1.6}$$

and

$$d = \circ d_q d_{q-1} \cdots d_1 d_0 . d_{-1} d_{-2} d_{-3} \cdots , \tag{5.1.7}$$

and let n be the largest index for which $c_n \neq d_n$. Without loss of generality we may assume $c_n < d_n$, i.e., $c_n + 1 \leq d_n$. Then $c < d$ since

$$c = \sum_{\nu=p}^{-\infty} c_\nu b^\nu = \sum_{\nu=p}^{n} c_\nu b^\nu + \sum_{\nu=n-1}^{-\infty} c_\nu b^\nu < \sum_{\nu=p}^{n} c_\nu b^\nu + b^n \leq \sum_{\nu=p}^{n} d_\nu b^\nu \leq d.$$

Negative b-adic expansions are treated correspondingly.

The definition of the operations for b-adic expansions, however, is an involved one. An infinite b-adic expansion cannot easily be added or multiplied in a simple way. The operations for infinite b-adic expansions must be defined in terms of a sequence of approximations.

For finite b-adic expansions of the form (5.1.5), the operations of addition, subtraction, multiplication, and division can easily be executed by well-known rules (at least for $b = 10$). These algorithms can easily be derived by using the representation (5.1.5). They reduce the operations with real numbers to operations with single digits. To prescribe these algorithms for addition, subtraction, and multiplication, tables for the addition, subtraction, and multiplication of all possible combinations of the digits $0, 1, 2, 3, \ldots, b-1$ suffice. Division can be executed as a repeated subtraction. Since these algorithms are very well known from computation with decimal numbers, we refrain from deriving them here.

However, direct operation with infinite b-adic expansions is not possible. Operations are defined by means of successive approximations. Since the error of such approximations can, in principle, be made arbitrarily small, they can be used as legitimate replacements for the operations for real numbers. We describe this process in some further detail. Once again let

$$c = \circ c_p c_{p-1} \cdots c_1 c_0 . c_{-1} c_{-2} c_{-3} \cdots \tag{5.1.8}$$

and

$$d = \circ d_q d_{q-1} \cdots d_1 d_0 . d_{-1} d_{-2} d_{-3} \cdots \tag{5.1.9}$$

be the b-adic expansions of two real numbers. We denote the nth partial sums by

$$\gamma_n = \circ c_p c_{p-1} \cdots c_1 c_0 . c_{-1} c_{-2} \cdots c_{-n} \tag{5.1.10}$$

and

$$\delta_n = \circ d_q d_{q-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n} . \tag{5.1.11}$$

To define $s := c \circ d$, $\circ \in \{+, -, \cdot, /\}$, in terms of the approximations γ_n and δ_n, we form the sequence $\{\sigma_n\}$, with

$$\sigma_n := \gamma_n \circ \delta_n, \ \circ \in \{+, -, \cdot, /\}.$$

If for division $\lim_{n\to\infty} \delta_n \neq 0$, the sequence $\{\sigma_n\}$ exists for sufficiently large n. The quantity σ_n can be calculated by the well-known algorithms for finite b-adic expansions. Appealing to a well-known theorem of analysis, we have

$$\lim_{n\to\infty} \sigma_n = \lim_{n\to\infty} (\gamma_n \circ \delta_n) = \lim_{n\to\infty} \gamma_n \circ \lim_{n\to\infty} \delta_n = c \circ d, \ \circ \in \{+, -, \cdot, /\},$$

and therefore $s = \lim_{n\to\infty} \sigma_n$.

However, the sequence $\{\sigma_n\}$ is not identical to the sequence $\{s_n\}$ of partial sums of s. We illustrate this with two examples.

Examples. 1. $\frac{1}{29} + \frac{2}{29} = \frac{3}{29}$ or $0.03448\ldots + 0.06896\ldots = 0.10344\ldots$

n	σ_n	s_n
-1	0.0	0.1
-2	0.09	0.10
-3	0.102	0.103
-4	0.1033	0.1034
$\vdots$	$\vdots$	$\vdots$

2. $\frac{1}{3} \cdot \frac{1}{3} = \frac{1}{9}$ or $0.33333\ldots \cdot 0.33333\ldots = 0.11111\ldots$

n	σ_n	s_n
-1	0.09	0.1
-2	0.1089	0.11
-3	0.110889	0.111
-4	0.11108889	0.1111
$\vdots$	$\vdots$	$\vdots$

Note that this example also shows that using double precision arithmetic does not guarantee single precision accuracy. Nevertheless, the real number s is well defined as the limit of the sequence $\{\sigma_n\}$. The irrational numbers can be defined as Cauchy sequences of rational numbers. Two sequences are equivalent if they have the same limit. In this sense $\{\sigma_n\}$ and $\{s_n\}$ both belong to the equivalence class that defines s.

The first part of this book is intended to provide an abstract setting for certain subsystems of real numbers. Their realization and implementation on computers is studied in the second part of the book. Floating-point systems are of particular interest.

5.2 Floating-Point Numbers and Roundings

We begin our discussion by recalling that, according to Theorem 5.2, every real number x can be uniquely represented by a b-adic expansion of the form

$$x = \circ d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots = \circ \sum_{\nu=n}^{-\infty} d_\nu b^\nu \tag{5.2.1}$$

with $\circ \in \{+,-\}$, $b \in \mathbb{N}$, $b > 1$, and

$$0 \le d_i \le b-1 \quad \text{for all } i = n(-1)-\infty, \tag{5.2.2}$$

$$d_i \le b-2 \quad \text{for infinitely many } i. \tag{5.2.3}$$

We may further assume that $d_n \neq 0$.

In (5.2.1) the (b-ary) point may be shifted to any other position if we compensate for this shifting by a multiplication with a corresponding power of b. If the point is shifted immediately to the left of the first nonzero digit d_n, the resulting expression is referred to as the *normalized b-adic representation* of the number x. Zero is the only real number that has no such representation. Conditions (5.2.2) and (5.2.3) assure the uniqueness of the normalized b-adic representation of all other real numbers. We employ the symbol $\mathbb{R}_b$ to denote all these numbers, including zero:

$$\begin{aligned} \mathbb{R}_b := \{0\} \cup \Big\{ x = \circ m \cdot b^e \ | \ & \circ \in \{+,-\}, b \in \mathbb{N}, b > 1, e \in \mathbb{Z}, \\ & m = \sum_{i=1}^{\infty} x_i b^{-i}, \\ & x_i \in \{0,1,\ldots,b-1\}, i = 1(1)\infty,\ x_1 \neq 0, \\ & x_i \le b-2 \text{ for infinitely many } i \Big\}. \end{aligned} \tag{5.2.4}$$

Here $\mathbb{Z}$ denotes the set of integers, b is called the *base* of the representation, $\circ$ the *sign* of x (sgn(x)), m the *mantissa* of x (mant(x)), and e the *exponent* of x (exp(x)).

In the literature m sometimes is also called the *fraction part* or the *significand*. Often the sign $\circ \in \{+, -\}$ is considered to be part of the mantissa.

In general the elements of $\mathbb{R}_b$ cannot be represented on a computer. Only truncated versions of these elements can be so represented. The following definition distinguishes a special subset of $\mathbb{R}_b$, which we use in the development to follow.

Definition 5.3. A real number is called a *normalized floating-point number* or simply a *floating-point number* if it is an element of the set $S = S(b, r, e1, e2)$:

$$S = S(b, r, e1, e2) := \left\{ x \in \mathbb{R}_b \mid m = \sum_{i=1}^{r} x_i b^{-i},\ e1 \leq e \leq e2,\ e1, e2 \in \mathbb{Z} \right\}.$$

In general $e1 < 0$ and $e2 > 0$. To have a unique representation of zero available in S, we assume additionally that sgn$(0) = +$, mant$(0) = 0.00 \ldots 0$, (r zeros after the (b-ary) point), and exp$(0) = 0$.

The set $S = S(b, r, e1, e2)$ is called a *floating-point system.* ■

In a floating-point system S the mantissa is restricted to a finite length r and the exponent is bounded by $e1$ and $e2$.

For a floating-point system $S = S(b, r, e1, e2)$ and its elements we have

(S3) $0, 1 \in S \wedge \bigwedge_{x \in S} -x \in S$

and

$$b^{-1} \leq |m| < 1.^{3} \tag{5.2.5}$$

Every element of S represents a rational number. The representation is unique. The number of elements in $S(b, r, e1, e2)$ is $2(b-1)b^{r-1}(e2 - e1 + 1) + 1$.

The greatest floating-point number in $S(b, r, e1, e2)$ is

$$B := +0.\underbrace{(b-1)(b-1)\cdots(b-1)}_{r-\text{digits}}\, b^{e2}.$$

The least number in S is $-B$. The nonzero floating-point numbers in S that are of least absolute value are

$$-0.100 \cdots 0 \cdot b^{e1}, \qquad +0.100 \cdots 0 \cdot b^{e1}. \tag{5.2.6}$$

The floating-point numbers in S are not uniformly distributed between the numbers in (5.2.6) and $-B$ and $+B$. The density decreases with increasing exponent. To illustrate this, we use an example of a floating-point system of 33 elements. This example is from [176] and corresponds to $b = 2$, $r = 3$, $e1 = -1$, and $e2 = 2$.

[3] In the binary number system the leading digit of a normalized floating-point number is always 1. Thus, in this particular case, the binary point is frequently placed directly after the leading 1 and the exponent is adjusted appropriately. In this case we have $1 \leq |m| < 2$.

Figure 3.6 depicts the floating-point system $S(2,3,-1,2)$. A floating-point system $S(b,r,e1,e2)$ is a discrete subset of $\mathbb{R}$.

To continue, we introduce the following notation:

$$\begin{aligned}
&\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty\} \cup \{+\infty\},\\
&\overline{S} := \overline{S}(b,r,e1,e2) := S(b,r,e1,e2) \cup \{-\infty\} \cup \{+\infty\},\\
&S' := [-B,+B] \subset \mathbb{R}.
\end{aligned}$$

Then $\overline{S}(b,r,e1,e2)$ is a screen of $\overline{\mathbb{R}}$. Further $S(b,r,e1,e2)$ is a screen of S'. All of these screens are symmetric, i.e., they have the property (S3).

We are now going to consider roundings from $\overline{\mathbb{R}}$ into $\overline{S}(b,r,e1,e2)$. Similar roundings could also be defined from S' into $S(b,r,e1,e2)$.

A rounding $\square : \overline{\mathbb{R}} \to \overline{S}$ is defined by the property

(R1) $\bigwedge\limits_{x\in\overline{S}} \square x = x.$

Other important properties of roundings are:

(R2) $\bigwedge\limits_{x,y\in\overline{\mathbb{R}}} (x \le y \Rightarrow \square x \le \square y)$ (monotonicity),

(R3) $\bigwedge\limits_{x\in\overline{\mathbb{R}}} \square x \le x$ or $\bigwedge\limits_{x\in\overline{\mathbb{R}}} x \le \square x$ (directed),

(R4) $\bigwedge\limits_{x\in\overline{\mathbb{R}}} \square(-x) = -\square x$ (antisymmetry).

In the following development we are especially concerned with the monotone directed roundings $\triangledown$ and $\vartriangle$ as well as the monotone and antisymmetric roundings. We shall consider the following roundings:

$$\bigwedge_{x\in\overline{\mathbb{R}}} \triangledown x := \max\{y \in \overline{S} \mid y \le x\}, \quad \text{monotone downwardly directed rounding,}$$

$$\bigwedge_{x\in\overline{\mathbb{R}}} \vartriangle x := \min\{y \in \overline{S} \mid y \ge x\}, \quad \text{monotone upwardly directed rounding,}$$

$$\bigwedge_{x\in\overline{\mathbb{R}}} \square_b x := \begin{cases} \triangledown x & \text{if } x \ge 0 \\ \vartriangle x & \text{if } x < 0 \end{cases}, \quad \text{monotone rounding towards zero,}$$

$$\bigwedge_{x\in\overline{\mathbb{R}}} \square_o x := \begin{cases} \vartriangle x & \text{if } x \ge 0 \\ \triangledown x & \text{if } x < 0 \end{cases}, \quad \text{monotone rounding away from zero.}$$

Using the notation

$$\bigwedge_{x\in\overline{\mathbb{R}}, x\ge 0} s_\mu(x) := \triangledown x + (\vartriangle x - \triangledown x)\cdot \mu/b, \quad \mu = 1(1)b-1,$$

we define the roundings $\square_\mu : \overline{\mathbb{R}} \to \overline{S}$, $\mu = 1(1)b-1$, which are in common use:

$$\bigwedge_{x\in[0,b^{e1-1})} \square_\mu x := 0,$$

$$\bigwedge_{x\geq b^{e1-1}} \square_\mu x := \begin{cases} \triangledown x & \text{for } x \in [\triangledown x, s_\mu(x)) \\ \triangle x & \text{for } x \in [s_\mu(x), \triangle x] \end{cases},$$

$$\bigwedge_{x<0} \square_\mu x := -\square_\mu(-x).$$

If b is an even number, $\bigcirc := \square_{b/2}$ denotes a rounding to the nearest floating-point number. Figure 5.1 contains a diagram of this rounding for the floating-point system $S(2,3,-1,2)$ illustrated in Figure 3.6.

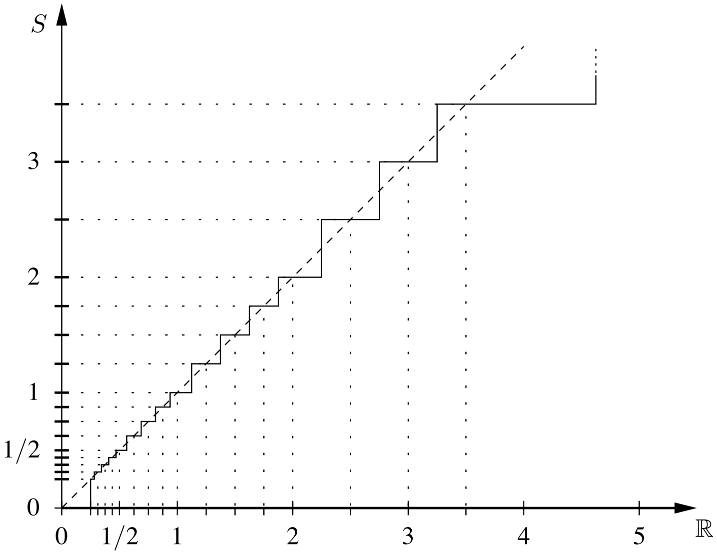


Figure 5.1. Rounding to the nearest floating-point number.

The roundings listed above have many familiar properties. For instance, we have

$$\triangledown x = -\triangle(-x) \wedge \triangle x = -\triangledown(-x), \tag{5.2.7}$$

$$\square_b x = \operatorname{sgn}(x) \cdot \triangledown|x| \wedge \square_0 x = \operatorname{sgn}(x) \cdot \triangle|x|. \tag{5.2.8}$$

All of these roundings are monotone (R2). The roundings $\square_\mu, \mu = 0(1)b$ are also antisymmetric (R4).

We recall that in a linearly ordered set every monotone rounding and therefore, in particular, all the roundings $\square_\mu$, $\mu = 0(1)b$, can be expressed in terms of the monotone downwardly or upwardly directed rounding $\triangledown$ or $\triangle$ (Section 3.5). Since

$\triangle$ can be expressed in terms of $\triangledown$ and vice versa, we give an explicit description of the rounding $\triangledown : \overline{R} \to \overline{S}$ only. In the following description of $\triangledown$ we use the abbreviation $B = 0.(b-1)(b-1)\ldots(b-1) \cdot b^{e2}$ for the greatest positive floating-point number. Then we obtain for $\triangledown x$:

$$\triangledown x = \begin{cases} +\infty & \text{for } x = +\infty, \\ +B & \text{for } +B \le x < +\infty, \\ +0.x_1x_2\ldots x_r \cdot b^e & \text{for } b^{e1-1} \le x < +B, \\ +0.000\ldots 0 \cdot b^{e1} & \text{for } 0 \le x < b^{e1-1}, \\ -0.100\ldots 0 \cdot b^{e1} & \text{for } -b^{e1-1} \le x < 0, \\ -0.x_1x_2\ldots x_r \cdot b^e & \text{for } -B \le x < -b^{e1-1} \\ & \wedge\ x_{r+i} = 0 \text{ for all } i \ge 1, \\ -0.100\ldots 0 \cdot b^{e+1} & \text{for } -B \le x < -b^{e1-1} \\ & \wedge\ x_i = b-1 \text{ for all } i = 1(1)r \\ & \wedge\ x_{r+i} \ne 0 \text{ for any } i \ge 1, \\ -(0.x_1x_2\ldots x_r + b^{-r}) \cdot b^e & \text{for } -B \le x < -b^{e1-1} \\ & \wedge\ x_i \ne b-1 \text{ for any } i \in \{1,2,\ldots,r\} \\ & \wedge\ x_{r+i} \ne 0 \text{ for any } i \ge 1, \\ -\infty & \text{for } -\infty \le x < -B. \end{cases}$$

Using the function $[x]$ (the greatest integer less than or equal to x) the description of $\triangledown x$ can be shortened:

$$\triangledown x = \begin{cases} +\infty & \text{for } x = +\infty, \\ +B & \text{for } +B \le x < +\infty, \\ [m \cdot b^r] \cdot b^{e-r} & \text{for } b^{e1-1} \le |x| \le +B, \\ +0.000\ldots 0 \cdot b^{e1} & \text{for } 0 \le x < b^{e1-1}, \\ -0.100\ldots 0 \cdot b^{e1} & \text{for } -b^{e1-1} \le x < 0, \\ -\infty & \text{for } -\infty \le x < -B. \end{cases}$$

The more detailed description of $\triangledown x$ above shows that a normalization may still be necessary.

A few additional but very similar cases occur if for $e < e1$ the exponent e is set to $e1$ and unnormalized mantissas are permitted.

Thus the implementation of the rounding $\triangledown$ on a computer is simplified if the function $[x]$ is available. In the algorithms for the implementation of computer arithmetic, which we describe in Chapter 6, we shall use this prescription of $\triangledown x$.

In these representations for $\triangledown x$ we have assumed that the mantissa of a floating-point number is represented by the so-called signed-magnitude representation. For real numbers $x \geq 0$ the rounded value ∇x then is obtained by truncation of x after the r^{th} digit of the normalized mantissa m of x. If we denote this process by $t(x)$ (truncation), we have

$$\nabla x = t(x) \text{ for } x \geq 0.$$

This is very easy to implement. Truncation can also be used to perform the rounding ∇x of negative numbers $x < 0$ if negative numbers are represented by their b-complement. Then the rounded value ∇x can be obtained by truncation of the b-complement $x + a$ of x via the process:

$$\nabla x = t(x+a) - a \quad \text{for} \quad x < 0 \tag{5.2.9}$$

with a suitable a. See Figure 5.2.

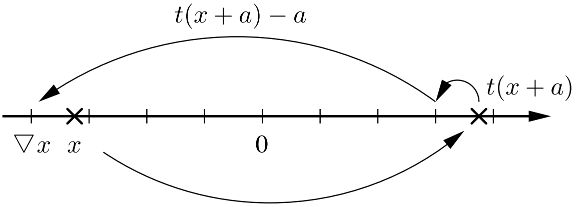


Figure 5.2. Execution of the rounding ∇x for b-complement representation of negative numbers $x < 0$.

Example 5.4. We assume that the decimal number system is used, and that the mantissa has three decimal digits. Then we obtain for the positive real number $x = 0.354672 \cdot 10^3 \in \mathbb{R}$:

$$\nabla x = t(x) = 0.354 \cdot 10^3.$$

For the negative number $x = -0.354672 \cdot 10^3$ we obtain obviously

$$\nabla x = -0.355 \cdot 10^3.$$

This value is obtained by application of (5.2.9) with $a = 1.00\ldots0 \cdot 10^3$:

$$\begin{aligned} x + a &= 0.645328 \cdot 10^3, \\ t(x+a) &= 0.645 \cdot 10^3, \\ \nabla x = t(x+a) - a &= -0.355 \cdot 10^3. \end{aligned}$$

Here the simple b-complement has been taken twice. In between the function $t(x)$ was applied which also is simple. These three steps are particularly simple if the binary number system is used.

It is interesting that in the case of the $(b-1)$-complement representation of negative numbers the monotone downwardly directed rounding ∇x cannot be executed by the function $t(x)$. This representation is isomorphic to the sign-magnitude representation.

The roundings of $\overline{\mathbb{R}} \to \overline{S}(b, r, e1, e2)$, which we have been discussing so far, are not the only ones that are used in computer arithmetic or in numerical analysis. Typically for error analyses of numerical processes, it is usually assumed that the rounding has the property

$$\square x = x(1 - \epsilon) \text{ with } |\epsilon| \le \epsilon^* = \text{const.} \tag{5.2.10}$$

Figure 5.3 illustrates a rounding with this property. The graph corresponding to $\square$ has to stay within a cone around the identity mapping. It is not necessarily a monotone nor even an antisymmetric function.

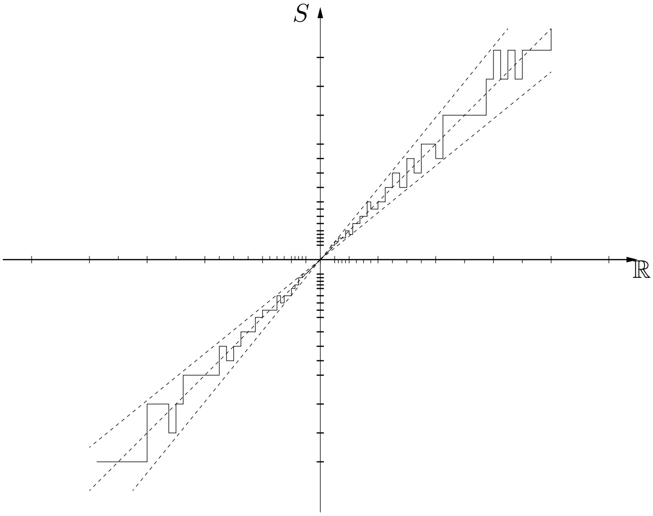


Figure 5.3. A special rounding.

Because of (R1) every rounding corresponds to a step function that crosses the graph of the identity mapping for all values in $S(b, r, e1, e2)$. This implies, in particular, that within an interval around zero the rounding cannot be described by (5.2.10). Compare the Figures 5.1 and 5.3.

Although equation (5.2.10) does not necessarily describe a monotone mapping, we show in the following theorem that within the interval $b^{e1-1} \le |x| \le B$, every monotone rounding has this property (5.2.10).

Theorem 5.5. *Let $\overline{S} = \overline{S}(b, r, e1, e2)$ be a floating-point system and $\square : \mathbb{R} \to \overline{S}$ any monotone rounding. Further, let $\delta(x) := x - \square x$ be the rounding error and $\epsilon_1 := \delta(x)/x$ and $\epsilon_2 := \delta(x)/\square x$ the relative rounding errors. Then*

$$\bigwedge_{x \in \mathbb{R}} (b^{e1-1} \le |x| \le B \Rightarrow \square x = x(1 - \epsilon_1) \textit{ with } |\epsilon_1| \le \epsilon^*$$
$$\wedge\, x = \square x(1 - \epsilon_2) \textit{ with } |\epsilon_2| \le \epsilon^*$$
$$\wedge\, |x - \square x| < \epsilon^* |x| \wedge |x - \square x| < \epsilon^* |\square x|).$$

Here ϵ^* is independent of x, and

$$\epsilon^* = \begin{cases} \frac{1}{2}b^{1-r} & \textit{for } \square = \bigcirc \\ b^{1-r} & \textit{for } \square \neq \bigcirc \end{cases}.$$

Here $\bigcirc$ denotes the rounding to the nearest floating-point number of $\overline{S}$, and $B = 0.(b-1)(b-1)\dots(b-1)\cdot b^{e2}$.

Proof. We consider the formula that includes ϵ_1. The formula with ϵ_2 can be dealt with analogously. We have

$$x = \square x + \delta(x) = \square x + \epsilon_1 x \Rightarrow \square x = x(1-\epsilon_1).$$

With $x = \circ m \cdot b^e$ we obtain using (5.2.5) that $b^{-1} \cdot b^e \le |x| < b^e$.
Since $|\delta(x)| \le b^{-r} \cdot b^e$, we get

$$|\epsilon_1| = |\delta(x)|/|x| \le (b^{-r} \cdot b^e)/(b^{-1} \cdot b^e) = b^{1-r}.$$

It is easy to see that actually $|\epsilon_1| < b^{1-r}$: If $x = b^{-1} \cdot b^e$, then x is a screen point, and because of (R1), $\delta(x) = 0$. If $\delta(x) = b^{-r} \cdot b^e$, then $|x| > b^{-1} \cdot b^e$.

The case $\square = \bigcirc$ can be proved analogously. ■

Theorem 5.5 furnishes error bounds for a rounding only for $|x| \in [b^{e1-1}, B]$. For $|x| < b^{e1-1}$, the relative error ϵ_1 can – depending on the definition of the rounding function – be identically unity or even tend to infinity. For $|x| > B$, the relative error ϵ_1 tends to unity as x goes to infinity.

Figure 5.4 shows the relative error of the rounding illustrated in Figure 5.1. For $b^{e1-1} \le |x| \le B$, we have $|\epsilon_1| < \epsilon^* = \frac{1}{2} \cdot b^{1-r}$. Within this region the relative error can only be made smaller by enlarging the number r of digits in the mantissa. Figure 5.4 also shows that it is difficult (resp. impossible) to approximate real numbers well on the computer when they lie to the left (resp. right) of this region. For $|x| < b^{e1-1}$, the relative error is unity. For $|x| \ge B$, it tends to unity as x goes to infinity. The only way to decrease the size of these outer regions is to decrease $e1$ and to increase $e2$. This requires enlarging the range used for the representation of the exponent.

We summarize these results in the following two rules:

(a) To approximate better within the region $b^{e1-1} \le |x| \le B$, more digits must be used to represent the mantissa.

(b) To enlarge the region $b^{e1-1} \le |x| \le B$ of good approximation, more digits must be used to represent the exponent.

If a number $|x| \in (0, b^{e1-1})$ occurs in a virtual sense on a computer one speaks of *exponent underflow* or simply of *underflow*. Correspondingly, one speaks of *exponent overflow* or simply of *overflow* if $|x| > B$. With these concepts, the condition $b^{e1-1} \le |x| \le B$, which appears in Theorem 5.5, is usually expressed by saying: *If no underflow and no overflow occurs, then ....*

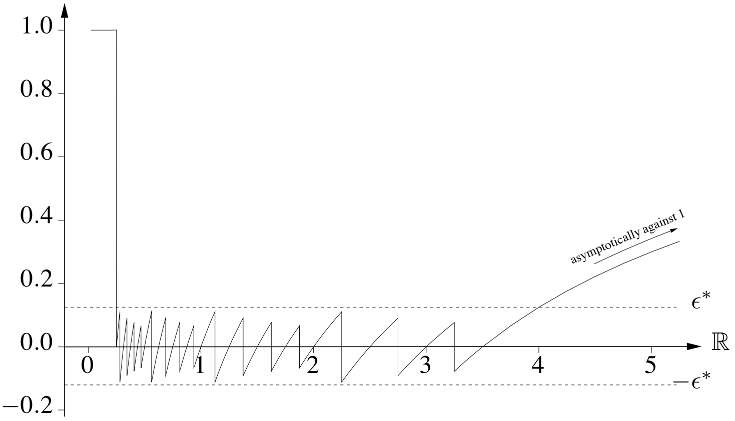


Figure 5.4. Relative rounding error.

5.3 Floating-Point Operations

We now turn to consideration of arithmetic operations in floating-point systems, followed by treatment of errors of such operations.

In Chapters 3 and 4 we established that the arithmetic in the subsets under D and S and in all rows of Figure 1 can be defined by semimorphisms. We recall that if M is any ringoid or vectoid and $N \subseteq M$ a symmetric lower or upper screen (or both) of M, then a semimorphism $\square$ is defined by the following formulas:

(R1) $\bigwedge_{x \in N} \square x = x,$

(R2) $\bigwedge_{x,y \in M} (x \leq y \Rightarrow \square x \leq \square y),$

(R4) $\bigwedge_{x \in M} \square(-x) = -\square x,$

(RG) $\bigwedge_{x,y \in N} (x \mathbin{\square\!\!\!\!\!\circ}\, y := \square(x \circ y)),$ for $\circ \in \{+, -, \cdot, /\}$ with $y \neq 0$ for $\circ = /$.

In general the operations defined by (RG) are not associative, nor are addition and multiplication distributive. We show this by means of a few simple examples.

Examples. Consider the floating-point system $S = S(10, 1, -1, 1)$. Let $x, y, z \in S$.

1. Consider the operation $x \mathbin{\triangle\!\!\!\!\!+} y := \triangle(x + y)$. For $x := 0.6$, $y := 0.5$, $z = 0.4$, we obtain:

$$(x \mathbin{\triangle\!\!\!\!\!+} y) \mathbin{\triangle\!\!\!\!\!+} z = (\triangle 1.1) \mathbin{\triangle\!\!\!\!\!+} 0.4 = 0.2 \cdot 10 \mathbin{\triangle\!\!\!\!\!+} 0.4 = \triangle(0.24 \cdot 10) = 0.3 \cdot 10,$$

$$x \mathbin{\triangle\!\!\!\!\!+} (y \mathbin{\triangle\!\!\!\!\!+} z) = 0.6 \mathbin{\triangle\!\!\!\!\!+} 0.9 = \triangle 1.5 = 0.2 \cdot 10,$$

i.e., $(x \mathbin{\triangle\!\!\!\!\!+} y) \mathbin{\triangle\!\!\!\!\!+} z \neq x \mathbin{\triangle\!\!\!\!\!+} (y \mathbin{\triangle\!\!\!\!\!+} z)$.

2. Consider the operation $x \triangledown y := \triangledown(x \cdot y)$. For $x := 0.3$, $y := 0.4$, $z := 0.4$, we obtain:

$$(x \triangledown y) \triangledown z = (\triangledown 0.12) \triangledown 0.4 = 0.1 \triangledown 0.4 = 0.4 \cdot 10^{-1},$$
$$x \triangledown (y \triangledown z) = 0.3 \triangledown (\triangledown 0.16) = 0.3 \triangledown 0.1 = 0.3 \cdot 10^{-1},$$

i.e., $(x \triangledown y) \triangledown z \neq x \triangledown (y \triangledown z)$.

The same effect can be shown for nondirected roundings. For instance, let $\square\colon \overline{\mathbb{R}} \to \overline{S}$ be the rounding to the nearest number of the screen with the property that the midpoint between two neighboring screen numbers is rounded upwardly. Then:

3. With $x := 0.7$, $y := 0.7$, $z := 0.9$, we obtain for the operation $x \boxplus y := \square(x + y)$

$$(x \boxplus y) \boxplus z = (\square 1.4) \boxplus 0.9 = 0.1 \cdot 10 \boxplus 0.9 = \square 0.19 \cdot 10 = 0.2 \cdot 10,$$
$$x \boxplus (y \boxplus z) = 0.7 \boxplus (\square 1.6) = 0.7 \boxplus 0.2 \cdot 10 = \square 0.27 \cdot 10 = 0.3 \cdot 10,$$

i.e., $(x \boxplus y) \boxplus z \neq x \boxplus (y \boxplus z)$.

4. With $x := 0.3$, $y := 0.4$, $z := 0.4$, we obtain for the operation $x \boxdot y := \square(x \cdot y)$

$$(x \boxdot y) \boxdot z = (\square 0.12) \boxdot 0.4 = 0.1 \boxdot 0.4 = \square 0.04 = 0.4 \cdot 10^{-1},$$
$$x \boxdot (y \boxdot z) = 0.3 \boxdot (\square 0.16) = 0.3 \boxdot 0.2 = \square 0.06 = 0.6 \cdot 10^{-1},$$

i.e., $(x \boxdot y) \boxdot z \neq x \boxdot (y \boxdot z)$.

The distributive law doesn't hold either:

5. For $x := 0.3$, $y := 0.7$, $z := 0.9$, we obtain

$$x \boxdot (y \boxplus z) = 0.3 \boxdot (\square 1.6) = 0.3 \boxdot 0.2 \cdot 10 = 0.6,$$
$$x \boxdot y \boxplus x \boxdot z = (\square 0.21) \boxplus (\square 0.27) = 0.2 \boxplus 0.3 = 0.5,$$

i.e., $x \boxdot (y \boxplus z) \neq x \boxdot y \boxplus x \boxdot z$.

If the operations in the subset $\overline{S}$ of $\overline{\mathbb{R}}$ are defined by (RG), employing a monotone rounding, then error bounds may be obtained. These are given in the following theorem whose proof follows directly from Theorem 5.5.

Theorem 5.6. *Let $\overline{S} = \overline{S}(b, r, e1, e2)$ be a floating-point system, $\square : \overline{\mathbb{R}} \to \overline{S}$ be a monotone rounding, and let operations in $\overline{S}$ be defined by*

(RG) $\bigwedge\limits_{x,y \in \overline{S}} x \boxed{\circ} y := \square(x \circ y)$ *for all* $\circ \in \{+, -, \cdot, /\}$ *with* $y \neq 0$ *for* $\circ = /$.

If $\delta := x \circ y - x \boxdot y$ denotes the absolute error and $\epsilon_1 := \delta/(x \circ y)$ and $\epsilon_2 := \delta/(x \boxdot y)$ relative errors, then for all operations $\circ \in \{+, -, \cdot, /\}$, the following error relations hold:

$$\bigwedge_{x,y \in \overline{S}} \Big(b^{e1-1} \leq |x \circ y| \leq B \Rightarrow x \boxdot y = (x \circ y)(1 - \epsilon_1) \text{ with } |\epsilon_1| < \epsilon^* \\ \wedge\ x \circ y = (x \boxdot y)(1 - \epsilon_2) \text{ with } |\epsilon_2| < \epsilon^* \\ \wedge\ |(x \circ y) - (x \boxdot y)| < \epsilon^* |x \circ y| \\ \wedge\ |(x \circ y) - (x \boxdot y)| < \epsilon^* |x \boxdot y|\Big).$$

Here ϵ^* and B are the constants defined in Theorem 5.5. Division is not defined if $y = 0$. ■

In Theorems 5.5 and 5.6, as well as in several of the following theorems, the condition on the left-hand side of the ⇒ sign means that neither underflow nor overflow occurs. The error estimates in these theorems are valid in this case.

Note that the relations on the right-hand side of the ⇒ sign in Theorem 5.6 are precise quantitative statements about the errors in arithmetic. Contrast this with the usual error estimates in numerical analysis. Those estimates have the same form as the bounds here, but are only qualitative statements.

Theorem 5.6 is valid for all roundings $\square \in \{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$ and in particular for all semimorphisms.

Then, as a result of the definition of the arithmetic by semimorphism in rows 2, 3, 7, 8 and 9 of Figure 1, similar error relations can also be derived. As an example, we consider the matrix operations in row 3. The results are summarized in the following theorem. (See also Theorem 3.9).

Theorem 5.7. *Let $\overline{S} = \overline{S}(b, r, e1, e2)$ be a floating-point system, $\square : \overline{\mathbb{R}} \to \overline{S}$ a monotone rounding, and $M_n\mathbb{R}$ the set of $n \times n$ matrices over $\mathbb{R}$. We define a rounding $\square : M_n\overline{\mathbb{R}} \to M_n\overline{S}$ by*

$$\bigwedge_{X=(x_{ij}) \in M_n\overline{\mathbb{R}}} \square X := (\square x_{ij})$$

and operations $\boxdot$ by

(RG) $\displaystyle\bigwedge_{X,Y \in M_n\overline{S}} X \boxdot Y := \square(X \circ Y),\ \circ \in \{+, \cdot\}.$

Then

$$\bigwedge_{X=(x_{ij}) \in M_n\overline{\mathbb{R}}} \Big(b^{e1-1} \leq |x_{ij}| \leq B \Rightarrow \square X = \big(x_{ij}(1 - \epsilon^1_{ij})\big) \text{ with } |\epsilon^1_{ij}| < \epsilon^* \\ \wedge\ X = (\square x_{ij}(1 - \epsilon^2_{ij})) \text{ with } |\epsilon^2_{ij}| < \epsilon^* \\ \wedge\ |X - (\square X)| < \epsilon^* |X| \\ \wedge\ |X - (\square X)| < \epsilon^* |\square X|\Big).$$

With $Z := (z_{ij}) := X \circ Y$, $\circ \in \{+, \cdot\}$, *we obtain the following error relations for the operations:*

$$\begin{aligned}\bigwedge_{X,Y \in M_n\overline{S}} (b^{e1-1} \leq |z_{ij}| \leq B \Rightarrow\ & X \boxdot Y = (z_{ij}(1-\epsilon^1_{ij})) \ \textit{with}\ |\epsilon^1_{ij}| < \epsilon^* \\ &\wedge\ X \circ Y = (X \boxdot Y)(1-\epsilon^2_{ij}) \ \textit{with}\ |\epsilon^2_{ij}| < \epsilon^* \\ &\wedge\ |X \circ Y - (X \boxdot Y)| < \epsilon^* |X \circ Y| \\ &\wedge\ |X \circ Y - (X \boxdot Y)| < \epsilon^* |X \boxdot Y|).\end{aligned}$$

Here all absolute values are to be taken componentwise. ϵ^* *and* B *are defined as in Theorem 5.5.* ■

As before, we may note now that Theorem 5.7 holds for all roundings $\square \in \{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$ and in particular for all semimorphisms. It is clear that corresponding formulas hold for the matrix-vector multiplication. Corresponding theorems can also be derived for all complex operations and for the complex matrix and vector operations. All these results lead to error relations that are especially simple by comparison with error estimates derived on the basis of the conventional definition of arithmetic. To illustrate this, we derive the corresponding error relations for matrix multiplication when the latter is defined by the conventional method.

The essence of the point to be made is illustrated by a comparison of the error relations for the scalar product defined by the conventional method and by semimorphism. Actually, the error relations for matrix multiplication derived in Theorem 5.7 deal with such scalar products if the former are written component-wise. If $x = (x_i)$ and $y = (y_i)$ with $x_i, y_i \in \overline{S}$, $i = 1(1)n$, the definition of the scalar product by semimorphism simply leads to the error relations

$$x \boxdot y := \square \left(\sum_{i=1}^{n} x_i y_i \right) = (1-\epsilon_1) \sum_{i=1}^{n} x_i y_i = (x \cdot y)(1-\epsilon_1) \quad \text{with } |\epsilon_1| < \epsilon^*, \tag{5.3.1}$$

$$x \cdot y = \sum_{i=1}^{n} x_i y_i = (1-\epsilon_2) \cdot \square \left(\sum_{i=1}^{n} x_i y_i \right) = (x \boxdot y)(1-\epsilon_2) \quad \text{with } |\epsilon_2| < \epsilon^*, \tag{5.3.2}$$

and to the error bounds

$$\left| \sum_{i=1}^{n} x_i y_i - \square \left(\sum_{i=1}^{n} x_i y_i \right) \right| < \epsilon^* \left| \sum_{i=1}^{n} x_i y_i \right|, \tag{5.3.3}$$

$$\left| \sum_{i=1}^{n} x_i y_i - \square \left(\sum_{i=1}^{n} x_i y_i \right) \right| < \epsilon^* \left| \square \left(\sum_{i=1}^{n} x_i y_i \right) \right|, \tag{5.3.4}$$

provided that neither underflow nor overflow occurs. The relations hold for all monotone roundings and in particular for all $\square \in \{\bigtriangledown, \bigtriangleup, \square_\mu, \mu = 0(1)b\}$. ϵ^* is defined as in Theorem 5.5. The error estimates (5.3.3) and (5.3.4) also can be written in the simpler form $|x \cdot y - x \boxdot y| < \epsilon^* |x \cdot y|$ and $|x \cdot y - x \boxdot y| < \epsilon^* |x \boxdot y|$.

If the floating-point matrix product is defined by the conventional method, the scalar products are defined by the formula

$$\boxed{\sum_{i=1}^{n}} x_i \boxdot y_i = (x_1 \boxdot y_1) \boxplus (x_2 \boxdot y_2) \boxplus \ldots \boxplus (x_n \boxdot y_n).$$

If we apply Theorem 5.6 to this relation, we obtain the following expression when $n = 4$:

$$\begin{aligned}
&(x_1 \boxdot y_1) \boxplus (x_2 \boxdot y_2) \boxplus (x_3 \boxdot y_3) \boxplus (x_4 \boxdot y_4) \\
&\quad = \Big(\Big(\big(x_1 y_1 (1 - \epsilon_1) + x_2 y_2 (1 - \epsilon_2)\big)(1 - \epsilon_5) + x_3 y_3 (1 - \epsilon_3)\Big)(1 - \epsilon_6) \\
&\quad\quad + x_4 y_4 (1 - \epsilon_4)\Big)(1 - \epsilon_7) \\
&\quad = x_1 y_1 (1 - \epsilon_1)(1 - \epsilon_5)(1 - \epsilon_6)(1 - \epsilon_7) \\
&\quad\quad + x_2 y_2 (1 - \epsilon_2)(1 - \epsilon_5)(1 - \epsilon_6)(1 - \epsilon_7) \\
&\quad\quad + x_3 y_3 (1 - \epsilon_3)(1 - \epsilon_6)(1 - \epsilon_7) \\
&\quad\quad + x_4 y_4 (1 - \epsilon_4)(1 - \epsilon_7).
\end{aligned} \tag{5.3.5}$$

Here $|\epsilon_i| < \epsilon^*$ for all $i = 1(1)7$.

The many epsilons occurring in this expression make it more complicated than the simple formulas (5.3.1) and (5.3.2), which occur in Theorem 5.7. We may expect that an error analysis of numerical algorithms is more troublesome if it is based on (5.3.5) instead of (5.3.1) and (5.3.2). Moreover, in general (5.3.1) and (5.3.2) are more accurate than (5.3.5).

We now estimate the error in (5.3.5) as a means of simplifying it and generalizing it simultaneously. We shall use Bernoulli's inequality

$$1 + nx \leq (1 + x)^n \text{ for all } n \in \mathbb{N} \text{ and all } x \geq -1, \tag{5.3.6}$$

and Bernoulli's formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{1 \cdot 2 \cdot 3 \cdots k}. \tag{5.3.7}$$

We have to estimate the expression:

$$\left| \sum_{i=1}^{n} x_i y_i - \boxed{\sum_{i=1}^{n}} x_i \boxdot y_i \right|.$$

We prove the following simple lemma.

Lemma 5.8. *If* $\epsilon^* \in \mathbb{R}$, $n \in \mathbb{N}$ *and* $0 \le n\epsilon^* < 1$ *then*

$$\bigwedge_{i=1(1)n} |\epsilon_i| < \epsilon^* \quad \Rightarrow$$

$$\prod_{i=1}^{n}(1-\epsilon_i) \in \left[1-n\epsilon^*, \frac{1}{1-n\epsilon^*}\right] \subseteq \left[1-\frac{n\epsilon^*}{1-n\epsilon^*}, 1+\frac{n\epsilon^*}{1-n\epsilon^*}\right].$$

Proof. By Bernoulli's inequality (5.3.6) we have

$$1-\frac{n\epsilon^*}{1-n\epsilon^*} \le 1-n\epsilon^* \le (1-\epsilon^*)^n.$$

On the other hand we obtain

$$\begin{aligned}
(1-\epsilon^*)^n \;&\le\; \prod_{i=1}^{n}(1-\epsilon_i) \le (1+\epsilon^*)^n = \sum_{k=0}^{n}\binom{n}{k}\left(\epsilon^*\right)^k \\
&\underset{(5.3.7)}{\le} \sum_{k=0}^{n} n^k\left(\epsilon^*\right)^k \le \sum_{k=0}^{\infty}\left(n\epsilon^*\right)^k = \frac{1}{1-n\epsilon^*} \\
&= \;1+\frac{n\epsilon^*}{1-n\epsilon^*}.
\end{aligned}$$

■

We now use the abbreviation

$$\eta := \frac{n\epsilon^*}{1-n\epsilon^*},$$

and distinguish two index sets:

$$I_+ := \{i \mid x_i y_i \ge 0\}, \quad I_- := \{i \mid x_i y_i < 0\}.$$

Generalizing (5.3.5) we obtain

$$\boxed{\sum}_{i=1}^{n} x_i \boxdot y_i = \boxed{\sum}_{i=1}^{n} x_i y_i (1-\epsilon_{i1}) = \sum_{i=1}^{n} x_i y_i \prod_{k=1}^{n}(1-\epsilon_{ik}). \tag{5.3.8}$$

Here the formula on the right-hand side has been completed by factors $(1-\epsilon_{ik})$ to obtain n factors for every summand where possibly $\epsilon_{ik} = 0$ for some k.

We are now going to derive upper and lower bounds for (5.3.8).

Upper bound:

$$\boxed{\sum}_{i=1}^{n} x_i \boxdot y_i = \sum_{i\in I_+} x_i y_i \prod_{k=1}^{n}(1-\epsilon_{ik}) - \sum_{i\in I_-} |x_i y_i| \prod_{k=1}^{n}(1-\epsilon_{ik}).$$

Applying Lemma 5.8 and the abbreviation η we now obtain

$$\begin{aligned}\overset{n}{\underset{i=1}{\boxed{\sum}}} x_i \boxdot y_i &\leq \sum_{i \in I_+} x_i y_i (1+\eta) - \sum_{i \in I_-} |x_i y_i| (1-\eta) \\ &= \sum_{i=1}^{n} x_i y_i + \eta \sum_{i \in I_+} x_i y_i + \eta \sum_{i \in I_-} |x_i y_i| \\ &= \sum_{i=1}^{n} x_i y_i + \eta \sum_{i=1}^{n} |x_i y_i|.\end{aligned}$$

Lower bound:

$$\begin{aligned}\overset{n}{\underset{i=1}{\boxed{\sum}}} x_i \boxdot y_i &\geq \sum_{i \in I_+} x_i y_i (1-\eta) - \sum_{i \in I_-} |x_i y_i| (1+\eta) \\ &= \sum_{i=1}^{n} x_i y_i - \eta \sum_{i \in I_+} x_i y_i - \eta \sum_{i \in I_-} |x_i y_i| \\ &= \sum_{i=1}^{n} x_i y_i - \eta \sum_{i=1}^{n} |x_i y_i|.\end{aligned}$$

Thus we obtain for the absolute error of the conventional computation of the scalar product the estimate:

$$\left| \sum_{i=1}^{n} x_i y_i - \overset{n}{\underset{i=1}{\boxed{\sum}}} x_i \boxdot y_i \right| \leq \eta \sum_{i=1}^{n} |x_i y_i| = \frac{n\epsilon^*}{1-n\epsilon^*} \sum_{i=1}^{n} |x_i y_i|. \tag{5.3.9}$$

This error estimate holds as long as $0 \leq n\epsilon^* < 1$, i.e., $\epsilon^* < 1/n$. For the double precision binary data format of the IEEE arithmetic standard 754, $\epsilon^* = 2^{-53}$. Thus $n < 9 \cdot 10^{15}$. This are very large vector length indeed.

Since $n\epsilon^* < 1$ the factor $n\epsilon^*/(1-n\epsilon^*)$ on the right-hand side of (5.3.9) is greater than $n\epsilon^*$. This, of course, is larger than the factor ϵ^* in (5.3.3) obtained for the computation of the scalar product defined by semimorphism.

By another error estimate the absolute error can be shown to be bounded by

$$\left| \sum_{i=1}^{n} x_i y_i - \overset{n}{\underset{i=1}{\boxed{\sum}}} x_i \boxdot y_i \right| \leq (n+0.1)\epsilon^* \sum_{i=1}^{n} |x_i y_i|. \tag{5.3.10}$$

In any case these bounds for the relative error are at least n times as large as those we obtained in (5.3.3) for the computation of the scalar product defined by semimorphism.

For matrix multiplication defined by the conventional method, error bounds may also be obtained by the process discussed above. Letting $X := (x_{ij})$, $Y := (y_{ij})$, $Z := (z_{ij}) := X \cdot Y$, these bounds are

$$\bigwedge_{X,Y \in M_n S} \left(\bigwedge_{i,k} b^{e1-1} \leq |z_{ik}| \leq B \Rightarrow |X \cdot Y - X \boxdot Y| \leq \frac{n\epsilon^*}{1 - n\epsilon^*} \left(\sum_{j=1}^{n} |x_{ij} y_{jk}| \right) \right)$$

if $n\epsilon^* < 1$. As before, these bounds are more than n times as large as those obtained if the matrix multiplication is defined by semimorphism. Above all the error formulas obtained for the semimorphism are simpler. The two properties of being more accurate and simpler are reproduced in the error analysis of many algorithms in numerical analysis.

In the next chapter we will study hardware circuits for floating-point arithmetic. In Chapter 8 hardware circuits for the computation of scalar products of floating-point vectors will be developed. These circuits compute the exact scalar product extremely fast, which has consequences for an error analysis of many numerical algorithms.

The error relations established for arithmetic operations so far can also be extended to the interval rows of Figure 1. Of course, such an extension would be mainly of theoretical interest since a basic principle of interval mathematics is that the computer controls the rounding error and other errors automatically. Nevertheless, as an example of interval arithmetic error relations, give Theorem 5.9 below. Since for all interval rows in Figure 1, arithmetic is defined by semimorphism, the validity of this theorem follows easily by employing Theorems 5.5 and 5.6.

In order to state Theorem 5.9, we require a few concepts of interval mathematics. See Chapter 9. The distance between the two intervals $A = [a_1, a_2]$, $B = [b_1, b_2] \in I\mathbb{R}$ is defined by

$$q(A, B) := \max\{|a_1 - b_1|, |a_2 - b_2|\},$$

while the absolute value of the interval A is given by

$$|A| := q(A, [0, 0]) = \max(|a_1|, |a_2|) = \max_{a \in A} |a|.$$

If $\boldsymbol{A} = (A_{ij}), \boldsymbol{B} = (B_{ij}) \in M_n I\mathbb{R}$ are matrices with interval components, the distance matrix and the absolute value matrix are defined by

$$q(A, B) := (q(A_{ij}, B_{ij})), \quad |A| := (|A_{ij}|).$$

Theorem 5.9. *Let $S = S(b, r, e1, e2)$ be a floating-point system and let $\Diamond : I\mathbb{R} \to IS$*

be the monotone upwardly directed rounding. Then

$$
\begin{aligned}
\bigwedge_{X=[x_1,x_2]\in I\mathbb{R}} \big(b^{e1-1} \le |x_1|, |x_2| \le B \\
&\Rightarrow \Diamond X = [\nabla x_1, \triangle x_2] = [x_1(1-\epsilon_1), x_2(1-\epsilon_2)] \\
&\qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
&\wedge \;\; X = [x_1, x_2] = [\nabla x_1(1-\epsilon_1), \triangle x_2(1-\epsilon_2)] \\
&\qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
&\wedge \;\; q(X, \Diamond X) < \epsilon^* |X| \;\; \wedge \;\; q(X, \Diamond X) < \epsilon^* |\Diamond X| \big).
\end{aligned}
$$

If $X, Y \in IS$ *and* $Z := [z_1, z_2] := X \boxdot Y$, $\circ \in \{+, \cdot, /\}$ *with* $0 \notin Y$ *for* $o = /$, *then*

$$
\begin{aligned}
\bigwedge_{X,Y\in IS} \big(b^{e1-1} \le |z_1|, |z_2| \le B \\
&\Rightarrow X \mathbin{\Diamond\!\!\!\!\circ} Y = [\nabla z_1, \triangle z_2] = [z_1(1-\epsilon_1), z_2(1-\epsilon_2)] \\
&\qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
&\wedge \;\; X \boxdot Y = [z_1, z_2] = [\nabla z_1(1-\epsilon_1), \triangle z_2(1-\epsilon_2)] \\
&\qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
&\wedge \;\; q(X \boxdot Y, X \mathbin{\Diamond\!\!\!\!\circ} Y) < \epsilon^* |X \boxdot Y| \\
&\wedge \;\; q(X \boxdot Y, X \mathbin{\Diamond\!\!\!\!\circ} Y) < \epsilon^* |X \mathbin{\Diamond\!\!\!\!\circ} Y| \big).
\end{aligned}
$$

If, moreover, a rounding $\Diamond : M_n I\mathbb{R} \to M_n IS$ *and operations* $\mathbin{\Diamond\!\!\!\!\circ}$ *in* $M_n IS$ *are defined by*

$$
\bigwedge_{\boldsymbol{X}=(X_{ij})\in M_n I\mathbb{R}} \Diamond \boldsymbol{X} := (\Diamond X_{ij}),
$$

(RG) $\displaystyle\bigwedge_{\boldsymbol{X},\boldsymbol{Y}\in M_n IS} \boldsymbol{X} \mathbin{\Diamond\!\!\!\!\circ} \boldsymbol{Y} := \Diamond(\boldsymbol{X} \boxdot \boldsymbol{Y}), \qquad \circ \in \{+, \cdot\},$

then

$$
\begin{aligned}
\bigwedge_{\boldsymbol{X}=[x_{ij}^{(1)}, x_{ij}^{(2)}]\in M_n I\mathbb{R}} \Bigg(\bigwedge_{ij} b^{e1-1} \le |x_{ij}^{(1)}|, |x_{ij}^{(2)}| \le B \\
&\Rightarrow \Diamond \boldsymbol{X} = [\nabla x_{ij}^{(1)}, \triangle x_{ij}^{(2)}] = [x_{ij}^{(1)}(1-\epsilon_1), x_{ij}^{(2)}(1-\epsilon_2)] \\
&\qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
&\wedge \;\; \boldsymbol{X} = [x_{ij}^{(1)}, x_{ij}^{(2)}] = [\nabla x_{ij}^{(1)}(1-\epsilon_1), \triangle x_{ij}^{(2)}(1-\epsilon_2)] \\
&\qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
&\wedge \;\; q(\boldsymbol{X}, \Diamond \boldsymbol{X}) < \epsilon^* |\boldsymbol{X}| \;\; \wedge \;\; q(\boldsymbol{X}, \Diamond \boldsymbol{X}) < \epsilon^* |\Diamond \boldsymbol{X}| \Bigg).
\end{aligned}
$$

If $\boldsymbol{X}, \boldsymbol{Y} \in M_n IS$ *and* $\boldsymbol{Z} := [z_{ij}^{(1)}, z_{ij}^{(2)}] := \boldsymbol{X} \boxed{\circ} \boldsymbol{Y}$, $\circ \in \{+, \cdot, /\}$, *then*

$$\begin{aligned}
\bigwedge_{\boldsymbol{X},\boldsymbol{Y} \in IS} \Bigg(\bigwedge_{ij} \; & b^{e1-1} \le |z_{ij}^{(1)}|, |z_{ij}^{(2)}| \le B \\
& \Rightarrow \boldsymbol{X} \Diamond\!\!\!\!\circ\; \boldsymbol{Y} = [\nabla z_{ij}^{(1)}, \triangle z_{ij}^{(2)}] = [z_{ij}^{(1)}(1-\epsilon_1), z_{ij}^{(2)}(1-\epsilon_2)] \\
& \qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
& \wedge \; \boldsymbol{X} \boxed{\circ} \boldsymbol{Y} = [z_{ij}^{(1)}, z_{ij}^{(2)}] = [\nabla z_{ij}^{(1)}(1-\epsilon_1), \triangle z_{ij}^{(2)}(1-\epsilon_2)] \\
& \qquad \textit{with } |\epsilon_1|, |\epsilon_2| < \epsilon^* = b^{1-r} \\
& \wedge \; q(\boldsymbol{X} \boxed{\circ} \boldsymbol{Y}, \boldsymbol{X} \Diamond\!\!\!\!\circ\; \boldsymbol{Y}) < \epsilon^* |\boldsymbol{X} \boxed{\circ} \boldsymbol{Y}| \\
& \wedge \; q(\boldsymbol{X} \boxed{\circ} \boldsymbol{Y}, \boldsymbol{X} \Diamond\!\!\!\!\circ\; \boldsymbol{Y}) < \epsilon^* |\boldsymbol{X} \Diamond\!\!\!\!\circ\; \boldsymbol{Y}| \Bigg).
\end{aligned}$$

Proof. The proof is simple. We only indicate it for $I\mathbb{R}$. The matrix properties then follow componentwise,

$$\begin{aligned}
q(X, \Diamond X) &= \max\{|x_1 - \nabla x_1|, |x_2 - \triangle x_2|\} \\
&= \max\{|\epsilon_1 x_1|, |\epsilon_2 x_2|\} < \epsilon^* |X|, \\
q(X, \Diamond X) &= \max\{|\nabla x_1(1-\epsilon_1) - \nabla x_1|, |\triangle x_2(1-\epsilon_2) - \triangle x_2|\} \\
&= \max\{|\epsilon_1 \nabla x_1|, |\epsilon_2 \triangle x_2|\} < \epsilon^* |\Diamond X|. \qquad \blacksquare
\end{aligned}$$

Using (RG), (R1) and (R2), we observe even more importantly than these error bounds, that semimorphisms provide maximal accuracy in the sense that there is no element of the screen N between the results of the operation $\circ$ and of its approximation $\boxed{\circ}$.

5.4 Subnormal Floating-Point Numbers

In a normalized floating-point system $S = S(b, r, e1, e2)$ an element $a \in S$ may have more than one additive inverse. Examples for this have been given in Section 3.6. In particular it was shown that the multiplicative unit 1 may have more than one additive inverse.

Theorem 3.12 in Section 3.7 establishes a necessary and sufficient condition for the uniqueness of an additive inverse in a discrete subset of the real numbers $\mathbb{R}$. In Theorem 3.13 this result is extended to more general structures.

In the case of a floating-point system $S(b, r, e1, e2)$ it was shown that this condition is met if for $e = e1$ unnormalized mantissas are admitted in addition to the normalized floating-point numbers of $S(b, r, e1, e2)$ (Definition 5.3). This extended set of floating-point numbers is specified by the following definition.

Definition 5.10. Let $S(b, r, e1, e2)$ be a floating-point system, and let

$$\begin{aligned} D &= D(b, r, e1) \\ &:= \left\{ x = \circ m b^{e1} \mid \circ \in \{+, -\}, m = \sum_{i=1}^{r} x_i b^{-i}, x_i \in \{0, 1, \ldots, b-1\} \right\}. \end{aligned}$$

The elements of $D(b, r, e1)$ are called denormalized or subnormal numbers.

The union of $S(b, r, e1, e2)$ and $D(b, r, e1)$

$$F = F(b, r, e1, e2) := S(b, r, e1, e2) \cup D(b, r, e1)$$

then forms the full floating-point system. ■

Subnormal numbers provide representations for values smaller than the smallest normalized number. They lower the probability of an exponent underflow. Subnormal numbers reduce the gap between the smallest normalized floating-point number and zero. The addition of subnormal numbers to the normalized floating-point numbers has been termed gradual underflow or graceful underflow. Denormalized numbers are not included in all the designs of arithmetic units that follow the IEEE arithmetic standard. This is mainly due to the high cost associated with their implementation. Compared to normalized floating-point numbers, the representation and the operations for denormalized numbers require a more complex design and possibly a longer overall execution time.

5.5 On the IEEE Floating-Point Arithmetic Standard

Early computers designed and built by Konrad Zuse, the Z3 (1941) and the Z4 (1945), are among the first computers that used the binary number system and floating-point for number representation [43, 63, 500, 501, 533, 640, 641, 642].

Floating-point numbers were normalized. The mantissa was of the form "$1.\cdots$" and had a value between 1 and 2. In the memory the leading 1 was not stored. It was used as what today is called a hidden bit. Both machines carried out the four basic arithmetic operations of addition, subtraction, multiplication, division, and the square root by hardware. In the Z4 floating-point numbers were represented by 32 bits. 8 bits were used for the exponent, one bit for the sign and 24 bits for the mantissa. During execution of the arithmetic operations two additional bits were used to improve the accuracy and to allow a correct rounding. Special representations and corresponding wirings were available to handle the three special values: 0, ∞, and *indefinite* (for $0/0$ or $\infty - \infty$, and others). An additional bit was used to distinguish between numbers and these special values. If it was 0, the word represented a normal floating-point number. If it was 1, the word represented a special value and the coding of the word decided whether it was 0, ∞, or *indefinite*. The least exponent was used to represent zero and the greatest exponent to represent ∞. These early computers built by Konrad Zuse possessed all the essential capabilities now required by the so-called IEEE floating-point arithmetic standard.

Today's advanced technology allows extra features such as additional word sizes and differences in the coding and number of special cases.

There are two different IEEE standards for floating-point arithmetic: IEEE 754 (1985) [644] and IEEE 854 (1987) [645].

IEEE 754 specifies formats and arithmetic for binary floating-point numbers. It defines exceptional conditions and specifies default handling of these conditions. One purpose of the standard is to greatly simplify porting of programs. For operations specified by the standard, numerical results and exceptions are uniquely determined by the value of the input data, the sequence of operations, and the destination formats. Thus, when a program is moved from one machine to another, the results of the basic operations will be the same in every bit if both machines support the standard.

IEEE 754 is a standard with base $b = 2$. It defines four formats for floating-point numbers, a 32-bit single precision and a 64-bit double precision format and extended formats for both of these which are used for intermediate results. The single extended format should have at least 44 bits, and the double extended format should have at least 80 bits. The single precision format uses 24 bits for the mantissa and 8 bits for the exponent, and the double precision format uses 53 bits for the mantissa and 11 bits for the exponent. The formats are designed to squeeze the maximum information into 32 and 64 bits respectively. Since in the binary number system the leading digit of a normalized floating-point number is always 1, this bit is not carried along when the numbers are stored. It is kept as a hidden bit. This gives room to use one bit for the sign of the mantissa in both the single and double precision format. The two extended formats use more bits for the mantissa and the exponent than the corresponding 32- and 64-bit formats.

The IEEE 854 standard allows either base $b = 2$ or base $b = 10$. Unlike IEEE 754 it does not specify how floating-point numbers are encoded into bits. It specifies constraints on the allowable number of bits for the representation of the mantissa for single and double precision formats.

To avoid a sign digit for the representation of the exponent e the IEEE binary standard uses a so-called characteristic or bias c. c is chosen in such a way that the biased exponent E is an unsigned integer,

$$E := c + e.$$

If the biased exponent is E, then the exponent of the floating-point number is $e = E - c$. In the IEEE binary standard the bias is 127 for single precision and it is 1023 for double precision. The two data formats are coded as shown in Figure 5.5 where M stands for the mantissa or significand.

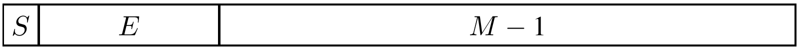


Figure 5.5. Coding of floating-point numbers in the IEEE 754 standard.

The value of the floating-point number f is

$$f = (-1)^S \cdot M \cdot 2^{E-c}.$$

If S is 1 the sign of the number is negative and it is positive if S is 0.

An advantage of the particular coding of floating-point numbers with a biased exponent is that nonnegative floating-point numbers can be treated as integers for comparison purposes.

Table 5.1 shows the parameters of the IEEE 754 formats.

	Single	Single extended	Double	Double extended
Word length in bits	32	≥ 44	64	≥ 80
Significand in bits	$1+23$	≥ 32	$1+52$	≥ 64
Exponent width in bits	8	≥ 11	11	≥ 15
Bias	127		1023	
Approximate range	$2^{128} \approx 3.8 \cdot 10^{38}$		$2^{1024} \approx 9 \cdot 10^{307}$	
Approximate precision	$2^{-24} \approx 1.6 \cdot 10^{-7}$		$2^{-53} \approx 10^{-16}$	

Table 5.1. The formats of the IEEE 754 standard.

The IEEE 754 standard requires that the four basic arithmetic operations of addition, subtraction, multiplication, and division are provided with four different roundings: rounding downwards, rounding upwards, rounding toward 0, and rounding to the nearest floating-point number. In the latter case it is additionally required that the midpoint between two adjacent floating-point numbers is rounded in a way that the result has a least significant digit which is even. This rounding scheme is called *round-to-nearest-even.*

The IEEE 754 standard uses denormalized numbers when the exponent is $emin$, i.e., denormalized mantissas are permitted. One reason for this may be that denormalized numbers guarantee the uniqueness of additive inverses (Theorem 3.12 in Section 3.7), i.e.,

$$x - y = 0 \Leftrightarrow x = y.$$

Denormalized numbers are called subnormal in 854 and denormal in 754. One also speaks of gradual underflow or graceful underflow. Providing for denormalized numbers in the hardware arithmetic unit makes the unit more complex, more costly, and possibly slower. Many arithmetic units that follow the IEEE arithmetic standard, therefore, do not support denormalized numbers directly but leave their treatment to software. Even hardware designs that implement denormalized numbers allow the programmer to avoid their use if faster execution is desired.

During a computation exceptional events like overflow or division by zero may occur. For such events the IEEE standard reserves some bit patterns to represent special quantities. It specifies special representations for $+\infty$, $-\infty$, $+0$, -0, and for NaN (not a number). Traditionally, an overflow or division by zero would cause a computation to be interrupted. There are, however, examples for which it makes sense for a computation to continue. In IEEE arithmetic the general strategy upon an exceptional event is to deliver a result and continue the computation. This requires the result of operations on or resulting in special values to be defined. Examples are: $4/0 = \infty$, $-4/0 = -\infty$, $0/0 = \mathtt{NaN}$, $\infty - \infty = \mathtt{NaN}$, $0 \cdot \infty = \mathtt{NaN}$, $\infty/\infty = \mathtt{NaN}$, $1/(-\infty) = -0$, $3/(+\infty) = -0$, $\log 0 = -\infty$, $\log x = \mathtt{NaN}$ when $x < 0$, $4 - \infty = -\infty$. When an NaN participates in a floating-point operation, the result is always an NaN.

The purpose of these special operations and results is to allow programmers to postpone some tests and decisions to a later time in the program when it is more convenient. When an exceptional event like division by zero or overflow occurs during execution of a program and the computation is continued the user must be informed about this circumstance. Status flags serve this purpose. The IEEE standard makes a distinction between five classes of exceptions: overflow, underflow, division by zero, invalid operation (like $0 \cdot \infty$ or $\infty - \infty$), and inexact. There is one status flag for each of the five exceptions. When any exception occurs the corresponding status flag is set. To help users to analyse exceptions in a program the IEEE standard recommends so-called *trap handlers* be installed. For more details see [187, 217, 309, 461].

The IEEE floating-point arithmetic standards 754 and 854 have been under revision for some time. It can be expected that in the future there will be only one *Standard for Floating-Point Arithmetic P 754*. The present draft specifies formats and methods for binary *and* decimal floating-point arithmetic in computer programming environments. Exception conditions are defined and default handling of these conditions is specified.

The new standard IEEE P 754 specifies 4 binary and 3 decimal floating-point formats. Table 5.2 shows the parameters of the IEEE P 754 formats.

	$b=2$	$b=2$	$b=2$	$b=2$	$b=10$	$b=10$	$b=10$
Word length	16	32	64	128	32	64	128
Significand	1 + 10	1 + 23	1 + 52	1 + 112	7	16	34
Exponent width	5	8	11	15	11	13	17
emax	+15	+127	+1023	+16383	+96	+384	+6144
emin	−14	−126	−1022	−16382	−95	−383	−6143
Bias	15	127	1023	16383	101	398	6176

Table 5.2. The binary and decimal formats of the IEEE P 754 standard.

For the binary 32 and 64 bit format and for the decimal 64 bit format also extended (so-called non-interchange) formats are specified.

The current *Draft Standard for Floating-Point Arithmetic P 754* does not explicitly require that each of the operations with directed roundings $\bigtriangledown\mkern-14mu+$, $\bigtriangledown\mkern-14mu-$, $\bigtriangledown\mkern-14mu\times$, $\bigtriangledown\mkern-14mu/$ and $\bigtriangleup\mkern-14mu+$, $\bigtriangleup\mkern-14mu-$, $\bigtriangleup\mkern-14mu\times$, $\bigtriangleup\mkern-14mu/$ is provided by a distinct operation code on future processors. With respect to rounded operations the current draft explicitely states: “Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the (rounding) modes.” The mode is determined by the value of a mode variable which assumingly has to be set before the operation is executed. This allows the conclusion that again the rounding will be separated from the arithmetic operation on future processors that confirm with the IEEE P 754 floating-point arithmetic standard. This raises the fear that interval arithmetic will again be slow and not in balance with the fast floating-point arithmetic on future processors.

An exact multiply and accumulate instruction or equivalently an exact scalar product is not required in the present draft of the IEEE P 754 standard.

Comments

For elementary binary floating-point computation the IEEE floating-point arithmetic standard, adopted in 1985, is undoubtedly thorough, consistent, and well defined [644]. It has been widely accepted and has been used in almost every processor developed since 1985. This has greatly improved the portability of floating-point programs.

However, computer technology has been dramatically improved since 1985. Arithmetic speed has gone from megaflops to gigaflops to teraflops, and it is already approaching the petaflops range. This is not just a gain in speed. A qualitative difference goes with it. At the time of the megaflops computer a conventional error analysis was recommended in every numerical analysis textbook. Today the PC is a gigaflops computer. For the teraflops or petaflops computer conventional error analysis is no longer practical. An avalanche of numbers is produced when a petaflops computer runs. If the numbers processed in one hour were to be printed (500 on one page, 1000 on one sheet, 1000 sheets 10 cm high) they would need a pile of paper that reaches from the earth to the sun and back. Computing indeed has already reached astronomical dimensions!

This brings to the fore the question of whether the computed result really solves the given problem. The only way to answer this question is by using the computer itself. Every floating-point operation is potentially in error. The capability of a computer should not be judged by the number of operations it can perform in a certain amount of time without asking whether the computed result is correct. It would be much more reasonable to ask how fast a computer can compute correctly to 3, 5, 10 or 15 decimal places for certain problems. If the question were asked that way, it would very soon lead to better computers. Mathematical methods that give an answer to this question are available for very many problems. Computers, however, are at present not designed in a way that allows these methods to be used effectively.

During the last several decades, methods for very many problems have been developed which allow the computer *itself* to validate or verify its computed results. See the literature list and Chapter 9 of this book. These methods compute *unconditional* bounds for a solution and can iteratively improve the accuracy of the computed result. Very few of these verification techniques need higher precision floating-point arithmetic. Fast double precision floating-point arithmetic is the basic arithmetical tool.

Two additional arithmetical features are fundamental and necessary:

I. fast hardware support for extended[4] interval arithmetic and

**II. a fast and exact multiply and accumulate operation or,
what is equivalent to it, an exact scalar product.**[5]

The IEEE standards seem to support interval arithmetic. They require the basic four arithmetic operations to have rounding to nearest, towards zero, and downwards and upwards. The latter two are essential for interval arithmetic. But almost all processors that provide IEEE arithmetic separate the rounding from the basic operation, which proves to be a severe drawback. In a conventional floating-point computation this does not cause any difficulties. The rounding mode is set only once. Then a large number of operations is performed with this rounding mode, one in every cycle. However, when interval arithmetic is performed the rounding mode has to be switched very frequently. The lower bound of the result of every interval operation has to be rounded downwards and the upper bound rounded upwards. Thus, the rounding mode has to be reset for every arithmetic operation. If setting the rounding mode and the arithmetic operation are equally fast this slows down the computation of each bound unnecessarily by a factor of two in comparison to conventional floating-point arithmetic. On almost all existing commercial processors, however, setting the rounding mode takes a multiple (three, five, ten) of the time that is needed for the arithmetic operation. Thus an interval operation is unnecessarily at least eight (or twenty and even more) times slower than the corresponding floating-point operation not counting the necessary case distinctions for interval multiplication and interval division. This is fatal for interval arithmetic. The rounding should be an integral part of the arithmetic

[4] including division by an interval that includes zero

[5] To achieve high speed all conventional vector processors provide a ‘multiply and accumulate’ instruction. It is, however, not accurate. By pipelining, the accumulation (continued summation) is executed very swiftly. The accumulation is done in floating-point arithmetic. The pipeline usually has four or five stages. What comes out of the pipeline is fed back to become the second input into the pipeline. Typically, four or five sums are built up independently before being added together. This so-called partial sum technique alters the sequence of the summands and causes errors in addition to the usual floating-point errors. A vectorizing compiler uses this ’multiply and accumulate’ operation within a user’s program as often as possible, since it greatly speeds up the execution. Thus the user loses complete control of his computation.

operation. Every one of the rounded arithmetic operations with rounding to nearest, downwards or upwards should be equally fast.

For interval arithmetic we need to be able to call each of the operations $\nabla\!\!\!\!\!+$, $\nabla\!\!\!\!\!-$, $\nabla\!\!\!\!\!\times$, $\nabla\!\!\!\!\!/$ and $\Delta\!\!\!\!\!+$, $\Delta\!\!\!\!\!-$, $\Delta\!\!\!\!\!\times$, $\Delta\!\!\!\!\!/$ as one single instruction, that is, the rounding must be inherent to each. Therefore in programming languages notations for arithmetic operations with different roundings should be provided. They could be:

+, −, *, / for operations with rounding to the nearest floating-point number,
+>, −>, *>, /> for operations with rounding upwards,
+<, −<, *<, /< for operations with rounding downwards, and
+|, −|, *|, /| for operations with rounding towards zero (chopping).

New operation codes are necessary! Implementation of fast interval arithmetic is discussed in Chapter 7 of this book.

Frequently used programming languages regrettably do not provide four plus, minus, multiply, and divide operators for floating-point numbers. This, however, does not justify generally separating the rounding from the arithmetic!

Instead, a future floating-point arithmetic standard should require that **every future processor** shall provide the 16 operations listed above. A future standard even can and should dictate names for the corresponding assembler instructions as for instance: *addp*, *subp*, *mulp*, *divp*, *addn*, *subn*, *muln*, *divn*, *addz*, *subz*, *mulz*, and *divz*. Here *p* stands for rounding toward positive, *n* for rounding toward negative, and *z* for rounding toward zero. With these operators interval operations can easily be programmed. They would be very fast and fully transferable from one processor to another.

This is exactly the way interval arithmetic is provided in the C++ class library C-XSC which has been successfully used for more than 15 years [206, 288]. Since C++ only allows operator overloading, the operations with the directed roundings cannot and are not provided in C-XSC. These operations are rarely needed in applications. If they are needed, interval operations are performed instead and the upper or lower bound of the result then gives the desired answer.

There is a need for this older mechanism where first the rounding mode has to be set and then the arithmetic operation is carried out, in particular for those applications where the rounding mode is to be selected randomly. In this case the status register should provide a new mode for choosing the rounding mode randomly (by a hardware random number generator). Then all standard rounding operations would be performed with roundings that change at random. This would not have any effect on the arithmetic operations which have their rounding specified.

To obtain **close** bounds for a solution, interval arithmetic has to be combined with defect correction or iterative refinement techniques. To be effective these techniques require an exact *multiply and accumulate* instruction or, equivalent to this, an exact scalar product. It is realized by accumulating products of the full double length into a wide fixed-point register. This fixed-point accumulation is completely free of truncation error. Fast hardware circuitry for an exact *multiply and accumulate* instruction for all kinds of computers is discussed in Chapter 8 of this book.

A very natural pipelining of the *multiply and accumulate* instruction leads to very fast and simple circuits. The stages: loading the data, computing, shifting, and accumulation of the products are performed in one pipeline. Furthermore fixed-point accumulation of the products is simpler than accumulation in floating-point arithmetic. Many intermediate steps that are executed in a floating-point accumulation such as normalization and rounding of the products and of the intermediate sum, composition into a floating-point number and decomposition into mantissa and exponent for the next operation, do not occur in the fixed-point accumulation. It is simply performed by a shift and addition of the products of full double length into a wide fixed-point register. This brings a considerable speed increase compared to a possibly wrong accumulation of the scalar product using conventional floating-point arithmetic. The hardware expenditure for it is comparable to that for a fast multiplier with an adder tree, accepted years ago. All that is actually needed is a tiny local memory on the arithmetic unit of about 1K bytes. We really can and we should afford this at a time where computer memory is measured in gigabytes. The arithmetic itself is not much different to what is available on a conventional CPU. Fixed-point accumulation is error free!

With a fast and exact *multiply and accumulate* instruction, fast quadruple or multiple precision arithmetic can also be easily provided. A multiple precision number is represented as an array of floating-point numbers. The value of this number is the sum of its components. It can be represented in the wide fixed-point register. Addition and subtraction of multiple precision variables or numbers can easily be performed in this register. Multiplication of two such numbers is simply a sum of products of floating-point numbers. Details are developed in Chapter 9 of this book.

Interval arithmetic can bring guarantees into computation while an exact *multiply and accumulate* instruction can bring high accuracy via defect correction methods and at high speed. It also is the key operation for fast multiple precision arithmetic for real and interval data. See Chapter 9 of this book.

Fast and accurate hardware support for I. and II. must be added to conventional floating-point arithmetic. Both are necessary extensions. Instead of the computer being merely a fast calculating tool, they would turn it into a scientific instrument for mathematics. Computing that is continually and greatly speeded up makes this step necessary and it is that very speed that calls conventional computing into question.

Of course, the computer would often have to do more work to obtain verified results. But the mathematical safety should be worth it. The step from assembler to higher programming languages or the use of convenient operating systems also consumes a lot of computing power and nobody complains about it. Fast computers in particular are often used for safety critical applications. Severe, expensive, and tragic accidents can occur if the eigenfrequencies of a heavy electricity generator, for instance, are erroneously computed, or if a nuclear explosion is incorrectly simulated.

The IEEE standards 754 and 854 and also the expected future standard P 754 are very conservative. *Multiply and add fused* is the only operation which goes beyond Zuse's elementary floating-point arithmetic. Otherwise, the standards merely extend the word size. This, however, cannot keep up with the dramatic gain in computer speed. Computations will soon need 10^{20} arithmetic operations and more. The tremendous progress in computer technology and the great increase in computer speed needs to be accompanied by extension of the mathematical capacity of the computer.

Advanced computer arithmetic as developed in this book takes a big stride forward. It provides a large number of additional high quality arithmetic operations and it supplies high speed tools for checking the quality of intermediate results and for dynamically extending the precision in critical situations. Basic tools to verify computed results ought to be standard equipment on every computer in the 21^{st} century. Computer arithmetic should go far beyond elementary floating-point arithmetic for a few fixed precisions.

Conclusion

In simplicity lies truth. The following requirements should be included in a future computer arithmetic standard:

(i) A well-defined double precision floating-point arithmetic.

(ii) Fast and direct hardware support for double precision interval arithmetic. See Chapter 7 of this book.

(iii) A fast and exact multiply and accumulate (i.e., continued addition) operation or, what is equivalent to it, an exact scalar product for the double precision format. It is the basic tool to achieve high speed dynamic (multiple) precision arithmetic for real and for interval data. Pipelining gives it high speed and exactitude brings very high accuracy into computation. See Chapters 8 and 9 of this book.

Of course, elementary functions with proven and reliable *a priori* error bounds are necessary as part of computer arithmetic. In C-XSC elementary functions are available for multiple precision arithmetics with large exponent ranges for real, interval, complex and complex interval data, see [331].

For problems which require an extremely large exponent range the 128-bit arithmetic of the proposed IEEE P 754 floating-point arithmetic standard may be useful.

However, more flexibility is highly desirable for future computing. Fast pipelined hardware implementation of an exact scalar product for the double precision format is no more complex than implementing a full 128-bit floating-point arithmetic. Multiple precision arithmetic as developed in Section 9.7 fully benefits from a high speed exact scalar product. The exponent range of the double precision format is not a limitation for multiple precision arithmetics. An exponent part can easily be added as a scaling factor, see Section 9.7.4 and [331].

Chapter 6

Implementation of Floating-Point Arithmetic on a Computer

In this chapter we deal with the implementation of arithmetic on a computer, and in particular by means of a floating-point screen S. The implementation will be described for all operations and for all rows displayed in Figure 1.

Floating-point arithmetic is based on integer arithmetic. Integer arithmetic is not a focus of this book. Nevertheless we review some basic aspects of integer arithmetic on computers in Section 6.1 of this chapter.

In subsequent sections we consider floating-point arithmetic. We split the implementation into two stages or levels with the details of level 2 based on level 1. The level 1 routines include different floating-point operations for the first row of Figure 1. These operations are defined by formula (RG) for all roundings of the set $\{\bigtriangledown, \bigtriangleup, \square_\mu, \mu = 0(1)b\}$. The level 2 routines then describe the operations defined in all the other rows of Figure 1.

With the very powerful computer technology that is available today, arithmetic on a computer, including floating-point arithmetic, will in general be supplied by the computer hardware. Vendors have developed sophisticated ideas and circuits to implement arithmetic on a computer. Discussion of these techniques is beyond the scope of this book.

Instead we give a detailed algorithmic description of the implementation of floating-point arithmetic. We show that this process can be split up into several independent routines, which include routines for approximation of the arithmetic operations, for normalization, and for the different roundings. We discuss most of these routines for two different kinds of accumulator, which we call the *long* and the *short*. The latter represents the minimum requirement. Chapter 8 of this book is devoted to the computation of the scalar product. There very fast hardware circuits are developed which compute the scalar product of two vectors with floating-point components exactly or with only a single rounding. This operation is used in the arithmetic of many rows of Figure 1. We treat this method as a level 1 operation.

The last section of this chapter contains a brief discussion of the level 2 routines. We simply summarize the definition of the operations in the different rows of Figure 1 and point out that they all can be performed by using the level 1 operations. These ideas have already been extensively studied in preceding chapters. We don't derive algorithms for these operations because they are simple and offer no difficulties in principle provided the level 1 routines are available and the operations are clearly defined.

6.1 A Brief Review on the Realization of Integer Arithmetic

This part of the book deals with floating-point arithmetic. Floating-point arithmetic is based on integer arithmetic. The implementation of integer arithmetic on computers is only of secondary significance here. Good books on the topic, for instance [309], are available. However, some basic appreciation of integer arithmetic on computers is needed to fully understand this and the following two chapters. We provide it in this section. We are not aiming for completeness. Nor are we discussing the most advanced technologies.

A few basic logical operators are the building blocks for integer arithmetic. Electrical circuits that realize these operators are called *gates*. The three operators *and*, *or*, and *not* form a complete operator system. These operators are defined by the following table.

		$a \wedge b$	$a \vee b$	$\neg a$
a	b	(and)	(or)	(not)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Table 6.1. Definition of the logical operators *and*, *or*, and *not*.

We shall represent these operators by the symbols shown in Figure 6.1.

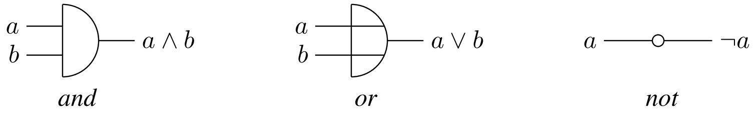
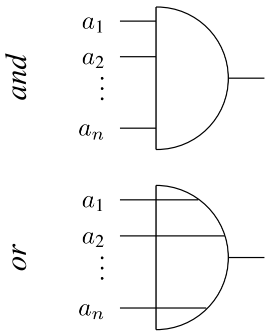


Figure 6.1. Symbols for the logical operators *and*, *or*, and *not*.

Occasionally generalizations of these operators with more input entities will be used, see Figure 6.2.

When a switch is used as basic bi-stable element with the state 0 if it is open, and 1 if it is closed, the logical operation *or* can be obtained by putting two switches in parallel and the operation *and* can be obtained by putting two switches in series. See Figure 6.3.



$$a, \quad a = 1 \Leftrightarrow \bigwedge_{i=1}^{n} (a_i = 1),$$

$$a, \quad a = 0 \Leftrightarrow \bigwedge_{i=1}^{n} (a_i = 0),$$

i.e., $a = 1$, if at least for one i, $a_i = 1$.

Figure 6.2. Symbols for *and* and *or* operators.

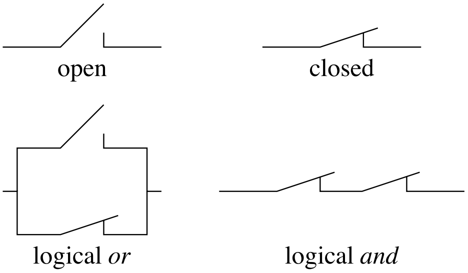


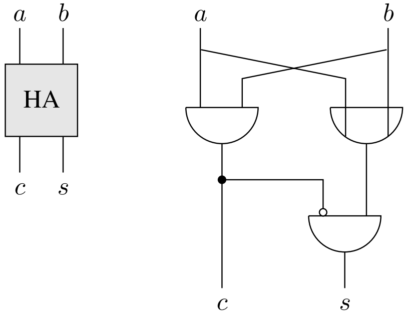
Figure 6.3. Realization of an *and* and an *or* gate by switches.

The British mathematician George Boole proved in 1854 in his *Treatise on the Laws of Human Thought* that every logical function f of n logical variables $v_1, v_2, \ldots, v_n$ can be fully described by an expression consisting of the logical values *true* and *false* and the three logical operators *and*, *or*, and *not*. Thus these three logical operators are called a complete operator system.

Instead of the logical operator system *and*, *or* and *not*, other logical operators can be used as a complete operator system. It is interesting that among the various complete operator systems, either of the two functions *nand* ($\neg(x \wedge y)$ and *nor* ($\neg(x \vee y)$) by itself form a complete operator system. We leave it to the reader to prove that the logical operators *and*, *or*, and *not* can be expressed by *nand* and *nor* alone.

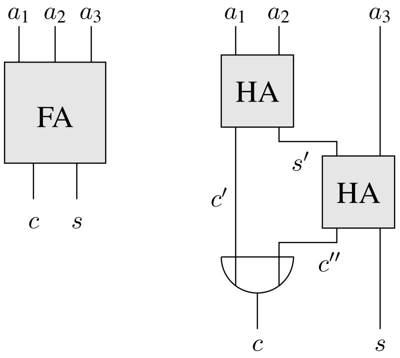
For binary numbers arithmetic operators can be built with the logical operators *and*, *or*, and *not* as well as with any other complete operator system. Elementary building blocks for arithmetic operators are the *half adder* (HA) and the *full adder* (FA). The half adder takes two binary inputs and produces two binary outputs, the sum digit (s), and the carry digit (c). A symbol for the half adder, a simple circuit and an analysis are given in Figure 6.4.

The full adder also produces two binary outputs, the sum digit (s) and the carry digit (c) from three binary inputs a_1, a_2, a_3. Its symbol, realization and analysis are shown in Figure 6.5.



a	0	1	0	1
b	0	0	1	1
$c = a \wedge b$	0	0	0	1
$\neg c$	1	1	1	0
$a \vee b$	0	1	1	1
s	0	1	1	0

Figure 6.4. Half adder.



a_1	0	1	0	1	0	1	0	1
a_2	0	0	1	1	0	0	1	1
a_3	0	0	0	0	1	1	1	1
s'	0	1	1	0	0	1	1	0
c'	0	0	0	1	0	0	0	1
c''	0	0	0	0	0	1	1	0
s	0	1	1	0	1	0	0	1
c	0	0	0	1	0	1	1	1

Figure 6.5. Full adder.

The half and full adders are themselves building blocks for the addition of n-bit-numbers (binary words). The simplest such adder is the *serial adder*. It computes the sum s of two summands

$$a = \sum_{i=0}^{n-1} a_i \cdot 2^i \quad \text{and} \quad b = \sum_{i=0}^{n-1} b_i \cdot 2^i, \quad a_i, b_i \in \{0,1\},\ i = 0(1)n-1,$$

using only one full adder. The summands a and b are placed into two registers as shown in Figure 6.6. The a-register is called the accumulator (AC). The addition begins with the least significant digits a_0 and b_0. Then the summands a and b are shifted one unit to the right and the sum digit of $a_0 + b_0$ is written into the empty space at the left end of the a-register. A possible carry is written into the carry register which is shown below the full adder in Figure 6.6. Continuing in this way, in each cycle a pair of digits a_i and b_i of a and b are shifted into the adder. There they are added together with the carry c_{i-1} from the addition of the less significant digits a_{i-1} and b_{i-1}. Thus the sum $s = a + b$ is built up in AC. The final sum s may have one more digit than the summands a and b. It is the carry digit of the addition $a_{n-1}+b_{n-1}$. n cycles are needed to compute the final sum by the serial adder.

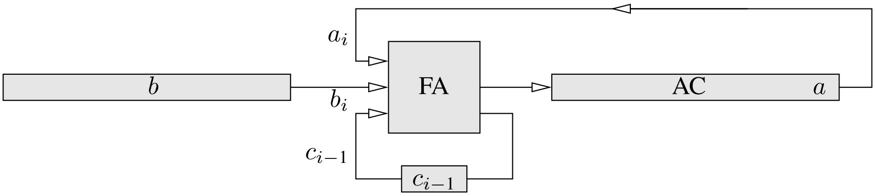


Figure 6.6. The serial adder.

The addition of two n-bit binary words can be done much faster by a *parallel adder*. Here instead of only one, n full adders are used as shown in Figure 6.7.

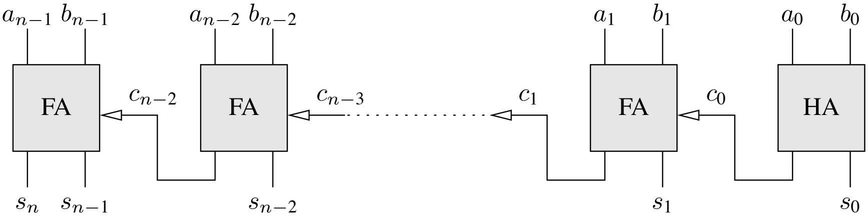


Figure 6.7. Parallel adder.

In the parallel adder, the adder for the least significant bits may be a half adder. s_n is the possible carry of the addition of the most significant bits a_{n-1} and b_{n-1} with the carry c_{n-2}.

For large n the parallel adder just described is not optimal with respect to computing time. Each gate needs its own run time and the many carries can make the addition quite slow.

Many ways to speed up the addition have been developed. We consider only one of these, the *carry-select-adder*. The idea is to subdivide the adder into several sections. The addition time for each of these sections then is much shorter. But carries may occur between adjacent sections. These carries are not known when the addition is started. Therefore for all but one adder section duplicate adders are provided. One of them adds the input data with an assumed input carry of zero, the other with an assumed input carry of one. Now all the adder sections can perform their additions simultaneously. Then the result is selected by the output carries of each adder section. See Figure 6.8.

The duplication within the adder sections in a carry-select-adder does not double the number of gates needed. The two halves of an adder section use nearly the same input data. Their data differ only in the input carry. Thus a look at the circuit for the full adder shows that most of the gates can be used for both halves of an adder section.

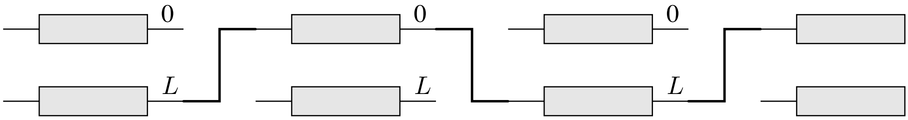


Figure 6.8. Carry-select-adder.

An even simpler parallel adder is the *John von Neumann adder*. It is the oldest parallel adder. Instead of full adders only half adders are used. See Figure 6.9.

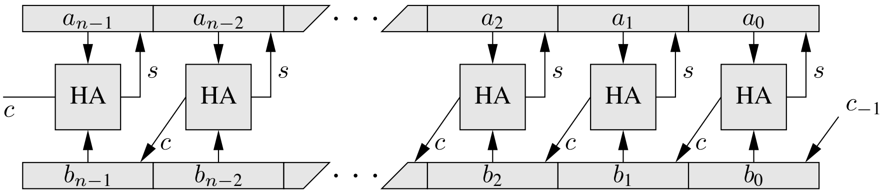


Figure 6.9. The John von Neumann adder.

The addition is performed in several cycles. After each cycle the sum exit of the half adders is returned into the a-register (the accumulator) while the carry is forwarded to the next more significant position of the b-register. After the first cycle b_0 is set to zero and a_0 is the least significant digit of the sum. After the second cycle b_1 is zero and a_1 is the next more significant digit of the sum, and so on. The addition is finished if all digits b_i, $i = 0(1)n - 1$, are zero. After at most n cycles this is the case. Then the a-register (accumulator) contains the sum. A possible carry c_{n-1} is the nth digit of the sum.

Another frequently used adder is the *carry-save-adder*, the CSA for short. Basically it consists of an array of full adders. See Figure 6.10.

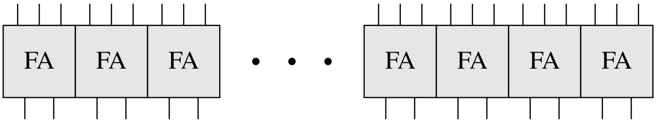
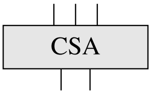


Figure 6.10. Arrangement of full adders in a carry save adder.

A full adder has three binary inputs and two binary outputs, the sum and the carry digit. Thus a CSA accepts three binary words as input and it produces two binary words as output. Since no carry propagation is performed between adjacent full adders the response time is very short. For brevity we shall use the following symbol for a CSA:



If the output of a CSA is fed back into the input registers a and b (as in the John von Neumann adder), the unit can be used to add in a new summand in each cycle.

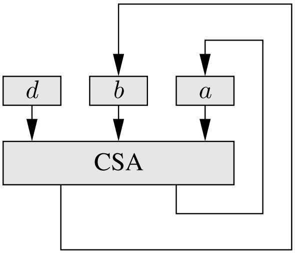


Figure 6.11. Continued addition by a carry save adder.

The runtime to add m summands $s_1, s_2, \ldots, s_m$ still can be shortened if several CSAs are used which are arranged in a chain. This saves the feedback time of the result into the a- and b-registers. The resulting adder circuit consists of $m - 2$ CSAs, each one consisting of n full adders. The output of every CSA together with a new summand is the input for the next CSA in the chain. For the addition of the two remaining summands (the output of the last CSA in the chain) a fast parallel adder has to be used.

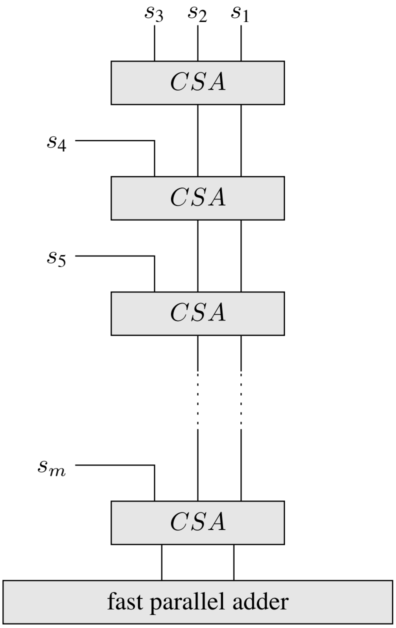


Figure 6.12. Addition of m summands by a carry save adder chain.

A further speed up could be obtained if several of the adders would work in parallel. The result is a tree structure of CSAs, an *adder tree* or *Wallace tree* ([604]). This structure is frequently used for fast multiplication where several multiples of one factor have to be added.

Figure 6.13 shows an adder tree for $m = 19$ summands. $m - 2 = 17$ CSAs are needed as for a chain. But the number of stages is reduced from 17 to 6.

While the number of CSAs and the number of stages in an adder tree are fixed, the topology is not. No carries are propagated in an adder tree. The response time, therefore, is very short. An adder tree is just an electrical network. Often the response time of the tree is shorter than the time needed for the final addition by a parallel adder.

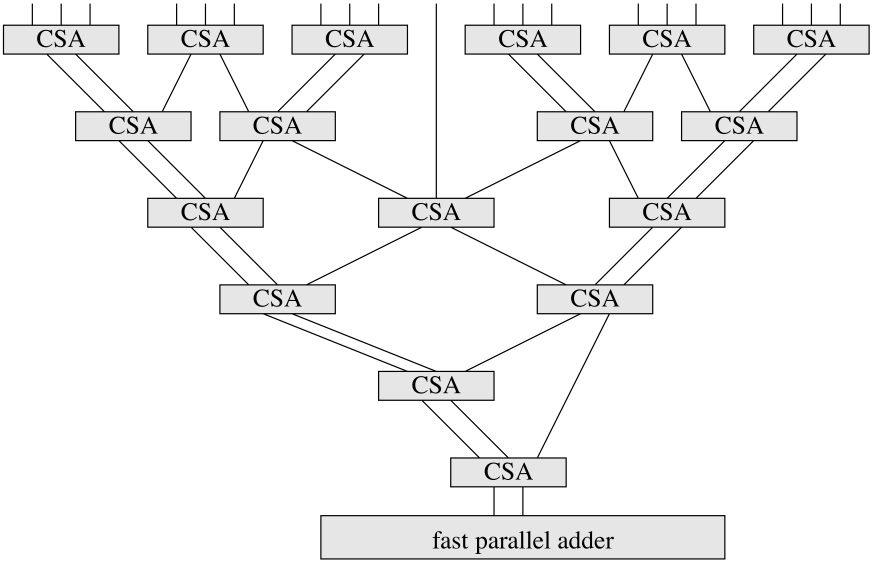


Figure 6.13. Adder tree for the addition of 19 binary words.

For fast multiplication many sophisticated methods have been developed. One method first computes multiples of the multiplicand (possibly in parallel) and then accumulates these in parallel.

Here we are going to discuss a very simple but straightforward multiplier and on the way mention techniques to speed up the multiplication. We assume that two n-digit binary numbers are to be multiplied. The product then has $2n$ digits. The simple multiplier is shown in Figure 6.14.

The two factors to be multiplied are placed in the registers MD for the multiplicand and MR for the multiplier. The multiplication begins with the least significant digit a_0 of MR.

The content of MD is multiplied by a_0 and the product is added to the accumulator AC. Then the combined register (AC,MR) is shifted one unit to the right. Now a_1 is

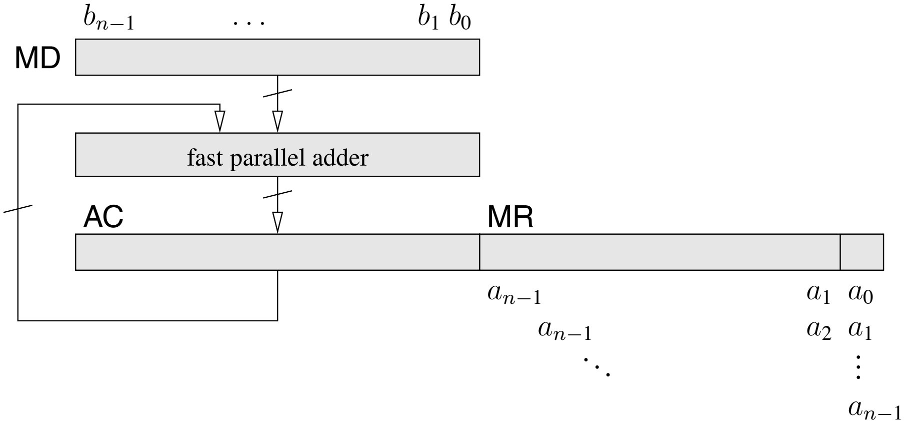


Figure 6.14. A simple multiplier.

in the position of the least significant digit and the procedure is repeated. After n such steps the product of double length appears in the combined register (AC,MR). A slashed line in Figure 6.14 indicates parallel data transfer.

The run time of this multiplication method can be shortened if an adder is provided for each digit of the multiplier. The adders are arranged in a chain. Each adder produces an n-digit sum and a carry bit. The least significant digit is a digit of the product. The upper $n - 1$ digits and the carry bit of the sum are passed along to the next adder. Compared to the method shown in Figure 6.14, this arrangement saves the shift and the feedback time of the intermediate sum Figure 6.15.

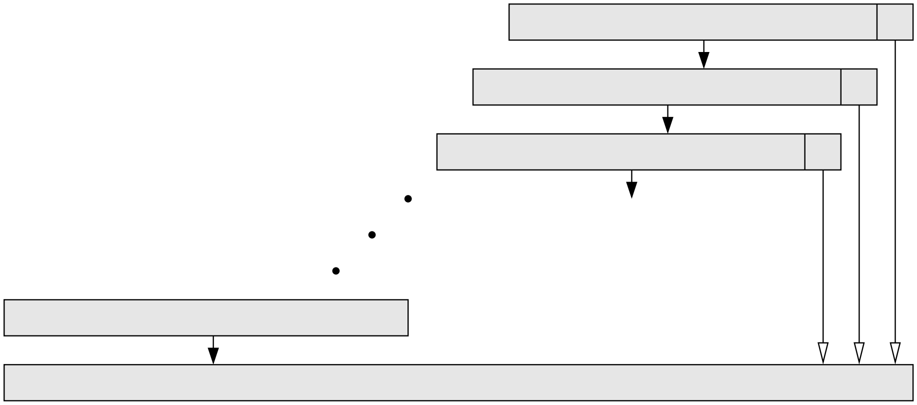


Figure 6.15. Fast multiplication.

If in any multiplication method the multiplier contains a block of k zeros, a shift by k digits can speed up the multiplication. In the case of a block of several ones in MR one can proceed as follows:

$$00\cdots 00\underset{v}{1}1\cdots 1\underset{u}{1}0\cdots 0 \;=\; \sum_{i=u}^{v} 2^i \;=\; 2^{v+1} - 2^u.$$

Thus the $v - u + 1$ additions of the multiplicand can be replaced by a subtraction in the position u and an addition in the position $v + 1$.

Large multipliers can also be built by making use of very fast smaller ones.

A very fast method of computing a product certainly is obtained if small multiples (by factors 2 or 3) of the multiplicand are added by an adder tree.

For division, various methods have also been developed and are in use. One method, of course, is the conventional approach which uses addition/subtraction and shifting. If proper positioning is assumed the standard method of computing x/y works as follows: Subtract y from x until what remains becomes negative. The number of subtractions minus one is the first digit of the quotient. Then shift y by one position to the right (or what remained by one position to the left). Now add y until what remains becomes positive. The second digit of the quotient is base b minus the number of these additions. Now again shift y by one position to the right (or what remains by one position to the left), and so on.

Another frequently applied method is to compute the reciprocal of the divisor by Newton's method and then the quotient by a fast multiplication. See, for instance, [309].

6.2 Introductory Remarks About the Level 1 Operations

The level 1 operations are designed as building blocks for the operations in the product sets and interval sets displayed in Figure 1. These operations include, in particular, all the operations of row 1 of Figure 1 for different roundings, including those operations which, in a narrower interpretation, are often called floating-point arithmetic. We develop algorithms for these operations in the following sections. To some extent the details of these algorithms depend on the technology being used.

Tremendous progress in computer technology has been made since the invention of the microprocessor in the early 1970s, and further advances can be expected for the future. The progress is expressed and becomes evident in measures like the clock frequency, the number of transistors on a chip, the memory size and the memory hierarchy, but also in the power of the design tools.

An early microprocessor was built on a chip with a few thousand transistors. It was running at 1 or 2 MHz and it provided an 8 bit adder. If at all, floating-point arithmetic could only be implemented in software by loops. As a simple consequence multiplication and division took significantly more computing time than floating-point addition and subtraction.

Today a microprocessor chip can carry 100 million transistors and more. It runs at 4 GHz. Floating-point operations are implemented in hardware for word sizes of 64, 80, and even more bits. Operations such as *addition*, *subtraction*, *multiplication*, *division*, and also certain compound operations like *multiply and add fused* can be delivered every cycle. High computing speed is supported by a memory hierarchy consisting of register memory, cache memory, main memory, and external memory. Under these circumstances vendors have developed sophisticated ideas and circuits to implement floating-point arithmetic. Discussion of these techniques is beyond the scope of this book.

It is, however, important to understand the basic principles, that semimorphic operations can be implemented for all rows of Figure 1, and what the minimum requirements for a given word size are. Computer hardware should more and more support arithmetic operations in the product spaces. The mathematical formulas for their realization are developed and listed in this chapter. Details of operations such as those for intervals and vectors and matrices are considered in the next two chapters.

We now turn to a detailed description of the implementation of what is conventionally called floating-point arithmetic on computers. Although nowadays on practically all computers floating-point operations are made available in hardware, we do not discuss particular hardware circuits here. Nor do we assume a particular representation of floating-point numbers. We confine the description to the logic flow using detailed flow diagrams.

We assume that floating-point numbers are elements of the floating-point system $S = S(b, r, e1, e2)$ and that they are given in the so-called *signed-magnitude representation*:

$$x = \text{mant}(x) \cdot b^{\exp(x)}$$

with $\text{mant}(x) = \circ \sum_{i=1}^{r} x_i b^{-i}$. We assume here, that the mantissa carries the sign of the number, $\circ \in \{+, -\}$, $x_1 \neq 0$, and $e1 \leq \exp(x) \leq e2$ with $e1 < 0$ and $e2 > 0$.

Zero is assumed to be represented by $\text{sign}(0) = +$, $\exp(0) = 0$, and all digits of the mantissa are 0. Thus, a floating-point number x is represented by the pair $(\text{mant}(x), \exp(x))$. The base b is implicit in the representation.

The *signed-magnitude representation* is employed in common practice, and it appears to be the most natural representation as well. The following algorithms are described for a fixed but arbitrary base $b > 1$.

However, the base 10 would be the most natural base at least for the representation of the mantissas. The base 10 is used in common practice, and it avoids many unnecessary anomalies and confusion that may occur when other bases are used in computers. In particular, errors arising from conversion of numbers to and from the decimal system are avoided. $b = 2$ or $b = 16$ in particular are other frequently used bases.

The architecture of the computer that we use to describe the algorithms is that of the classical computer, circa 1960. The floating-point numbers (mantissa and exponent together) are stored in *words*. The accumulator is able to accommodate twice as many digits as are in the floating-point mantissa and a few extra. This computer is a very convenient tool with which to express the essential steps of the algorithms.

For a different computer, for instance one in which the storage is organized bytewise or in which the accumulator is only 8 or 16 bits long, a few additional considerations of minor difficulty are necessary. A convenient way of implementing the arithmetic that we describe is then a simulation of our ideal computer on an actual computer.

Now we derive the algorithms for the implementation of arithmetic for the first row of Figure 1. According to our theory, the only way to define arithmetic for this row is by means of the formula

(RG) $\bigwedge_{x,y \in S} x \boxdot y := \square(x \circ y), \circ \in \{+,-,\cdot,/\}$ with $y \neq 0$ for $\circ = /$.

We implement (RG) for all roundings $\square \in \{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}$. Division is not defined if $y = 0$.

At first sight it seems to be doubtful that formula (RG) can be implemented on computers at all. To determine the approximation $x \boxdot y$, the exact but unknown result $x \circ y$ seems to be required in (RG). If, for instance, in the case of addition in a decimal floating-point system, x is of the magnitude 10^{50} and y of the magnitude 10^{-50}, more than 100 decimal digits in the accumulator would be necessary to represent $x + y$. Even if the largest computers had such long accumulators, it would hardly be necessary to employ so many digits. We will show by means of the following algorithms for all $\circ \in \{+,-,\cdot,/\}$ that whenever $x \circ y$ is not representable on the computer, it is sufficient to replace it by an appropriate and representable value $x \mathbin{\tilde{\circ}} y$. The latter will have the property $\square(x \circ y) = \square(x \mathbin{\tilde{\circ}} y)$ for all roundings $\square \in \{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}$. Then $x \mathbin{\tilde{\circ}} y$ can be used to define $x \boxdot y$ by means of the relations

$$\bigwedge_{x,y \in S} x \boxdot y = \square(x \circ y) = \square(x \mathbin{\tilde{\circ}} y), \quad \square \in \{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}.$$

The algorithms that implement this relation can, in principle, be separated into the following six steps.

1. Decomposition of x and y, i.e., separation of x and y into mantissa and exponent. If these parts of floating-point numbers are stored in separate words, this step is vacuous.

2. Determination of $x \mathbin{\tilde{\circ}} y$. It may be that $x \mathbin{\tilde{\circ}} y = x \circ y$.

3. Normalization of $x \mathbin{\tilde{\circ}} y$. If the result of ii is already normalized, this step can be skipped.

4. Treatment of exponent overflow and underflow.

5. Rounding of $x \mathbin{\tilde{\circ}} y$ to determine $x \boxdot y = \square(x \mathbin{\tilde{\circ}} y) = \square(x \circ y)$.

6. Composition, i.e., assembling the mantissa and exponent of the result into a floating-point number. If these parts of floating-point numbers are stored in separate words, this step is vacuous.

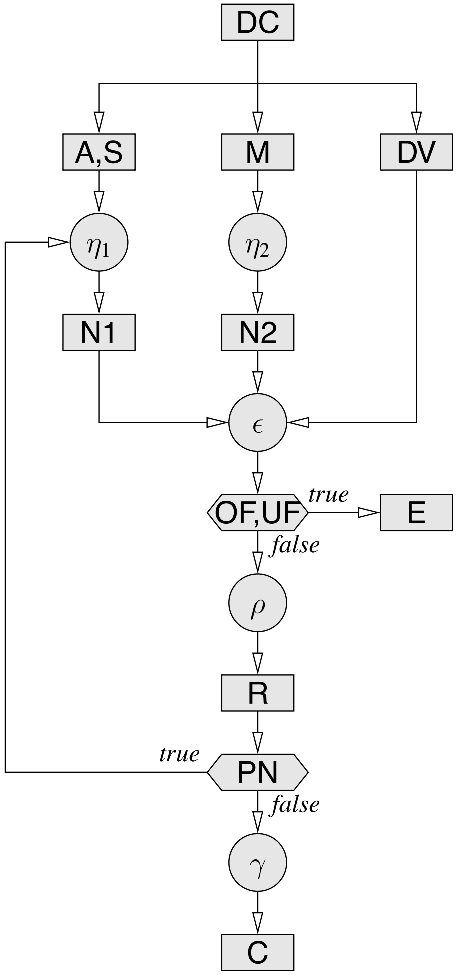


Figure 6.16. Flow diagram for the arithmetic operations. DC: Decomposition; A,S: Addition and Subtraction; M: Multiplication; DV: Division; N1,N2: Normalization; OF,UF: Overflow or Underflow; E: Exception; R: Rounding; PN: Postnormalization; C: Composition.

Figure 6.16 shows these six steps in the form of a flow diagram. Labels are introduced between single steps to label segments of the detailed diagrams, which we consider below.

We shall see that division can be executed in a manner that eliminates the need for normalization.

Since we deal with monotone roundings only, the normalization has to be performed before the rounding since otherwise the monotonicity is destroyed. After rounding a post normalization may become necessary.

In the following sections we briefly discuss the main features of the steps of the algorithms that are enumerated in Figure 6.16. We summarize the results by flow diagrams.

First we develop algorithms for the floating-point operations using a so-called *long accumulator*. By this we understand a computer register with one digit, which may be a binary digit, in front of the point and $2r + 1$ digits of base b after the point. See Figure 6.17(a). Here by point we mean the b-ary point, i.e., the decimal point for $b = 10$. A *long accumulator* is a convenient tool to implement the floating-point operations.

We shall discuss alternative algorithms for the execution of floating-point operations using a so-called *short accumulator* in Section 6.9. See Figure 6.17(b). By this we understand a computer register with one digit, which can be a binary digit, in front of the point and $r + 2$ digits of base b plus one binary digit after the point. The algorithms show that further reduction of the length of the accumulator is not possible if the operations are defined by (RG) for all roundings $\square \in \{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}$.

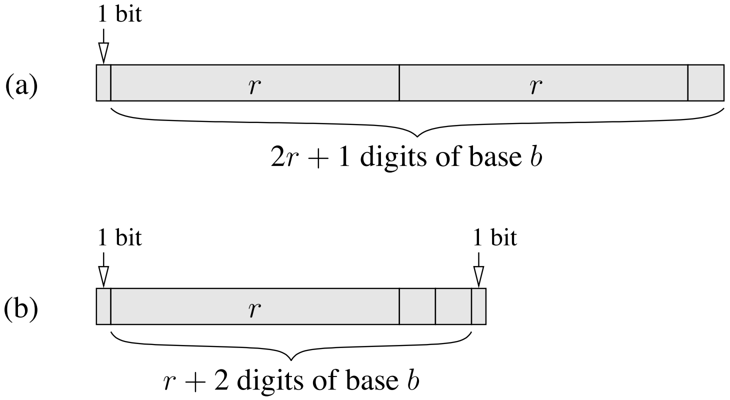


Figure 6.17. (a) long accumulator; (b) short accumulator.

The question of whether the algorithms using the *short* or the *long* accumulator are satisfactory has no simple answer. In the discussion of this question we assume that an adder of the same length as the accumulator is used.

If the accumulator and the algorithms are implemented in hardware, both methods may be nearly equal in speed. If the accumulator is available in hardware while the algorithms are to be implemented in software, the algorithms that use the long accumulator are likely to be faster because they are simpler. If the computer has a relatively short accumulator and storage word (for instance, 8 or 16 bits) and if the longer accumulator has to be simulated, then the algorithms using the short accumulator are probably faster.

In principle the following can be said: Extending the length of the short accumulator simplifies the algorithmic flow. So an ideal length of the accumulator might consist of the largest number of digits over which a parallel addition can be delivered in every addition cycle. This length is technology dependent. For the binary number system accumulators of 170 bits are in use, but this width can probably be extended further.

The key to the whole implementation is to take care that the formula (RG), as well as the rounding properties (R1)–(R4), which are operative, are strictly realized. This means that these formulas have to be valid for all $x, y \in S$ and not only for some or most of such x, y. Even an interval around zero, however small, may not be excluded from this requirement.

With these provisos, a principal result of the following sections is that the whole implementation can be separated into six *independent* steps as indicated in Figure 6.16 and its context. This means, in particular, that the provisional result, $x \mathbin{\tilde{\circ}} y$ for all $\circ \in \{+, -, \cdot, /\}$, may be determined independently of the rounding function. The latter is to be applied so that $\square\,(x \circ y) = \square\,(x \tilde{\circ} y)$ for all $x, y \in S$, and for all roundings $\square \in \{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$. Consequently, in the flow diagram of Figure 6.16, the rounding R may be any of the roundings $\square \in \{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$, and the entire algorithm delivers the result defined by (RG) and this particular rounding.

In the following flow diagrams, we use the usual conventions: rectangles denote statements; circles, labels; and hexagons, conditions. The flow diagrams are otherwise self-explanatory. The operands are always called x and y. The result is called z. In the following algorithms, all operator symbols $+, -, \cdot, /, \leq, \geq, <, >$ are defined and used in the sense of integer arithmetic. A left (resp. right) shift is expressed by a multiplication (resp. division) by powers of the base b. As a (rather awkward) device to describe certain digit manipulations, we occasionally employ the function $[x] :=$ entire(x). It determines the greatest integer less than or equal to x.

6.3 Addition and Subtraction

We assume that the long accumulator is being used. We denote the operands by $x = (mx, ex)$, $y = (my, ey)$, and the result by $z = (mz, ez)$. Without loss of generality, we assume that $ex \geq ey$. We set $ez := ex$. When $my \neq 0$ we distinguish two cases:

(i) $ex - ey \geq r + 2$. Here y is too small in absolute value to influence the first r digits of the sum $x + y$. In the case of the roundings $\square_\mu$, $\mu = 1(1)b - 1$, we therefore simply obtain $mz = mx$. However, an arbitrarily small y can change the mantissa of x in the case of the roundings $\triangledown$, $\triangle$, $\square_0$, and $\square_b$. To handle these cases correctly, we set

$$my := \text{sign}(my) \cdot b^{-(r+3)}.$$

(ii) $ex - ey \leq r + 1$. Here we divide my by b^{ex-ey}, i.e., we shift my to the right by $ex - ey$ digits of base b, and we set $my := my \cdot b^{-(ex-ey)}$.

Then we compute the intermediate sum $mz := mx + my$. Upon completion of this addition algorithm, the result is still to be normalized and rounded.

Figure 6.18 displays a flow diagram for the addition algorithm that we have just sketched. The first statement in this diagram represents the decomposition of x and y. The second statement selects the greater (and lesser) of the operands x and y. If realized in hardware this is just a comparison plus a selection.

$2r + 1$ digits of base b suffice for the representation of this sum in all cases. The binary digit in front of the point, which comes into play upon mantissa overflow, is not strictly necessary. It is convenient to use it to avoid complicated shiftings, which slow down the addition.

By way of comment on the condition $ex - ey \geq r + 2$ displayed in the flow chart of Figure 6.18, we interpolate an example in which $ex - ey = r + 1$ and which shows that for the rounding $\square_\mu$, addition of a digit in the $(r + 2)$th place can change all digits of mx.

In the following four examples let $r = 3$. We assume that the operands x and y are exact and are already appropriately positioned as indicated. The letter η indicates normalization.

x	0.	1	0	...	0	0	0	$\cdot b^3$
y	−0.	0	0	...	0	0	$b - \mu + 1$	$\cdot b^3$
$x + y$	0.	0	$b - 1$	...	$b - 1$	$b - 1$	$\mu - 1$	$\cdot b^3$
$\eta(x + y)$	0.	$b - 1$	$b - 1$	...	$b - 1$	$\mu - 1$		$\cdot b^2$
$\square_\mu(x + y)$	0.	$b - 1$	$b - 1$	...	$b - 1$			$\cdot b^2$

The effect does not occur for $\mu = b$, i.e., we assume here $\mu \leq b - 1$. The following example shows that a corresponding addition in the $(r + 3)$th place, however, does not change the mantissa of x.

x	0.	1	0	...	0	0	0	0	$\cdot b^3$
y	−0.	0	0	...	0	0	0	$b - \mu + 1$	$\cdot b^3$
$x + y$	0.	0	$b - 1$	...	$b - 1$	$b - 1$	$b - 1$	$\mu - 1$	$\cdot b^3$
$\eta(x + y)$	0.	$b - 1$	$b - 1$	...	$b - 1$	$b - 1$	$\mu - 1$		$\cdot b^2$
$\square_\mu(x + y)$	0.	1	0	...	0				$\cdot b^2$

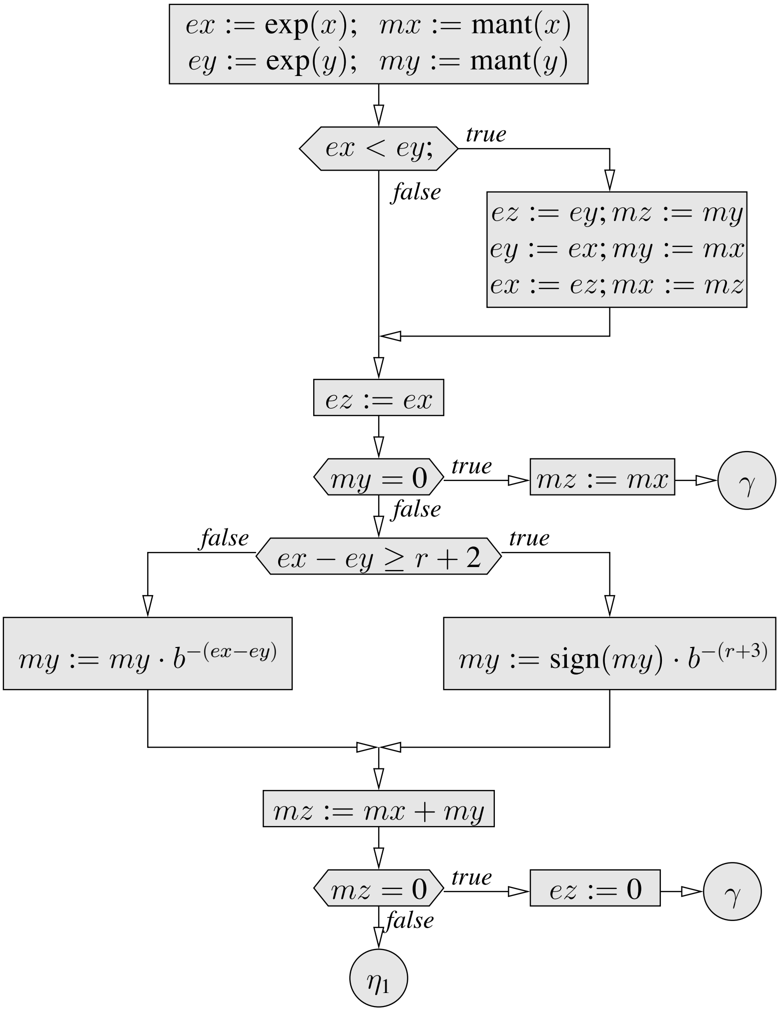


Figure 6.18. Execution of the addition $x \mathbin{\widetilde{+}} y$.

These examples make sense only if $b - \mu + 1$ is a nonzero digit of the number system that is used, i.e., if $1 \leq b - \mu + 1 \leq b - 1$. From this inequality we obtain $2 \leq \mu$. On the other hand it was assumed $\mu \leq b - 1$. From the two inequalities one may conclude $b \geq 3$. We comment on the binary case $b = 2$ below. If $b \geq 3$, the rounding $\square_1$ $(\mu = 1)$ is certainly of only minor interest.

Now we consider the case $b = 2$, for which the two examples studied above are irrelevant. The rounding $\square_1$ corresponds to the standard rounding to the nearest number of the screen. We show again by a simple example that an addition can change all digits of mx if the exponents differ by $r + 1$:

x	0.	1	0	...	0	0	0	0	0	$\cdot b^3$
y	−0.	0	0	...	0	0	1	0	1	$\cdot b^3$
$x + y$	0.	0	1	...	1	1	0	1	1	$\cdot b^3$
$\eta(x + y)$	0.	1	1	...	1	0	1	1		$\cdot b^2$
$\square_1(x + y)$	0.	1	1	...	1					$\cdot b^2$

An addition, however, does not change the mantissa of mx if the difference in the exponents is $r + 2$. We illustrate this by a further example:

x	0.	1	0	...	0	0	0	0	0	$\cdot b^3$
y	−0.	0	0	...	0	0	0	1	1	$\cdot b^3$
$x + y$	0.	0	1	...	1	1	1	0	1	$\cdot b^3$
$\eta(x + y)$	0.	1	1	...	1	1	0	1		$\cdot b^2$
$\square_1(x + y)$	0.	1	0	...	0					$\cdot b^3$

These examples show that if $b \geq 3$ and $\mu \geq 2$, as well as if $b = 2$ and $\mu = 1$, the circumstance $ex - ey = r + 1$ must be treated under case 2 where nondegenerate operations are performed on the mantissa. This state of affairs occurs in practice, for instance, when $b = 2$, 10, or 16 and for rounding to the nearest number of the screen $(\mu = b/2)$.

We now give several examples to illustrate the addition algorithm. Let $r = 4$.

Examples. (a) We consider the case $ex - ey \geq r + 2$.

(1) $x = +0.d_1 d_2 d_3 d_4 \cdot b^3$, $d_4 \neq b - 1$, and $y > 0$. Then

$$x \tilde{+} y = 0.d_1 d_2 d_3 d_4 001 \cdot b^3,$$

$$\triangledown(x + y) = \triangledown(x \tilde{+} y) = 0.d_1 d_2 d_3 d_4 \cdot b^3,$$

$$\vartriangle(x + y) = \vartriangle(x \tilde{+} y) = 0.d_1 d_2 d_3 (d_4 + 1) \cdot b^3,$$

$$\square_\mu(x + y) = \square_\mu(x \tilde{+} y) = 0.d_1 d_2 d_3 d_4 \cdot b^3 \text{ for } 1 \leq \mu \leq b - 1.$$

(2) $x = -0.d_1d_2d_3d_4 \cdot b^3$, $d_4 \neq b-1$, and $y < 0$. Then

$$\begin{aligned}
x \tilde{+} y &= -0.d_1d_2d_3d_4001 \cdot b^3, \\
\bigtriangledown(x+y) &= \bigtriangledown(x \tilde{+} y) = -0.d_1d_2d_3(d_4+1) \cdot b^3, \\
\bigtriangleup(x+y) &= \bigtriangleup(x \tilde{+} y) = -0.d_1d_2d_3d_4 \cdot b^3, \\
\square_\mu(x+y) &= \square_\mu(x \tilde{+} y) = -0.d_1d_2d_3d_4 \cdot b^3 \ \text{ for } \ 1 \leq \mu \leq b-1.
\end{aligned}$$

(3) $x = +0.1000 \cdot b^3$ and $y < 0$. Then

$$\begin{aligned}
x \tilde{+} y &= 0.0(b-1)(b-1)(b-1)\big|(b-1)(b-1)(b-1) \cdot b^3, \\
\eta(x \tilde{+} y) &= 0.(b-1)(b-1)(b-1)(b-1)\big|(b-1)(b-1) \cdot b^2, \\
\bigtriangledown(x+y) &= \bigtriangledown(x \tilde{+} y) = 0.(b-1)(b-1)(b-1)(b-1) \cdot b^2, \\
\bigtriangleup(x+y) &= \bigtriangleup(x \tilde{+} y) = 0.1000 \cdot b^3, \\
\square_\mu(x+y) &= \square_\mu(x \tilde{+} y) = 0.1000 \cdot b^3 \ \text{ for } \ 1 \leq \mu \leq b-1.
\end{aligned}$$

(4) $x = -0.1000 \cdot b^3$ and $y > 0$. Then

$$\begin{aligned}
x \tilde{+} y &= -0.0(b-1)(b-1)(b-1)\big|(b-1)(b-1)(b-1) \cdot b^3, \\
\eta(x \tilde{+} y) &= -0.(b-1)(b-1)(b-1)(b-1)\big|(b-1)(b-1) \cdot b^2, \\
\bigtriangledown(x+y) &= \bigtriangledown(x \tilde{+} y) = -0.1000 \cdot b^3, \\
\bigtriangleup(x+y) &= \bigtriangleup(x \tilde{+} y) = -0.(b-1)(b-1)(b-1)(b-1) \cdot b^2, \\
\square_\mu(x+y) &= \square_\mu(x \tilde{+} y) = -0.1000 \cdot b^3 \ \text{ for } \ 1 \leq \mu \leq b-1.
\end{aligned}$$

(b) Now we consider the case $ex - ey \leq r + 1$ and in particular for $b = 10$.

(1) $x = 0.1000 \cdot 10^6$ and $y = -0.5001 \cdot 10^1$. Then

$$\begin{aligned}
x + y &= 0.0999\big|9499\big|9 \cdot 10^6, \\
\eta(x+y) &= 0.9999\big|4999 \cdot 10^5, \\
\bigtriangledown(x+y) &= 0.9999 \cdot 10^5, \\
\bigtriangleup(x+y) &= 0.1000 \cdot 10^6, \\
\square_\mu(x+y) &= 0.9999 \cdot 10^5 \ \text{ for } \ 5 \leq \mu \leq 9.
\end{aligned}$$

(2) $x = 0.1000 \cdot 10^6$ and $y = -0.5000 \cdot 10^1$. Then

$$\begin{aligned} x + y &= 0.0999|9500|0 \cdot 10^6, \\ \eta(x+y) &= 0.9999|5000 \cdot 10^5, \\ \bigtriangledown(x+y) &= 0.9999 \cdot 10^5, \\ \bigtriangleup(x+y) &= 0.1000 \cdot 10^6, \\ \square_\mu(x+y) &= 0.1000 \cdot 10^6 \ \text{ for } \ \mu = 5. \end{aligned}$$

In the last two examples the operands are identical except for the last digit of y. If the $(2r+1)$th digit of the accumulator was in fact not present, the corresponding sums $x + y$ would be identical. Since these sums differ, these two examples show that it is really necessary to have $2r + 1$ digits available in the accumulator if the addition is to be executed as prescribed and the result is required to be correct for all the roundings $\bigtriangledown$, $\bigtriangleup$, and $\square_5$.

However, we shall see in Section 6.9 that the desired correct answer can also be obtained with the short accumulator.

6.4 Normalization

A normalization consists of appropriate right or left shifts of the mantissa and a corresponding adjustment of the exponent.

The addition algorithm described in Figure 6.18 can result in a mantissa with the property $|mz| \geq 1$. The inequality

$$|mx| + |my| \cdot b^{-(ex-ey)} < 1 + 1,$$

however, shows that a shift of at most one digit to the right may be required. It also shows that a single bit is sufficient to record such an overflow of the mantissa for an addition since the leading digit can only be unity or zero. If this bit is one after an addition, then the $2r$th as well as the $(2r+1)$th digit in the accumulator is necessarily zero. Thus the result after the right shift is still represented correctly, having all the digits.

After an addition, however, a left shift of more than one digit may be necessary. Figure 6.19 shows the flow diagram for the execution of the normalization after an addition or subtraction. We assume that the hardware is able to compute the number of leading zeros in the intermediate sum. Otherwise a loop would have to be provided to do the left shift digit by digit. We denote the number of leading zeros by l. In general the shift of the mantissa and the corresponding correction of the exponent are executed in different parts of the hardware of the computer.

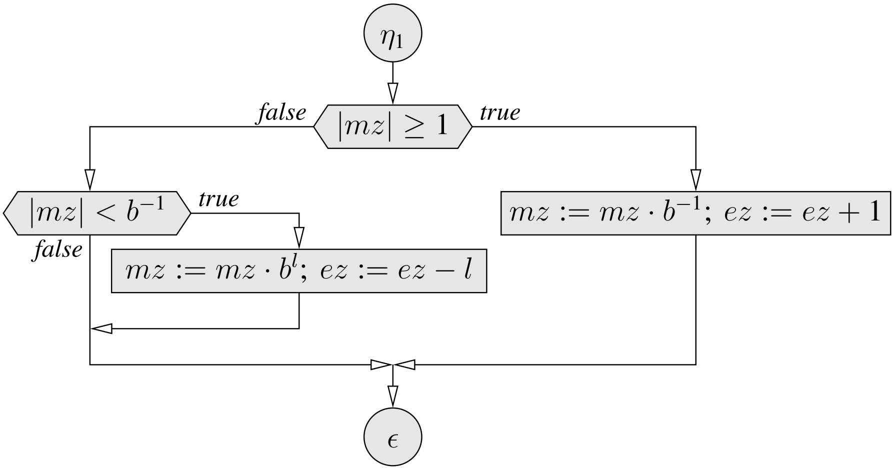

η_1
false
$|mz| \geq 1$
true
$|mz| < b^{-1}$
true
false
$mz := mz \cdot b^l;\ ez := ez - l$
$mz := mz \cdot b^{-1};\ ez := ez + 1$
ϵ

Figure 6.19. Execution of the normalization after the addition.

The algorithm for multiplication will be discussed in the next section. However, we foreshadow here that the normalization procedure for multiplication is simpler. To see this, note that since $b^{-1} \leq |mx| < 1$ and $b^{-1} \leq |my| < 1$, we obtain $b^{-2} \leq |mx||my| < 1$. Then no right shift and at most one left shift is necessary after multiplication. Figure 6.20 shows the flow diagram for the execution of the normalization after a multiplication.

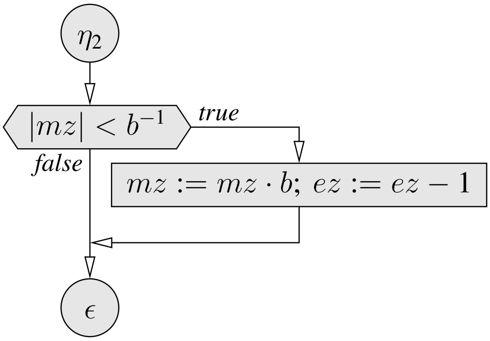


Figure 6.20. Execution of the normalization after a multiplication.

6.5 Multiplication

If $x = 0$ or $y = 0$ then $x \cdot y = 0$ and $\square(x \cdot y) = 0$. Otherwise we determine $ez := ex + ey$ (the possibility that ez lies outside the range $[e1, e2]$ is considered in Section 6.9) and $mz := mx \cdot my$. This multiplication can be executed correctly within the long accumulator. Because $b^{-2} \leq |mx| \cdot |my| < 1$, a normalization consisting of at most one left shift may be necessary. Figure 6.21 displays the flow diagram for multiplication. The relevant normalization procedure is described in the previous section.

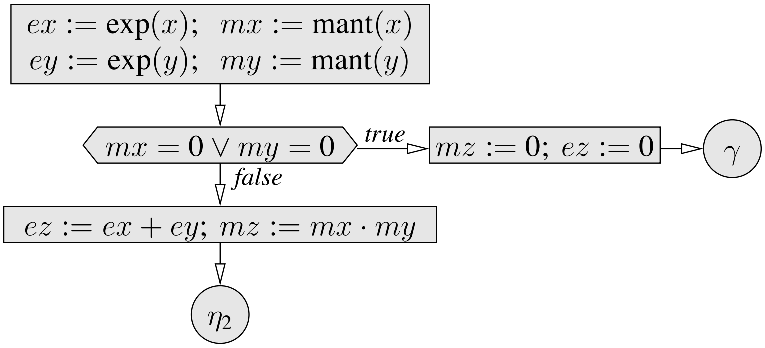

$ex := \exp(x);\ \ mx := \text{mant}(x)$
$ey := \exp(y);\ \ my := \text{mant}(y)$
$mx = 0 \vee my = 0$
true
$mz := 0;\ ez := 0$
γ
false
$ez := ex + ey;\ mz := mx \cdot my$
η_2

Figure 6.21. Execution of multiplication.

Every digital processor should also be able to deliver the double length unrounded product as it is needed for many applications.

6.6 Division

We determine the quotient $x \mathbin{\tilde{/}} y$. If $y = 0$, an error message is to be given. An exception handling routine may be started, which if $x \neq 0$ may set the quotient to $\pm\infty$. If $x = 0$ and $y \neq 0$, then $x/y = 0$ and $\square(x/y) = 0$. If $x \neq 0$ and $y \neq 0$ we determine $ez := ex - ey$ (the possibility that ez lies outside the range $[e1, e2]$ is dealt with in Section 6.9). We read mx into the first r digits of the accumulator and set all the remaining digits of the accumulator to zero. If $|mx| \geq |my|$, we shift the contents of the accumulator to the right by one digit and increase ez by unity. Now we divide the contents of the accumulator by my. It is sufficient to determine the first $r + 1$ digits of the quotient. We denote this quotient by $(mx/my)_{r+1}$. It is already normalized. The $(r + 1)$th digit is only needed for the execution of the roundings $\square_\mu$, $1 \leq \mu \leq b - 1$. However for one of the roundings $\triangledown$, $\triangle$, $\square_0$ and $\square_b$, even if the $(r + 1)$th digit is zero, an arbitrarily small remainder can influence the mantissa (even its leading digit). The contents of the accumulator after the first $r + 1$ digits is the remainder. We denote it by rmd.

For proper treatment of all these cases, we proceed as follows:

(i) $mz := (mx/my)_{r+1}$, i.e., compute the first $r + 1$ digits of mz.

(ii) If $rmd = 0$, there is no further treatment of mz.

(iii) If $rmd \neq 0$, we write a 1 in the $(r + 3)$th digit of mz.

Figure 6.22 shows a flow chart for the execution of division.

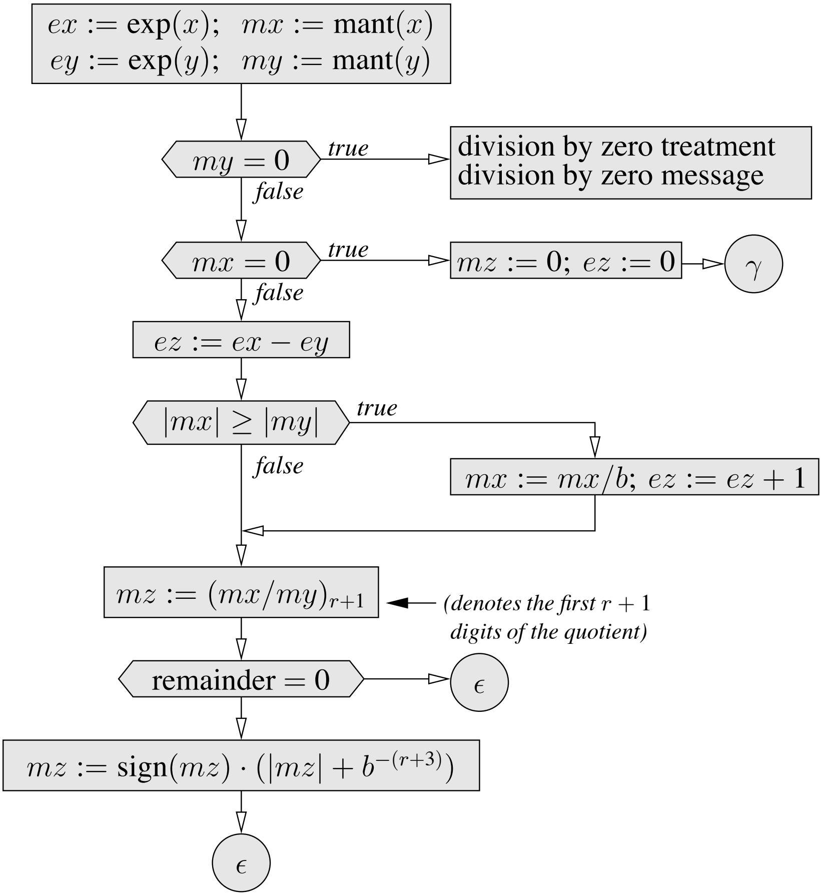

$ex := \exp(x);\ \ mx := \text{mant}(x)$
$ey := \exp(y);\ \ my := \text{mant}(y)$
$my = 0$
true
false
division by zero treatment
division by zero message
$mx = 0$
true
false
$mz := 0;\ ez := 0$
γ
$ez := ex - ey$
$|mx| \geq |my|$
true
false
$mx := mx/b;\ ez := ez + 1$
$mz := (mx/my)_{r+1}$
(denotes the first $r+1$ digits of the quotient)
remainder $= 0$
ϵ
$mz := \text{sign}(mz) \cdot (|mz| + b^{-(r+3)})$
ϵ

Figure 6.22. Execution of the division $x \mathbin{\tilde{/}} y$.

6.7 Rounding

After normalization the mantissa, which in general has a length of $2r + 1$ digits, is to be rounded to r digits. All roundings of the set $\square \in \{\nabla, \triangle, \square_\mu, \mu = 0(1)b\}$ are monotone functions. Rounding a number either truncates the number or enlarges its absolute value. The latter case requires another add operation. So rounding is a relatively complicated process. The add operation may cause a carry which propagates from the least significant bit across all digits of the number. Thus a rounding can alter the mantissa so that $|mz| = 1$ which causes another normalization. See Figure 6.16.

For the roundings $\square_\mu$, $\mu = 1(1)b - 1$ we give a symbolic formal description of the rounding part of Figure 6.16 in the following Figure 6.23. The description employs the function $[x] :=$ entire(x) which determines the greatest integer less than or equal to x.

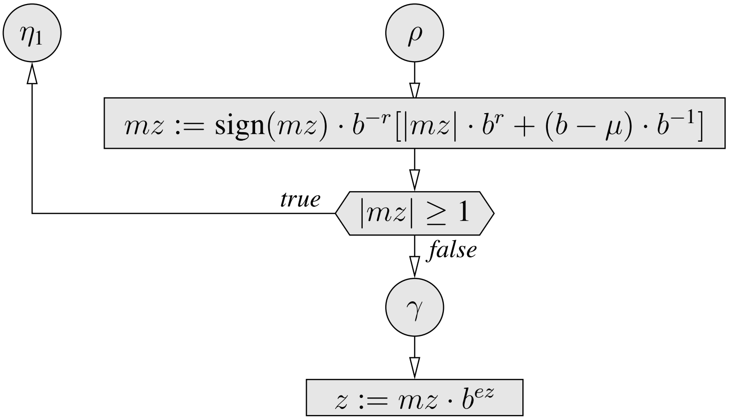

η_1
ρ
$mz := \text{sign}(mz) \cdot b^{-r}[|mz| \cdot b^r + (b-\mu) \cdot b^{-1}]$
true
$|mz| \geq 1$
false
γ
$z := mz \cdot b^{ez}$

Figure 6.23. Execution of the roundings $\square_\mu$, $\mu = 1(1)b-1$.

The last statement in Figure 6.23 denotes the composition of mz and ez to form the result z. See Figure 6.16.

Figure 6.24 gives a symbolic description in the form of a flow diagram for the rounding $\bigtriangledown$.

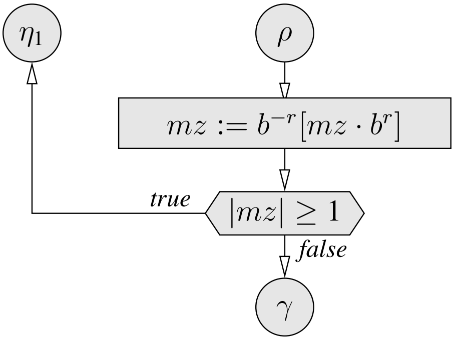


Figure 6.24. Execution of the rounding $\bigtriangledown$.

Here the description does not explicitly indicate that an addition is involved. The add operation is implicit in the function $[mz \cdot b^r]$ if it operates on negative values.

Similarly the execution of the rounding $\bigtriangleup$ requires an add operation for positive arguments. The rounding $\bigtriangleup$ also can be produced in terms of $\bigtriangledown$ by using the relation $\bigtriangleup x = -\bigtriangledown(-x)$.

The simplest of all the roundings under consideration, in a certain sense, is the rounding towards zero, $\square_b$ in our notation. It is also known as truncation or chopping. Here no additional add operation is involved for the rounding and as a consequence no additional normalization after the rounding can occur. Figure 6.25 shows the corresponding flow chart.

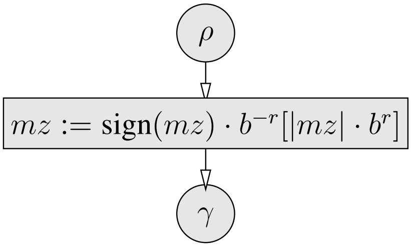


Figure 6.25. The rounding towards zero $\square_b$, truncation or chopping.

Base 16 and rounding by truncation was used on IBM/360 computers for many years, and is still used in their /370 and /390 architectures. Although these machines performed pretty well, their numerical accuracy has frequently been judged as very poor in the literature. For rounding by truncation the relative rounding error is bounded by one ulp (unit in the last place), while for rounding to nearest it is bounded by $1/2$ ulp. Such arguments are of value as long as the computer technology is relatively limited. But their value drops as computing technology improves, which it has been doing very rapidly. With increasing word length the fact that truncation requires no additional add operation for the rounding process may outweigh the rounding error problem and speed up the operation.

We mention here again that the rounding $\triangledown$ can also be performed by truncation if negative numbers are represented by their b-complement. See (5.2.9) and its context in Section 5.2. This may require taking the b-complement twice.

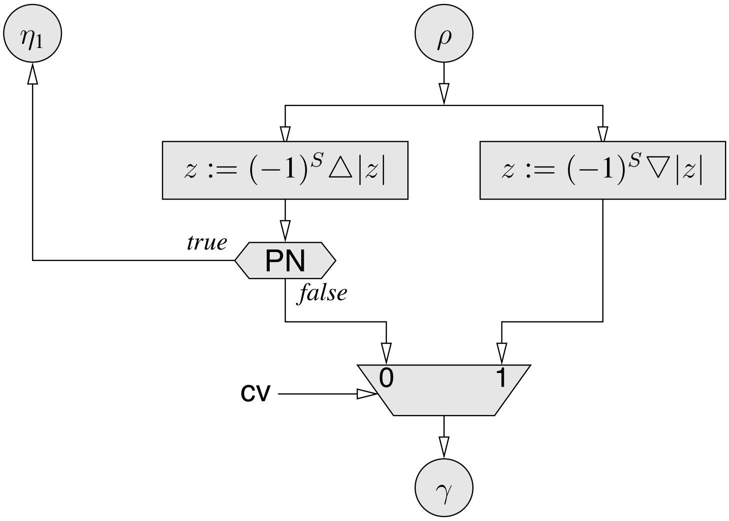
Finally, we remark once more that the algorithms for all operations $\circ \in \{+, -, \cdot, /\}$ show that the intermediate result $x \,\widetilde{\circ}\, y$ can be chosen so that for all roundings $\square \in \{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}$, the property

$$\bigwedge_{x,y \in S} \square\,(x \circ y) = \square\,(x \,\widetilde{\circ}\, y)$$

holds. This assures us that for the rounding step R in the execution of arithmetic operations in Figure 6.16, any of the roundings $\triangledown$, $\vartriangle$, $\square_\mu$, $\mu = 0(1)b$ can be employed. The result obtained is the one defined by (RG) and that rounding. No other part of the entire algorithm in Figure 6.16 need be changed.

6.8 A Universal Rounding Unit

The remark of the last paragraph allows the construction of a universal rounding unit. It can be used to perform all roundings of the set $\{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}$. Any rounding can only produce one out of two different answers. It either truncates the mantissa or it enlarges it by one unit in the last place. If these two values are available the results of all roundings of the set $\{\bigtriangledown, \bigtriangleup, \square_\mu, \mu = 0(1)b\}$ can easily be performed by case selection through a simple multiplexer.



η_1
ρ
$z := (-1)^S \bigtriangleup |z|$
$z := (-1)^S \bigtriangledown |z|$
true
PN
false
0
1
cv
γ

Figure 6.26. A universal rounding unit.

Figure 6.26 gives a brief sketch of such a rounding unit. There the labels ϱ, η_1, and γ are used as defined in Figure 6.16. The intermediate unrounded result of the arithmetic operation that reaches the rounding unit is assumed to be of the form

$$z = (-1)^S \cdot 0.z_1 z_2 \cdots z_n z_{n+1} z_{n+2} \cdots \cdot b^e.$$

Thus the sign of the number is negative if $S = 1$ and it is positive if $S = 0$. The multiplexer is controlled by a logical control variable cv. We assume that the value $true$ is represented by 0 and the value $false$ by 1. The two assignments in Figure 6.26 perform the roundings $\square_0 z = (-1)^S \cdot \bigtriangleup |z|$ and $\square_b z = (-1)^S \cdot \bigtriangledown |z|$. See (5.2.8).

The different roundings of the set $\{\bigtriangledown, \bigtriangleup, \square_\mu, \mu = 0(1)b\}$ can be obtained by very simple values of the control variable cv. The following table displays values of cv for different roundings.

rounding	$\bigtriangledown z$	$\bigtriangleup z$	$\square_0 z$	$\square_b z$	$\square_\mu, \mu = 1(1)b-1$
cv	$\overline{S}$	S	0	1	$z_{n+1} \geq \mu$

Table 6.2. A universal rounding unit.

If b is an even number then $\square_{b/2}$ is the rounding to the nearest floating-point number. In case of the binary number system also the rounding to nearest even can be obtained by a simple expression for the control variable cv.

The simplest and fastest of all roundings under consideration is truncation, i.e., the rounding towards zero. It never produces an overflow of the mantissa. The left hand pass in the sketch of Figure 6.26 on the other hand is certainly slower. It enlarges the absolute value of the mantissa and may cause another normalization pass.

Thus a first glance at Figure 6.26 seems to reveal that the universality of the rounding unit slows down the execution of the rounding truncation. This, however, needs not to be so. If the required rounding is truncation the left hand pass in Figure 6.26 needs not to be executed at all.

In Figure 6.26 the control variable cv selects the rounding direction. There are important and very useful applications in numerical analysis where the direction of the rounding is to be selected randomly. In this case the control variable cv is chosen by a random number generator possibly by hardare.

6.9 Overflow and Underflow Treatment

For traditional fixed-point computations it is well known that all computed values must be less than unity and the problem data themselves must be preprocessed, typically by scaling, so that the computation proceeds coherently. There are analogous requirements for the use of all classical calculating devices such as analog computers, planimeters, and so forth. It is easy to forget that the same requirement exists for modern digital computers, even though they employ floating-point operations and an enormous range of representable numbers.

Indeed, a situation can occur in a floating-point computation where this representable range is exceeded. The terms *underflow* and *overflow* characterize these occurrences, and we now consider them.

We saw in Chapter 5 that real numbers x with the property

$$b^{e1-1} \le |x| \le G := 0.(b-1)(b-1)\dots(b-1)\cdot b^{e2}$$

are conveniently mapped into a floating-point system. Any associated error need not be larger than the distance between two neighboring floating-point numbers of x. Whenever in a floating-point computation a number x occurs with

$$0 < |x| < b^{e1-1} \text{ respectively } |x| > G$$

we speak of an *exponent underflow* respectively of an *exponent overflow*. Since in a floating-point computation the mantissa is always normalized, any *underflow* or *overflow* will be evident in the exponent. An attempt to represent a real number in the event of underflow or overflow in the floating-point system results in the loss of nearly all information about the number. Therefore, whenever in a floating-point computation an exponent underflow or overflow occurs this must be made known to the user. This indication makes it possible to decide what to do about it.

The algorithms discussed in this chapter show that underflow and overflow can occur in the performance of all arithmetic operations. Although such occurrences may be rather rare, they must be tested for at all times. See Figure 6.16.

A frequent source of underflow and overflow in conventional floating-point computations is multiplication, in particular in scalar products. This is absolutely unnecessary. Scalar products can always be computed exactly and without any underflow or overflow. This problem will be dealt with and ultimately solved in Chapter 8 where we shall see that the exact computation of scalar products is faster than a computation in conventional floating-point arithmetic.

While complete recovery from underflow and overflow is not possible in conventional floating-point arithmetic, several computer architectures offer special treatments to handle these situations. Two different procedures are in common use.

The first is a trial and error approach. During the preparation and programming of the problem, care is to be taken to ensure that the entire computation can be performed within the available range of numbers. The special difficulty is that it is hard to judge in advance that all intermediate results will stay within this range. It is a convenient and perhaps the most customary practice, therefore, simply to go ahead with the computation without too much analysis concerning the range of numbers that will occur. If an underflow or overflow occurs, the computer stops the computation with an error message. The user than tries a little harder and preprocesses the problem somewhat further, typically by rescaling. The computation is then rerun. Several such rescaling steps are often needed.

The second method exploits the observation that some underflows and overflows are benign with respect to further computation. As an example let us consider the case of underflow. Suppose that the sum

$$S := \sum_{i=1}^{n} u_i(x)$$

is to be calculated, leading to a result of magnitude 10^{30}. If the computation of one of the summands u_i causes an underflow, viz., $|u_i| < 10^{e1-1}$, then u_i does not influence the floating-point sum at all and can be ignored. On the premise that it is sometimes sensible to continue a computation which has underflowed or overflowed, a well-defined underflow or overflow procedure is to be invoked. It often consists of mapping the numbers x for which $|x| < b^{e1-1}$ to zero, and those for which $|x| > G$ to $-G$ or $-\infty$, or to $+G$ or $+\infty$ depending on the sign of the mantissa, and continue with the computation. Additionally, a detailed underflow or overflow message is given to the user. At the end of the computation the user can then decide whether the underflow or overflow that occurred during the calculation was harmless or not. In the former case the computation is finished. In the latter, a better scaling of the problem is required, followed by recomputation.

Each of these two recovery processes has advantages and disadvantages. The first may cause too many interruptions, especially when the interruptions are unnecessary from the point of view of the second method. The second method may waste computer time persisting with a computation that is afterwards rejected as incorrect.

For completeness we finish by giving the flow diagram for the part of Figure 6.16 between the labels denoted by ϵ and ρ.

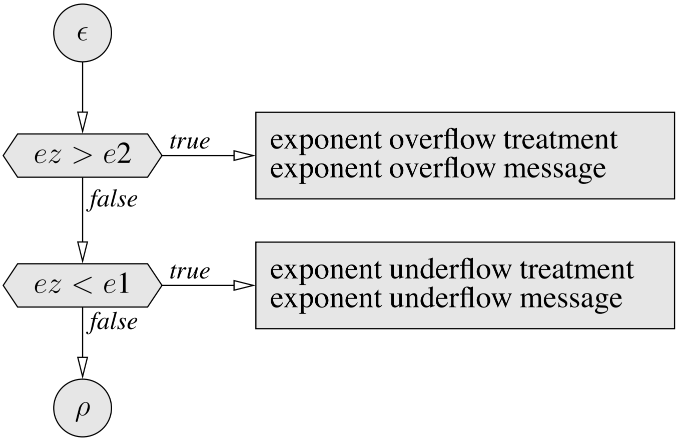

ϵ
$ez > e2$
true
exponent overflow treatment
exponent overflow message
false
$ez < e1$
true
exponent underflow treatment
exponent underflow message
false
ρ

Figure 6.27. Exponent underflow and exponent overflow.

6.10 Algorithms Using the Short Accumulator

In the case $ex - ey \leq r + 1$ of the algorithm for the addition shown in Figure 6.18, the mantissa my may be shifted to the right as many as $r + 1$ digits. Thus the ensuing addition, as described above, requires an accumulator of $2r + 1$ digits after the point. The multiplication algorithm commences with a correct calculation of the product of the mantissas $mx \cdot my$. As with addition, this operation can be conveniently accommodated within an accumulator of $2r + 1$ digits after the point. However, we mentioned at the beginning of this chapter that all floating-point operations can also be performed within shorter accumulators. In this case extra care must be taken so that the operations are executed appropriately, i.e., so that formula (RG) holds.

Figure 6.28 displays a flow chart for the execution of addition using the short accumulator. See Figure 6.17. This accumulator consists of $r+2$ digits of base b followed by an additional bit as well as an additional bit in front of the point. This latter bit is used for treatment of mantissa overflow during addition. The contents of the accumulator which follow the point are denoted by

$$.d_1 d_2 \dots d_r d_{r+1} d_{r+2} d.$$

Here the d_i are digits of base b, while d is a binary digit. To permit a simple description of the algorithms corresponding to the flow charts in Figures 6.28 and 6.29, we take the $(l+3)$th digit of the accumulator to be a full digit of base b. It could, however, be replaced by a single binary digit as indicated here.

The branches $ex - ey \geq r+2$ in Figures 6.18 and 6.28 are the same. The complementary branch $ex - ey \leq r+1$ in Figure 6.28 is split by the condition $ex - ey > 2$. If $ex - ey \leq 2$, addition with the short accumulator proceeds as before. The condition $ex - ey > 2 \wedge ex - ey \leq r+1$ specifies that branch of the algorithm of Figure 6.28 in which addition is executed differently than in the algorithm of Figure 6.18. In this case, $|mz| = |mx + my| \geq ||mx| - |my|| > b^{-1} - b^{-2} = (b-1)b^{-2} \geq b^{-2}$, i.e., the first nonzero digit of mz occurs within the first or second place after the point. Then for the roundings $\square_\mu$, $1 \leq \mu \leq b-1$, the digit of mz, which keys the rounding, is no further than the $(r+2)$th place to the right of the point. However, for the roundings ∇, $\triangle$, $\square_0$ or $\square_b$, a digit beyond the $(r+2)$th can influence the result. Then during any shifting of my to the right, note must be taken whether digits shifted beyond the $(r+2)$th place are zero or not. This information is recorded in the $(r+3)$th digit d. Thus d need only be a binary digit. The long condition in Figure 6.28 can be realized by an or-gate.

As before, the addition $mz := mx + my$ may cause an overflow in the mantissa. This would require a right shift within the normalization algorithm. The $(r+3)$th digit may be altered as a result of this right shift. Figure 6.29 displays the algorithm for the execution of the normalization using the short accumulator.

To simplify the description of the algorithms in Figures 6.28 and 6.29, we assume that the $(r+3)$th digit after the point is a full digit of base b. In fact as we have already remarked, it suffices to treat this digit as a binary digit. As in Figure 6.19, in Figure 6.29 the number of leading zeros if a left shift is necessary is denoted by l.

We illustrate the algorithms of Figures 6.28 and 6.29 with several examples. We take $r = 4$ and use the decimal system $b = 10$. The provisional and possibly unnormalized and unrounded result is denoted by $x \tilde{+} y$. We begin with the following case.

Examples. (a) $\text{sign}(x) = \text{sign}(y)$.

(1) $x = 0.9999 \cdot b^3$, $y = 0.1001 \cdot b^0$,

$$\begin{aligned}
x + y &= 1.0000|001 \cdot b^3, \\
x \tilde{+} y &= 0.1000|001 \cdot b^4, \\
\nabla(x+y) &= \nabla(x \tilde{+} y) = 0.1000 \cdot b^4, \\
\triangle(x+y) &= \triangle(x \tilde{+} y) = 0.1001 \cdot b^4, \\
\square_\mu(x+y) &= \square_\mu(x \tilde{+} y) = 0.1000 \cdot b^4, \ \text{for } 1 \leq \mu \leq 9.
\end{aligned}$$

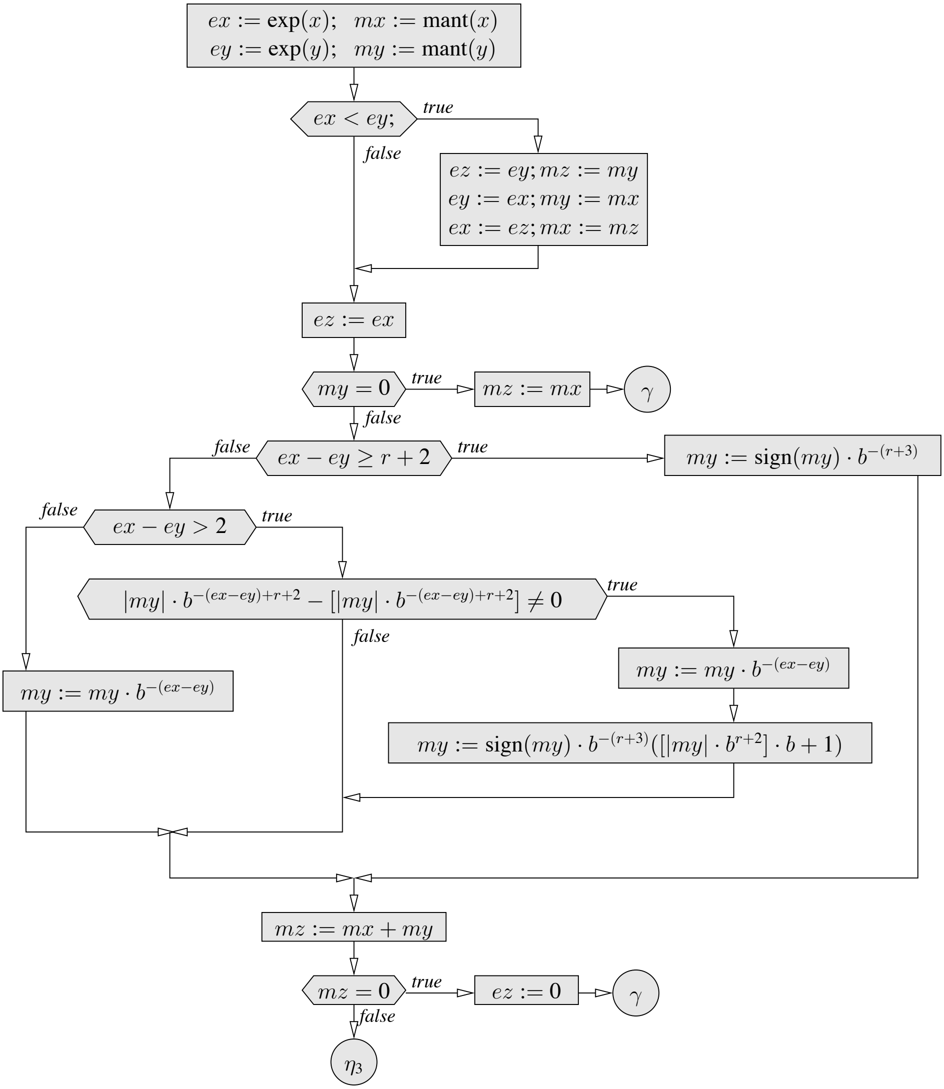

$ex := \exp(x)$; $mx := \text{mant}(x)$
$ey := \exp(y)$; $my := \text{mant}(y)$
$ex < ey$;
true
false
$ez := ey; mz := my$
$ey := ex; my := mx$
$ex := ez; mx := mz$
$ez := ex$
$my = 0$
true
$mz := mx$
γ
false
$ex - ey \geq r + 2$
true
false
$my := \text{sign}(my) \cdot b^{-(r+3)}$
$ex - ey > 2$
true
false
$|my| \cdot b^{-(ex-ey)+r+2} - [|my| \cdot b^{-(ex-ey)+r+2}] \neq 0$
true
false
$my := my \cdot b^{-(ex-ey)}$
$my := my \cdot b^{-(ex-ey)}$
$my := \text{sign}(my) \cdot b^{-(r+3)}([|my| \cdot b^{r+2}] \cdot b + 1)$
$mz := mx + my$
$mz = 0$
true
$ez := 0$
γ
false
η_3

Figure 6.28. Execution of addition with the short accumulator.

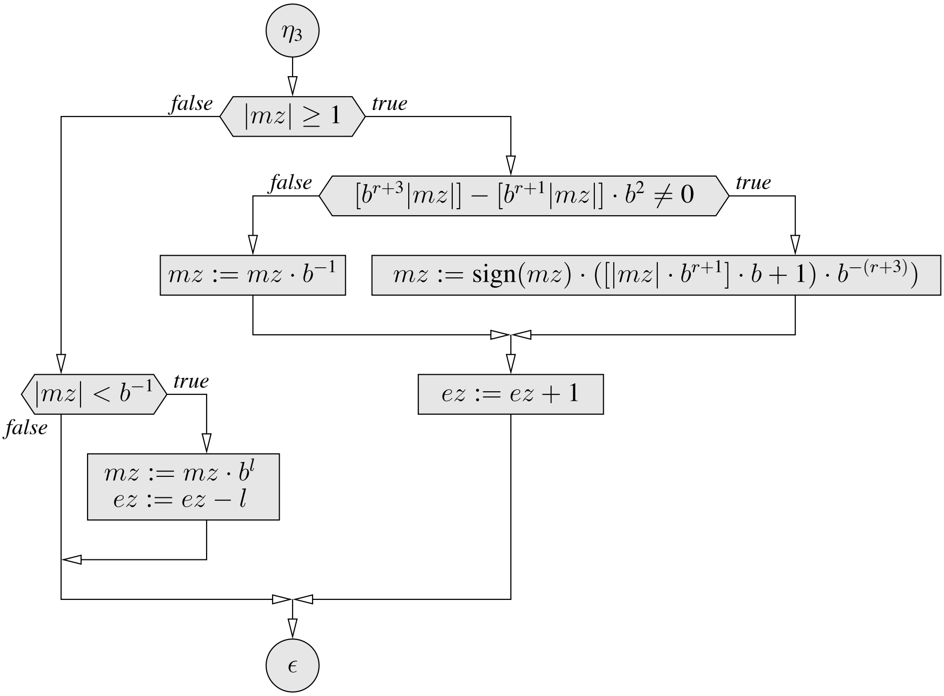


Figure 6.29. Execution of normalization with the short accumulator.

(2) $x = 0.9998 \cdot b^3$, $y = 0.2070 \cdot b^0$,

$$\begin{aligned} x + y &= 1.0000|070 \cdot b^3, \\ x \,\tilde{+}\, y &= 0.1000|001 \cdot b^4, \\ \triangledown(x+y) &= \triangledown(x \,\tilde{+}\, y) = 0.1000 \cdot b^4, \\ \vartriangle(x+y) &= \vartriangle(x \,\tilde{+}\, y) = 0.1001 \cdot b^4, \\ \square_\mu(x+y) &= \square_\mu(x \,\tilde{+}\, y) = 0.1000 \cdot b^4, \text{ for } 1 \le \mu \le 9. \end{aligned}$$

(3) $x = 0.8234 \cdot b^5$, $y = 0.5012 \cdot b^1$,

$$\begin{aligned} x + y &= 0.8234|5012 \cdot b^5, \\ x \,\tilde{+}\, y &= 0.8234|501 \cdot b^5, \\ \triangledown(x+y) &= \triangledown(x \,\tilde{+}\, y) = 0.8234 \cdot b^5, \\ \vartriangle(x+y) &= \vartriangle(x \,\tilde{+}\, y) = 0.8235 \cdot b^5, \\ \square_\mu(x+y) &= \square_\mu(x \,\tilde{+}\, y) = 0.8235 \cdot b^5, \text{ for } 1 \le \mu \le 5, \\ \square_\mu(x+y) &= \square_\mu(x \,\tilde{+}\, y) = 0.8234 \cdot b^5, \text{ for } 6 \le \mu \le 9. \end{aligned}$$

(4) $x = -0.8234 \cdot b^5$, $y = -0.5021 \cdot b^1$,

$$\begin{aligned} x + y &= -0.8234|5021 \cdot b^5, \\ x \,\tilde{+}\, y &= -0.8234|501 \cdot b^5, \\ \triangledown(x+y) &= \triangledown(x \,\tilde{+}\, y) = -0.8235 \cdot b^5, \\ \vartriangle(x+y) &= \vartriangle(x \,\tilde{+}\, y) = -0.8234 \cdot b^5, \\ \square_\mu(x+y) &= \square_\mu(x \,\tilde{+}\, y) = -0.8235 \cdot b^5, \text{ for } 1 \le \mu \le 5, \\ \square_\mu(x+y) &= \square_\mu(x \,\tilde{+}\, y) = -0.8234 \cdot b^5, \text{ for } 6 \le \mu \le 9. \end{aligned}$$

(5) $x = -0.9998 \cdot b^3$, $y = -0.2070 \cdot b^0$,

$$\begin{aligned} x + y &= -1.0000|070 \cdot b^3, \\ x \,\tilde{+}\, y &= -0.1000|001 \cdot b^4, \\ \triangledown(x+y) &= \triangledown(x \,\tilde{+}\, y) = -0.1001 \cdot b^4, \\ \vartriangle(x+y) &= \vartriangle(x \,\tilde{+}\, y) = -0.1000 \cdot b^4, \\ \square_\mu(x+y) &= \square_\mu(x \,\tilde{+}\, y) = -0.1000 \cdot b^4, \text{ for } 1 \le \mu \le 9. \end{aligned}$$

(b) Now let $\text{sign}(x) \neq \text{sign}(y)$.

(6) $x = 0.1000 \cdot b^4$, $y = -0.1001 \cdot b^{-1}$,

$$\begin{aligned}
x + y &= 0.0999|98999 \cdot b^4,\\
x \mathbin{\tilde{+}} y &= 0.0999|989 \cdot b^4,\\
\bigtriangledown(x + y) = \bigtriangledown(x \mathbin{\tilde{+}} y) &= 0.9999 \cdot b^3,\\
\bigtriangleup(x + y) = \bigtriangleup(x \mathbin{\tilde{+}} y) &= 0.1000 \cdot b^4,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.9999 \cdot b^3, \text{ for } \mu = 9,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.1000 \cdot b^4, \text{ for } 1 \leq \mu \leq 8.
\end{aligned}$$

(7) $x = 0.1000 \cdot b^6$, $y = -0.5001 \cdot b^1$,

$$\begin{aligned}
x + y &= 0.0999|94999 \cdot b^6,\\
x \mathbin{\tilde{+}} y &= 0.0999|949 \cdot b^6,\\
\bigtriangledown(x + y) = \bigtriangledown(x \mathbin{\tilde{+}} y) &= 0.9999 \cdot b^5,\\
\bigtriangleup(x + y) = \bigtriangleup(x \mathbin{\tilde{+}} y) &= 0.1000 \cdot b^6,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.9999 \cdot b^5, \text{ for } 5 \leq \mu \leq 9,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.1000 \cdot b^6, \text{ for } 1 \leq \mu \leq 4.
\end{aligned}$$

(8) $x = 0.1000 \cdot b^6$, $y = -0.5000 \cdot b^1$,

$$\begin{aligned}
x + y &= 0.0999|95000 \cdot b^6,\\
x \mathbin{\tilde{+}} y &= 0.0999|950 \cdot b^6,\\
\bigtriangledown(x + y) = \bigtriangledown(x \mathbin{\tilde{+}} y) &= 0.9999 \cdot b^5,\\
\bigtriangleup(x + y) = \bigtriangleup(x \mathbin{\tilde{+}} y) &= 0.1000 \cdot b^6,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.1000 \cdot b^6, \text{ for } 1 \leq \mu \leq 5,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.9999 \cdot b^6, \text{ for } 6 \leq \mu \leq 9.
\end{aligned}$$

(9) $x = 0.1000 \cdot b^4$, $y = -0.5412 \cdot b^0$,

$$\begin{aligned}
x + y &= 0.0999|4588 \cdot b^4,\\
x \mathbin{\tilde{+}} y &= 0.0999|459 \cdot b^4,\\
\bigtriangledown(x + y) = \bigtriangledown(x \mathbin{\tilde{+}} y) &= 0.9994 \cdot b^3,\\
\bigtriangleup(x + y) = \bigtriangleup(x \mathbin{\tilde{+}} y) &= 0.9995 \cdot b^3,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.9995 \cdot b^3, \text{ for } 1 \leq \mu \leq 5,\\
\square_\mu(x + y) = \square_\mu(x \mathbin{\tilde{+}} y) &= 0.9994 \cdot b^3, \text{ for } 6 \leq \mu \leq 9.
\end{aligned}$$

(10) $x = -0.1000 \cdot b^6, y = 0.5001 \cdot b^1$,

$$\begin{aligned}
x + y &= -0.0999|94999 \cdot b^6,\\
x \tilde{+} y &= -0.0999|949 \cdot b^6,\\
\triangledown(x + y) &= \triangledown(x \tilde{+} y) = -0.1000 \cdot b^6,\\
\triangle(x + y) &= \triangle(x \tilde{+} y) = -0.9999 \cdot b^5,\\
\square_\mu(x + y) &= \square_\mu(x \tilde{+} y) = -0.1000 \cdot b^6, \text{ for } 1 \le \mu \le 4,\\
\square_\mu(x + y) &= \square_\mu(x \tilde{+} y) = -0.9999 \cdot b^5, \text{ for } 5 \le \mu \le 9.
\end{aligned}$$

(11) $x = -0.8234 \cdot b^4, y = 0.5021 \cdot b^0$,

$$\begin{aligned}
x + y &= -0.8233|4979 \cdot b^4,\\
x \tilde{+} y &= -0.8233|499 \cdot b^4,\\
\triangledown(x + y) &= \triangledown(x \tilde{+} y) = -0.8234 \cdot b^4,\\
\triangle(x + y) &= \triangle(x \tilde{+} y) = -0.8233 \cdot b^4,\\
\square_\mu(x + y) &= \square_\mu(x \tilde{+} y) = -0.8234 \cdot b^4, \text{ for } 1 \le \mu \le 4,\\
\square_\mu(x + y) &= \square_\mu(x \tilde{+} y) = -0.8233 \cdot b^4, \text{ for } 5 \le \mu \le 9.
\end{aligned}$$

The next example confirms that an accumulator of $r + 1$ digits followed by an additional binary digit d after the point cannot deliver correct results as defined by (RG) in all cases.

(12) $x = 0.1000 \cdot b^6, y = -0.5001 \cdot b^1$,

$$\begin{aligned}
x + y &= 0.0999|94999 \cdot b^6,\\
x \tilde{+} y &= 0.0999|99 \cdot b^6,\\
\square_\mu(x + y) &= 0.9999 \cdot b^5 \neq 0.1000 \cdot b^6 = \square_\mu(x \tilde{+} y), \text{ for } 5 \le \mu \le 9.
\end{aligned}$$

The preceding study shows that for addition and subtraction, the length of the short accumulator as shown in Figure 6.17(b) is both necessary and sufficient to realize formula (RG) for all roundings in the set $\{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$.

The short accumulator is also sufficient to perform the multiplication and division defined by (RG) for all of these roundings. We give a simple numerical example for multiplication:

$$\begin{array}{ccccccccl}
\multicolumn{8}{r}{0.9\,9\,9\,9 \cdot 0.9\,9\,9\,9} & \\ \hline
 & & & 8 & 9 & 9 & 9 & 1 & \\
 & & 8 & 9 & 9 & 9 & 1 & & \\ \cline{3-8}
 & & 9 & 8 & 9 & 9 & 0| & 1 & r+2 \text{ digits} \\
 & 8 & 9 & 9 & 9 & 1 & | & & \\ \cline{2-8}
 & 9 & 9 & 8 & 9 & 0| & 0 & & r+2 \text{ digits} \\
8 & 9 & 9 & 9 & 1 & | & & & \\ \cline{1-8}
9 & 9 & 9 & 8 & 0 & 0 & & & r+2 \text{ digits} \\
 & & & & & & & & \\
\multicolumn{8}{r}{0.9\,9\,9\,8\,0\,0\,0\,1} &
\end{array}$$

Proceeding in a straightforward manner, we can show that the short accumulator is sufficient to perform the division (defined by (RG)) of two floating-point numbers and with any one of the roundings $\{\nabla, \triangle, \square_\mu, \mu = 0(1)b\}$.

At the end of Section 6.5 where multiplication was considered, we mentioned that every digital processor should also be able to deliver the double length unrounded product. Such double length products cannot be efficiently produced by the short accumulator. This consideration gives very high priority to the choice of at least the long accumulator for floating-point operations.

Nevertheless, our study of the short accumulator is useful. In numerical analysis computation of the sum of two products $a \cdot b + c \cdot d$ of floating-point numbers is a frequent task which is often very critical numerically. It appears, for instance, as the determinant of a two by two matrix, or in the computation of the real and imaginary parts of the product of two complex numbers, or in the absolute value of a two dimensional vector. The results of this section show that such problems can be solved with very high accuracy.

If the processor delivers the products to the full double length an expression of the form $a \cdot b + c \cdot d$ can be computed with just one rounding $\square(a \cdot b + c \cdot d)$, $\square \in \{\nabla, \triangle, \square_\mu, \mu = 0(1)b\}$, with an error less than or equal to b^{-2r}, by an accumulator as shown in Figure 6.30. Here the products $a \cdot b$ and $c \cdot d$ are of length $2r$. The accumulator shown in Figure 6.30 is the short accumulator for floating-point numbers with a mantissa of $2r$ digits.

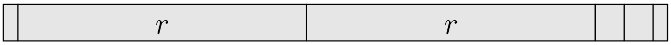


Figure 6.30. Accumulator for highly accurate computation of $\square(a{\cdot}b{+}c{\cdot}d)$.

6.11 The Level 2 Operations

This section contains a brief discussion of all arithmetic operations defined in the sets of Figure 1 as well as between these sets. The basic ideas of these operations have been extensively studied in Chapters 3 and 4. Thus our objective is simply to summarize the definition of these operations and to point out that they can be performed by using the operations that have been discussed earlier in this chapter. We do not derive algorithms for these operations because they are simple.

We consider the five basic data sets (types) $\mathbb{Z}$, S, $\mathbb{C}S$, IS, $I\mathbb{C}S$. Here $\mathbb{Z}$ denotes the bounded set of integers representable on the computer. In an appropriate programming language they may be called `integer`, `real`, `complex`, `real_interval`, `complex_interval`. We assume that integer arithmetic in $\mathbb{Z}$ is available and briefly repeat the definition of the arithmetic operations in the remaining basic data sets, beginning with S.

(a) The arithmetic in S (real). We assumed throughout the preceding chapters that floating-point arithmetic in S is defined by semimorphism, i.e., by means of the formula

(RG) $\bigwedge_{a,b\in S} a \boxed{\circ} b := \square(a\circ b)$, for $\circ \in \{+,-,\cdot,/\}$ with $b \neq 0$ for $\circ = /$,

where $\square$ denotes one of the monotone and antisymmetric roundings $\square_\mu$, $\mu = 0(1)b$. All of these roundings, as well as the operations (RG) have been discussed earlier in this chapter. Division is not defined for $b = 0$.

(b) The arithmetic in $\mathbb{C}S$ is also defined by semimorphism,

(RG) $\bigwedge_{(a,b),(c,d)\in\mathbb{C}S} (a,b) \boxed{\circ} (c,d) := \square((a,b)\circ(c,d))$, for $\circ \in \{+,-,\cdot,/\}$,

where the rounding $\square : \mathbb{C} \to \mathbb{C}S$ is defined by

$$\bigwedge_{(a,b)\in\mathbb{C}} \square(a,b) := (\square a, \square b).$$

Once again the rounding on the right-hand side denotes one of the roundings $\square_\mu$, $\mu = 0(1)b$. This leads to the following explicit representation of the operations for all couples of elements $(a,b),(c,d) \in \mathbb{C}S$:

$$\begin{aligned}
(a,b) \boxplus (c,d) &= (a \boxplus c, b \boxplus d),\\
(a,b) \boxminus (c,d) &= (a \boxminus c, b \boxminus d),\\
(a,b) \boxdot (c,d) &= (\square(ac-bd), \square(ad+bc)),\\
(a,b) \boxed{/} (c,d) &= \left(\square\left(\frac{ac+bd}{c^2+d^2}\right), \square\left(\frac{bc-ad}{c^2+d^2}\right)\right).
\end{aligned}$$

Division is not defined if $c = d = 0$.

Thus addition and subtraction can be executed in terms of the corresponding operations in S. The products occurring on the right-hand side of the multiplication formula lead to floating-point numbers of length $2r$. The summation and the rounding appearing in the formula for multiplication can, for instance, be performed by routines for the computation of exact scalar products which will be discussed in Chapter 8. An alternative and simpler method has already been indicated at the end of the last section. See Figure 6.30.

For the computation of the quotient several methods are available. One method is simply to apply an algorithm for evaluation of arithmetic expressions with maximum accuracy. See Section 9.6. Another method has been developed by [151, 152]. A third method could proceed in the following way. In the quotient formula, expressions of the form

$$q := (x_1y_1 + x_2y_2)/(u_1v_1 + u_2v_2)$$

occur with $x_i, y_i, u_i, v_i \in S$. First compute the products x_iy_i and u_iv_i to the full length of $2r$ digits. Then compute $r{+}8$ digit approximations $\tilde{n}$ and $\tilde{d}$ for the numerator and denominator of the quotient q. With these expressions an approximation $\tilde{q} := \tilde{n}/\tilde{d}$ of $r + 5$ digits for the exact quotient can be obtained. Then in most cases $\square\tilde{q} = \square q$ for all roundings $\square \in \{\triangledown, \vartriangle, \square_\mu, \mu = 0(1)b\}$. If $\square\tilde{q} \neq \square q$, the correct result $\square q$ can be determined from $\tilde{q}$, and the residual

$$R := (u_1v_1 \cdot \tilde{q} + u_2v_2 \cdot \tilde{q} - x_1y_1 - x_2y_2).$$

Since cancellation occurs in the computation of this expression, the exact scalar product (see Chapter 8) has to be applied for its evaluation.

The order relation $\leq$ in $\mathbb{C}S$ is defined in terms of the order relation in S:

$$(a, b) \leq (c, d) :\Leftrightarrow a \leq b \wedge c \leq d.$$

(c) The arithmetic in IS is defined by the semimorphism

(RG) $\bigwedge_{A,B \in IS} A \mathbin{\Diamond\mkern-13mu\circ} B := \Diamond(A \mathbin{\square\mkern-13mu\circ} B)$, for $\circ \in \{+, -, \cdot, /\}$,

with the monotone upwardly directed rounding $\Diamond : I\mathbb{R} \to IS$.

In Chapter 4 we derived the following formulas for the execution of (RG) for all couples of intervals $[a, b]$, $[c, d] \in IS$:

$$\begin{aligned}
[a, b] \mathbin{\Diamond\mkern-13mu+} [c, d] &= [a \mathbin{\triangledown\mkern-13mu+} c, b \mathbin{\vartriangle\mkern-13mu+} d],\\
[a, b] \mathbin{\Diamond\mkern-13mu-} [c, d] &= [a \mathbin{\triangledown\mkern-13mu-} d, b \mathbin{\vartriangle\mkern-13mu-} c],\\
[a, b] \mathbin{\Diamond\mkern-11mu\cdot} [c, d] &= [\min\{a \mathbin{\triangledown\mkern-11mu\cdot} c, a \mathbin{\triangledown\mkern-11mu\cdot} d, b \mathbin{\triangledown\mkern-11mu\cdot} c, b \mathbin{\triangledown\mkern-11mu\cdot} d\},\\
&\qquad \max\{a \mathbin{\vartriangle\mkern-11mu\cdot} c, a \mathbin{\vartriangle\mkern-11mu\cdot} d, b \mathbin{\vartriangle\mkern-11mu\cdot} c, b \mathbin{\vartriangle\mkern-11mu\cdot} d\}],\\
[a, b] \mathbin{\Diamond\mkern-13mu/} [c, d] &= [\min\{a \mathbin{\triangledown\mkern-13mu/} c, a \mathbin{\triangledown\mkern-13mu/} d, b \mathbin{\triangledown\mkern-13mu/} c, b \mathbin{\triangledown\mkern-13mu/} d\},\\
&\qquad \max\{a \mathbin{\vartriangle\mkern-13mu/} c, a \mathbin{\vartriangle\mkern-13mu/} d, b \mathbin{\vartriangle\mkern-13mu/} c, b \mathbin{\vartriangle\mkern-13mu/} d\}].
\end{aligned}$$

Here division is not defined if $0 \in [c, d]$. In the multiplication and division formulas, the minimum and maximum can be determined by using the Tables 4.5 and 4.6 in Section 4.6. Division by an interval that includes zero is described in the Tables 4.9 and 4.10 of Section 4.9.

The order relations $\leq$ and $\subseteq$ in *IS* are defined by

$$[a,b] \leq [c,d] :\Leftrightarrow a \leq c \wedge b \leq d,$$
$$[a,b] \subseteq [c,d] :\Leftrightarrow c \leq a \wedge b \leq d.$$

Here the relation $\leq$ on the right-hand side denotes the order relation in S.

In addition to the arithmetic operations in *IS*, the intersection $\cap$ and the interval hull $\bar{\cup}$ have to be made available:

$$[a,b] \cap [c,d] :\Leftrightarrow [\max\{a,c\}, \min\{b,d\}],$$
$$[a,b] \,\bar{\cup}\, [c,d] :\Leftrightarrow [\min\{a,c\}, \max\{b,d\}].$$

If $b < c$ or $d < a$ the intersection is the empty set.

(d) The arithmetic in $I\mathbb{C}S$ is also defined by semimorphism

(RG) $\bigwedge_{\Phi,\Psi \in I\mathbb{C}S} \quad \Phi \mathbin{\Diamond\kern-0.62em{\scriptstyle\circ}\kern0.25em} \Psi := \Diamond(\Phi \mathbin{\square\kern-0.6em{\scriptstyle\circ}\kern0.2em} \Psi)$, for $\circ \in \{+,-,\cdot,/\}$,

where $\Diamond : I\mathbb{C} \to I\mathbb{C}S$ denotes the upwardly directed rounding. We showed in Chapter 4 that these operations are isomorphic to the operations in $\mathbb{C}$*IS* as far as addition, subtraction, and multiplication are concerned. See also Figure 4.5 in Chapter 4. The division in $\mathbb{C}$*IS* delivers upper bounds for the quotient in $I\mathbb{C}S$.

Thus for the execution of the above semimorphism, we obtain the following formulas for all couples of elements $(A,B),(C,D) \in \mathbb{C}IS$ with $A = [a_1,a_2]$, $B = [b_1,b_2]$, $C = [c_1,c_2], D = [d_1,d_2] \in IS$:

$$\begin{aligned}
(A,B) \mathbin{\Diamond\kern-0.62em{\scriptstyle+}\kern0.25em} (C,D) &= (A \mathbin{\Diamond\kern-0.62em{\scriptstyle+}\kern0.25em} C, B \mathbin{\Diamond\kern-0.62em{\scriptstyle+}\kern0.25em} D)\\
&= \big([a_1 \mathbin{\bigtriangledown\kern-0.72em{\scriptstyle+}\kern0.3em} c_1, a_2 \mathbin{\bigtriangleup\kern-0.72em{\scriptstyle+}\kern0.3em} c_2], [b_1 \mathbin{\bigtriangledown\kern-0.72em{\scriptstyle+}\kern0.3em} d_1, b_2 \mathbin{\bigtriangleup\kern-0.72em{\scriptstyle+}\kern0.3em} d_2]\big),\\
(A,B) \mathbin{\Diamond\kern-0.62em{\scriptstyle-}\kern0.25em} (C,D) &= (A \mathbin{\Diamond\kern-0.62em{\scriptstyle-}\kern0.25em} C, B \mathbin{\Diamond\kern-0.62em{\scriptstyle-}\kern0.25em} D)\\
&= \big([a_1 \bigtriangledown c_2, a_2 \bigtriangleup c_1], [b_1 \bigtriangledown d_2, b_2 \bigtriangleup d_1]\big),\\
(A,B) \mathbin{\Diamond\kern-0.62em{\scriptstyle\cdot}\kern0.25em} (C,D) &= (\Diamond(A \boxdot C \boxminus B \boxdot D), \Diamond(A \boxdot D \boxplus B \boxdot C))\\
&= \Big(\big[\bigtriangledown(\min_{i,j=1,2}\{a_i c_j\} - \max_{i,j=1,2}\{b_i d_j\}),\\
&\qquad \bigtriangleup(\max_{i,j=1,2}\{a_i c_j\} - \min_{i,j=1,2}\{b_i d_j\})\big],\\
&\qquad \big[\bigtriangledown(\min_{i,j=1,2}\{a_i d_j\} + \min_{i,j=1,2}\{b_i c_j\}),\\
&\qquad \bigtriangleup(\max_{i,j=1,2}\{a_i d_j\} + \max_{i,j=1,2}\{b_i c_j\})\big]\Big),\\
(A,B) \mathbin{\Diamond\kern-0.62em{\scriptstyle/}\kern0.25em} (C,D) &= \big(\Diamond((A \boxdot C \boxplus B \boxdot D) \boxslash (C \boxdot C \boxplus D \boxdot D)),\\
&\qquad \Diamond((B \boxdot C \boxminus A \boxdot D) \boxslash (C \boxdot C \boxplus D \boxdot D))\big).
\end{aligned}$$

The division is defined only if $0 \notin C \boxdot C \boxplus D \boxdot D$. The operation $\boxed{\circ}$, $\circ \in \{+, -, \cdot, /\}$, on the right-hand side of the last two formulas denote operations in $I\mathbb{R}$. For the execution of $\boxdot$ and $\boxed{/}$, see Chapter 4, Tables 4.1 and 4.2, and Corollary 4.11.

The order relations $\leq$ and $\subseteq$ in $\mathbb{C}IS$ are defined by

$$(A, B) \leq (C, D) :\Leftrightarrow A \leq C \wedge B \leq D,$$
$$(A, B) \subseteq (C, D) :\Leftrightarrow A \subseteq C \wedge B \subseteq D.$$

Here the relations $\leq$ and $\subseteq$ on the right-hand side are those in IS.

The intersection $\cap$ and the interval hull $\bar{\cup}$ in $\mathbb{C}IS$ are defined componentwise:

$$(A, B) \cap (C, D) :\Leftrightarrow (A \cap C, B \cap D),$$
$$(A, B) \,\bar{\cup}\, (C, D) :\Leftrightarrow (A \,\bar{\cup}\, C, B \,\bar{\cup}\, D).$$

Here the operations $\cap$ and $\bar{\cup}$ on the right-hand side are those in IS.

We have now defined inner arithmetic operations and order relations for all the five basic data sets (types) $\mathbb{Z}$, S, $\mathbb{C}S$, IS, $\mathbb{C}IS$. We still have to define operations and relations between elements of different sets.

For addition, subtraction and multiplication, the types of the result of the operation are given in Table 6.3. The Figure also describes the result of division with the one exception that the quotient of two integers is defined to be of type S.

$a \diagdown b$	$\mathbb{Z}$	S	$\mathbb{C}S$	IS	$\mathbb{C}IS$
$\mathbb{Z}$	$\mathbb{Z}$	S	$\mathbb{C}S$	IS	$\mathbb{C}IS$
S	S	S	$\mathbb{C}S$	–	–
$\mathbb{C}S$	$\mathbb{C}S$	$\mathbb{C}S$	$\mathbb{C}S$	–	–
IS	IS	–	–	IS	$\mathbb{C}IS$
$\mathbb{C}IS$	$\mathbb{C}IS$	–	–	$\mathbb{C}IS$	$\mathbb{C}IS$

Table 6.3. Resulting type of operations among the basic data sets.

The general rule for performing the operations upon operands of different types is to lift the operand of simpler type into the type of the other operand by means of a type transfer. Then the operation can be executed as one of the inner operations, all of which we have already dealt with. If, for instance, a multiplication $a \cdot b$ has to be performed, where $a \in S$ and $B \in \mathbb{C}S$, we transfer a into an element of $\mathbb{C}S$ by adjoining an imaginary part that is zero. Then we multiply the two elements of $\mathbb{C}S$. Or if, for instance, an addition $a + b$ has to be performed with $a \in \mathbb{Z}$ and $b \in IS$, we first transfer a into a floating-point number of S and then this number into the interval $[a, a] \in IS$. Then the addition in IS can be used to execute the operation.

A dash in the Table 6.3 means that it is not reasonable to define the corresponding operation $a \circ b$ a priori. From our point of view, a floating-point number of S, for instance, is an approximate representation of a real number, while an interval is a precisely defined object. The product of the two, which ought to be an interval, may then not be precisely specified. However, if the user does indeed need, for instance, to multiply a floating-point number of S and a floating-point interval of IS, he may do so by employing a transfer function – which may be predefined – which transforms the floating-point operand of S into a floating-point interval of IS. However, in doing so, he should be aware of the possible loss of meaning of the interval as a precise bound. This requirement on the part of the user to invoke explicitly the transfer function in these cases is intended to alert him to questions of accuracy in the arithmetic.

Table 6.4 shows how to perform the intersection and the interval hull between operands of different basic data types whenever this is reasonable. If necessary, type lifting is also performed by automatic type transfer.

$\cap$ \ $\bar{\cup}$	IS	$\mathbb{C}IS$
IS	IS	$\mathbb{C}IS$
$\mathbb{C}IS$	$\mathbb{C}IS$	$\mathbb{C}IS$

Table 6.4. Table of intersections and interval hulls between the basic interval types.

We have now completed our definition and discussion of the inner and mixed operations for the five basic data sets. We are now going to define operations for matrices and vectors of these sets. All arithmetic operations are again defined by semimorphism.

Let T be one of the sets (types) $\mathbb{Z}$, S, $\mathbb{C}S$, IS, $\mathbb{C}IS$. We denote the set of $m \times n$ matrices with components of T by

$$M_{mn}T := \{(a_{ij}) \mid a_{ij} \in T \text{ for } i = 1(1)m \wedge j = 1(1)n\}.$$

In $M_{mn}T$ we define operations $\circ : M_{mn}T \times M_{mn}T \to M_{mn}T$ by

$$\bigwedge_{(a_{ij})(b_{ij}) \in M_{mn}T} (a_{ij}) \circ (b_{ij}) := (a_{ij} \circ b_{ij}).$$

Here the operation sign $\circ$ on the right-hand side denotes certain operations, according to the following three cases. In

Case $T = \mathbb{Z}$ one of the integer operations $+$, $-$;

Case $T = S$ or $\mathbb{C}S$ one of the rounded operations $\boxplus$, $\boxminus$;

Case $T = IS$ or $I\mathbb{C}S$ one of the operations $\diamondplus$, $\diamondminus$, $\cap$, $\bar{\cup}$.

In addition to these operations, we define outer multiplications $\cdot : T \times M_{mn}T \to M_{mn}T$ by

$$\bigwedge_{a \in T} \bigwedge_{(b_{ij}) \in M_{mn}T} a \cdot (b_{ij}) := (a \cdot b_{ij}).$$

Here the multiplication sign $\cdot$ on the right-hand side also has three realizations. In

Case $T = \mathbb{Z}$ integer multiplication;

Case $T = S$ or $\mathbb{C}S$ rounded multiplication $\boxdot$;

Case $T = IS$ or $I\mathbb{C}S$ multiplication $\mathbin{\Diamond\!\!\!\cdot\,}$.

Finally, we consider the product of matrices. It is a mapping $\cdot : M_{mn}T \times M_{np}T \to M_{mp}T$. There are five cases for definition of this multiplication:

Case $T = \mathbb{Z}$

$$\bigwedge_{(a_{ij}) \in M_{mn}Z} \bigwedge_{(b_{jk}) \in M_{np}Z} (a_{ij}) \cdot (b_{jk}) := \left(\sum_{j=1}^{n} a_{ij} \cdot b_{jk} \right).$$

Case $T = S$

$$\bigwedge_{(a_{ij}) \in M_{mn}S} \bigwedge_{(b_{jk}) \in M_{np}S} (a_{ij}) \cdot (b_{jk}) := \left(\square \sum_{j=1}^{n} a_{ij} \cdot b_{jk} \right).$$

Here $\square$ denotes one of the roundings $\square \in \{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$. If a_{ij} and b_{ij} are floating-point numbers of length r, the products $a_{ij} \cdot b_{jk}$ are of length $2r$. They are to be accumulated with a single rounding. For computation of the exact scalar product see Chapter 8.

Case $T = \mathbb{C}S$

$$\bigwedge_{((a_{ij}, b_{ij})) \in M_{mn}\mathbb{C}S} \bigwedge_{((c_{jk}, d_{jk})) \in M_{np}\mathbb{C}S} ((a_{ij}, b_{ij})) \boxdot ((c_{jk}, d_{jk}))$$
$$:= \left(\left(\square \sum_{j=1}^{n} (a_{ij}c_{jk} - b_{ij}d_{jk}), \square \sum_{j=1}^{n} (a_{ij}d_{jk} + b_{ij}c_{jk}) \right) \right).$$

Here $\square$ denotes one of the roundings $\square \in \{\triangledown, \triangle, \square_\mu, \mu = 0(1)b\}$. If a_{ij}, b_{ij}, c_{jk}, and d_{jk} are floating-point numbers of length r, the components of the product matrix can be calculated by one of the methods for the computation of the exact scalar product. See Chapter 8.

Case $T = IS$

$$\bigwedge_{([a_{ij},b_{ij}])\in M_{mn}IS} \; \bigwedge_{([c_{jk},d_{jk}])\in M_{np}IS} ([a_{ij}, b_{ij}]) \Diamond ([c_{jk}, d_{jk}])$$
$$:= \left(\left[\triangledown \sum_{j=1}^{n} \min\{a_{ij}c_{jk}, a_{ij}d_{jk}, b_{ij}c_{jk}, b_{ij}d_{jk}\}, \right.\right.$$
$$\left.\left. \triangle \sum_{j=1}^{n} \max\{a_{ij}c_{jk}, a_{ij}d_{jk}, b_{ij}c_{jk}, b_{ij}d_{jk}\} \right] \right) .$$

If the a_{ij}, b_{ij}, c_{jk}, and d_{jk} are floating-point numbers of length r, all the products in the braces are of length $2r$ and can be determined exactly. However, it is not necessary to calculate all these products, since to determine their minimum (resp. maximum), Table 4.1 in Section 4.2 is to be used. The sum can then be evaluated by one of the methods for the computation of the exact scalar product. See Chapter 8.

Case $T = \mathbb{C}IS$

For all $(([a^1_{ij}, a^2_{ij}], [b^1_{ij}, b^2_{ij}])) \in M_{mn}\mathbb{C}IS$ and $(([c^1_{jk}, c^2_{jk}], [d^1_{jk}, d^2_{jk}])) \in M_{np}\mathbb{C}IS$, we have

$$(([a^1_{ij}, a^2_{ij}],[b^1_{ij}, b^2_{ij}])) \Diamond (([c^1_{jk}, c^2_{jk}], [d^1_{jk}, d^2_{jk}]))$$
$$:= \left(\left(\Diamond \boxed{\sum_{j=1}^{n}} \left([a^1_{ij}, a^2_{ij}] \boxdot [c^1_{jk}, c^2_{jk}] \boxminus [b^1_{ij}, b^2_{ij}] \boxdot [d^1_{jk}, d^2_{jk}] \right) , \right.\right.$$
$$\left.\left. \Diamond \boxed{\sum_{j=1}^{n}} \left([a^1_{ij}, a^2_{ij}] \boxdot [d^1_{jk}, d^2_{jk}] \boxplus [b^1_{ij}, b^2_{ij}] \boxdot [c^1_{jk}, c^2_{jk}] \right) \right) \right)$$
$$= \left(\left(\left[\triangledown \sum_{j=1}^{n} \left(\min_{s,t=1,2}\{a^s_{ij}c^t_{jk}\} - \max_{s,t=1,2}\{b^s_{ij}d^t_{jk}\} \right), \right.\right.\right.$$
$$\left. \triangle \sum_{j=1}^{n} \left(\max_{s,t=1,2}\{a^s_{ij}c^t_{jk}\} - \min_{s,t=1,2}\{b^s_{ij}d^t_{jk}\} \right) \right],$$
$$\left[\triangledown \sum_{j=1}^{n} \left(\min_{s,t=1,2}\{a^s_{ij}d^t_{jk}\} + \min_{s,t=1,2}\{b^s_{ij}c^t_{jk}\} \right), \right.$$
$$\left.\left.\left. \triangle \sum_{j=1}^{n} \left(\max_{s,t=1,2}\{a^s_{ij}d^t_{jk}\} + \max_{s,t=1,2}\{b^s_{ij}c^t_{jk}\} \right) \right] \right) \right).$$

As before, the products occurring in this expression are of length $2r$. To determine their minimum and maximum, Table 4.1 in Section 4.2 can be used. The sums and differences can then be evaluated by applying one of the methods for the computation of the exact scalar product. See Chapter 8.

If T denotes one of the sets (types) $\mathbb{Z}$, S, $\mathbb{C}S$, IS, $\mathbb{C}IS$, we denote the set of vectors with components in T by

$$V_nT := \{(a_i) \mid a_i \in T \text{ for } i = 1(1)n\}$$

and the set of transposed vectors with components in T by

$$V_n'T := \{(a_i)' \mid a_i \in T \text{ for } i = 1(1)n\}.$$

The operations defined above for matrices are directly extended to vectors if we identify the sets

$$V_nT \equiv M_{n1}T, \quad V_n'T \equiv M_{1n}T, \quad T \equiv M_{11}T.$$

The relevant operators, as well as the types of the results, are shown in Table 6.5.

$\circ$	T	$M_{mn}T$	V_nT	$V_n'T$
T	T	–	–	–
$M_{mn}T$	–	$M_{mn}T$	–	–
V_nT	–	–	V_nT	–
$V_n'T$	–	–	–	$V_n'T$

$\circ$	T	$M_{np}T$	V_nT	$V_n'T$
T	T	$M_{np}T$	V_nT	$V_n'T$
$M_{mn}T$	–	$M_{mp}T$	V_mT	–
V_nT	–	–	–	$M_{nn}T$
$V_n'T$	–	$V_p'T$	T	–

$$\circ = \begin{cases} +, - & T = \mathbb{Z} \\ \boxplus, \boxminus & T = S \text{ or } \mathbb{C}S \\ \diamondplus, \diamondminus & T = IS \text{ or } \mathbb{C}IS \end{cases} \qquad \circ = \begin{cases} \cdot & T = \mathbb{Z} \\ \boxdot & T = S \text{ or } \mathbb{C}S \\ \diamonddot & T = IS \text{ or } \mathbb{C}IS \end{cases}$$

Table 6.5. Type of results of matrix and vector operations.

Transposed vectors are used to perform scalar or dyadic products. The left table also holds for the intersection and interval hull of matrices and vectors with components of IS or $\mathbb{C}IS$.

The order relation $\leq$ for matrices of $M_{mn}T$ with $T \in \{\mathbb{Z}, S, \mathbb{C}S, IS, \mathbb{C}IS\}$ is defined by

$$\bigwedge_{(a_{ij}),(b_{ij})\in M_{mn}T} (a_{ij}) \leq (b_{ij}) :\Leftrightarrow \bigwedge_{i=1(1)m} \bigwedge_{j=1(1)n} a_{ij} \leq b_{ij}$$

$$\text{and} \quad \bigwedge_{a,b\in M_{mn}T} a < b :\Leftrightarrow a \leq b \wedge a \neq b.$$

In $M_{mn}IS$, inclusion $\subseteq$ is defined by

$$\bigwedge_{(a_{ij}),(b_{ij})\in M_{mn}IS} (a_{ij}) \subseteq (b_{ij}) :\Leftrightarrow \bigwedge_{i=1(1)m} \bigwedge_{j=1(1)n} a_{ij} \subseteq b_{ij}$$

$$\text{and} \bigwedge_{a,b\in M_{mn}IS} a \subset b :\Leftrightarrow a \subseteq b \wedge a \neq b.$$

In $M_{mn}\mathbb{C}IS$, inclusion is defined by

$$\bigwedge_{((a_{ij},b_{ij})),((c_{ij},d_{ij}))\in M_{mn}\mathbb{C}IS} ((a_{ij},b_{ij})) \subseteq ((c_{ij},d_{ij}))$$
$$:\Leftrightarrow \bigwedge_{i=1(1)m} \bigwedge_{j=1(1)n} a_{ij} \subseteq c_{ij} \wedge b_{ij} \subseteq d_{ij}$$

$$\text{and} \bigwedge_{a,b\in M_{mn}\mathbb{C}IS} a \subset b :\Leftrightarrow a \subseteq b \wedge a \neq b.$$

We have now completed specification of matrix and vector operations when the components of both operands are of the same type for all the basic data sets (types) $\mathbb{Z}$, S, $\mathbb{C}S$, IS, $\mathbb{C}IS$. We now consider the operations for which the components of both operands are of different types.

As before, the general rule for performing the matrix and vector operations for operands of different component types is to lift the operand with the simpler component type into the type of the other operand by means of a type transfer. Then the operation can be executed as one of the same component type, all of which we have already dealt with. If, for instance, a matrix $a \in M_{nn}S$ has to be multiplied by a vector $b \in V_n\mathbb{C}S$, we transfer a into an element of $M_{nn}\mathbb{C}S$ by adding zero imaginary parts to all of the components of a. Then we multiply this matrix of $M_{nn}\mathbb{C}S$ by the vector $b \in V_n\mathbb{C}S$. Or if, for instance, a matrix $a \in M_{mn}\mathbb{Z}$ has to be multiplied by a vector $b \in V_nIS$, we first transfer a into a floating-point matrix of $M_{mn}S$ and then into an interval matrix of $M_{mn}IS$ whose components are all point intervals. Then we multiply the matrix $a \in M_{mn}IS$ by the vector $b \in V_nIS$.

Now let T_1, T_2, and T_3 each denote one of the basic data sets (types) $\mathbb{Z}$, S, $\mathbb{C}S$, IS, $\mathbb{C}IS$. We consider the set of matrices $M_{mn}T_i$, vectors V_nT_i, and transposed vectors $V_n'T_i$, $i = 1(1)3$, whose components are chosen from the basic data types. Among other operations, elements of these sets can be multiplied. Table 6.6 (bottom) displays the types of such products. In this table, T_3 is to be replaced by the result type of Table 6.3 if the components of the operands are of the type T_1 and T_2 respectively.

A corresponding table for matrix and vector addition, subtraction, intersection, and interval hull is also given in Table 6.6.

$+, -, \cap, \bar{\cup}$	T_2	$M_{mn}T_2$	V_nT_2	$V_n'T_2$
T_1	T_3	$-$	$-$	$-$
$M_{mn}T_1$	$-$	$M_{mn}T_3$	$-$	$-$
V_nT_1	$-$	$-$	V_nT_3	$-$
$V_n'T_1$	$-$	$-$	$-$	$V_n'T_3$

$\cdot$	T_2	$M_{np}T_2$	V_nT_2	$V_n'T_2$
T_1	T_3	$M_{np}T_3$	V_nT_3	$V_n'T_3$
$M_{mn}T_1$	$-$	$M_{mp}T_3$	V_mT_3	$-$
V_nT_1	$-$	$-$	$-$	$M_{nn}T_3$
$V_n'T_1$	$-$	$V_p'T_3$	T_3	$-$

Table 6.6. Type of the result of matrix and vector operations.

To conclude we emphasize that the formulas and the discussion in this section show that all of the level 2 operations and relations can be performed on a computer if the level 1 operations are available. The level 1 operations include an exact scalar product.

Chapter 7

Hardware Support for Interval Arithmetic

A hardware unit for interval arithmetic (including division by an interval that contains zero) is described in this chapter. After a brief introduction an instruction set for interval arithmetic is defined which is attractive from the mathematical point of view. The set consists of the basic arithmetic operations and comparisons for intervals including the relevant lattice operations. To enable high speed, the case selections for interval multiplication (9 cases) and interval division (14 cases) are done in hardware. The lower bound of the result is computed with rounding downwards and the upper bound with rounding upwards by parallel units simultaneously. The rounding mode must be inherent to all the arithmetic operations. Also the basic comparisons for intervals together with the corresponding lattice operations and the result selection in more complicated cases of multiplication and division are done in hardware by parallel units. The circuits described in this chapter show that with modest additional hardware costs interval arithmetic can be made almost as fast as simple floating-point arithmetic. Fast hardware support of interval arithmetic is an essential engredient of what is called *advanced computer arithmetic* in this book.

7.1 Introduction

Interval mathematics has been developed to a high standard over the last few decades. It provides methods which deliver results with guarantees. However, the arithmetic on existing processors makes these methods slow. This chapter deals with the question of how interval arithmetic can be provided effectively on computers. This is an essential prerequisite for the superior and fascinating properties of interval mathematics to be more widely used in the scientific computing community. With more suitable processors, rigorous methods based on interval arithmetic could be comparable in speed to today's approximate methods. Interval arithmetic is a natural extension to floating-point arithmetic, not a replacement for it. As computers speed up, from gigaflops to teraflops to petaflops, interval arithmetic becomes a principal and necessary tool for controlling the precision of a computation as well as the accuracy of the computed and guaranteed result.

In conventional numerical analysis *Newton's method* is the key algorithm for nonlinear problems. The method converges quadratically to the solution if the initial value of the iteration is already close enough to it. However, it may fail in finite or infinite precision arithmetic even if there is only one solution in the given interval. In contrast to this, the interval version of Newton's method is globally convergent. It never fails, not even in rounded arithmetic. Newton's method reaches its ultimate elegance and power in the *extended interval Newton method*. It yields enclosures of all single zeros in a given domain. It is quadratically convergent. The key operation to achieve these fascinating properties is division by an interval which contains zero. It separates different zeros from each other. A method which provides for computation of all the zeros of a system of equations in a given domain is very frequently used. This justifies taking division by an interval which contains zero into the basic set of interval operations, and implementing it by the hardware of the computer.

In this chapter the consideration is restricted to operands or $I\mathbb{R}$. These are closed and bounded intervals of real numbers. The result, however, may be an element of $(I\mathbb{R})$.

To handle critical situations it is generally agreed that interval arithmetic must be supplemented by some measure to increase the precision within a computation. An easy way to achieve this is an exact *multiply and accumulate* instruction or, what is equivalent to it, an exact scalar product. With it quadruple or multiple precision arithmetic can easily be provided. If the multiply and accumulate instruction is implemented in hardware it is very fast. The data can be stored and moved as double precision floating-point numbers. Generally speaking, interval arithmetic brings guarantees and mathematics into computation, while the exact multiply and accumulate instruction brings higher (dynamic) precision and accuracy. Hardware support for the exact multiply and accumulate instruction is discussed in Chapter 8 of the book. Fast quadruple or multiple precision arithmetic and multiple precision interval arithmetic are discussed in Chapter 9.

7.2 An Instruction Set for Interval Arithmetic

From the mathematical point of view, the following instructions for interval operations and comparisons are desirable. In the following $A = [a_1, a_2]$ and $B = [b_1, b_2]$ denote interval operands of $I\mathbb{R}$. $C = [c_1, c_2]$ denotes the result of an interval operation. If in C a bound is $-\infty$ or $+\infty$ the bound is not included in the interval. In this case a round bracket is used for the representation of the interval at this interval bound. For the operations $\circ \in \{+, -, \cdot, /\}$ with rounding downwards or upwards, the symbols $\triangledown\mkern-14mu\circ\,, \triangle\mkern-14mu\circ\,, \circ \in \{+, -, \cdot, /\}$ are used respectively.

7.2.1 Algebraic Operations

Addition. $C := [a_1 \triangledown\mkern-14mu+\, b_1, a_2 \triangle\mkern-14mu+\, b_2]$.

Subtraction. $C := [a_1 \triangledown b_2, a_2 \triangle b_1]$.

Multiplication.

	$b_1 \geq 0$	$b_1 < 0 \leq b_2$	$b_2 < 0$
$a_1 \geq 0$	$[a_1 \triangledown b_1, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, a_1 \triangle b_2]$
$a_1 < 0 \leq a_2$	$[a_1 \triangledown b_2, a_2 \triangle b_2]$	$[\min(a_1 \triangledown b_2, a_2 \triangledown b_1),$ $\max(a_1 \triangle b_1, a_2 \triangle b_2)]$	$[a_2 \triangledown b_1, a_1 \triangle b_1]$
$a_2 < 0$	$[a_1 \triangledown b_2, a_2 \triangle b_1]$	$[a_1 \triangledown b_2, a_1 \triangle b_1]$	$[a_2 \triangledown b_2, a_1 \triangle b_1]$

Division.

$0 \notin B$	$b_1 > 0$	$b_2 < 0$
$a_1 \geq 0$	$[a_1 \triangledown b_2, a_2 \triangle b_1]$	$[a_2 \triangledown b_2, a_1 \triangle b_1]$
$a_1 < 0 \leq a_2$	$[a_1 \triangledown b_1, a_2 \triangle b_1]$	$[a_2 \triangledown b_2, a_1 \triangle b_2]$
$a_2 < 0$	$[a_1 \triangledown b_1, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, a_1 \triangle b_2]$

$0 \in B$	$b_1 = b_2 = 0$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$
$a_2 < 0$	$\varnothing$	$[a_2 \triangledown b_1, +\infty)$	$[a_2 \triangledown b_1, a_2 \triangle b_2]$[1]	$(-\infty, a_2 \triangle b_2]$
$a_1 \leq 0 \leq a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$a_1 > 0$	$\varnothing$	$(-\infty, a_1 \triangle b_1]$	$[a_1 \triangledown b_2, a_1 \triangle b_1]$[1]	$[a_1 \triangledown b_2, +\infty)$

In the table for division by an interval which contains zero, $\varnothing$ denotes the empty interval. Since the result of an interval operation is supposed to always be a single interval, the results which consist of the union of two intervals are delivered and represented as improper intervals $[a_2 \triangledown b_1, a_2 \triangle b_2]$ and $[a_1 \triangledown b_2, a_1 \triangle b_1]$. In these special cases the left hand bound is higher than the right hand bound.

The empty set $\varnothing$ needs a particular encoding. (`+NaN`, `-NaN`) may be an appropriate encoding for the empty interval.

7.2.2 Comments on the Algebraic Operations

Except for a few cases in the table for division by an interval which contains zero, the lower bound of the result is always obtained with an operation rounded downwards and the upper bound with an operation rounded upwards. Multiplication and division need all combinations of lower and upper bounds of input intervals depending on the signs of the bounds. Thus an operand selection has to be performed before the

[1] special encoding of result $(-\infty, c_2] \cup [c_1, +\infty)$

operation can be executed. However, in all cases of computing the bounds of the result interval the left hand operand is a bound of the interval $A = [a_1, a_2]$ and the right hand operand is always a bound of the operand $B = [b_1, b_2]$. Thus the operand selection can be executed for the bounds of A and B separately.

In one special case, $0 \in A$ and $0 \in B$, multiplication requires the computation of two result pairs. Then the interval hull of these is taken.

Otherwise, if $0 \in B$ then division may produce various special results like $+\infty$, $-\infty$ or the empty interval.

7.2.3 Comparisons and Lattice Operations

c is a value of type boolean.
equality $c := (a_1 = b_1 \wedge a_2 = b_2)$,
less than or equal $c := (a_1 \leq b_1 \wedge a_2 \leq b_2)$,
greatest lower bound $C := [\min(a_1, b_1), \min(a_2, b_2)]$,
least upper bound $C := [\max(a_1, b_1), \max(a_2, b_2)]$,
inclusion $c := (b_1 \leq a_1 \wedge a_2 \leq b_2)$,
element of $c := (b_1 \leq a \wedge a \leq b_2)$, special case of inclusion,
interval hull $C := [\min(a_1, b_1), \max(a_2, b_2)]$,
intersection $C := [\max(a_1, b_1), \min(a_2, b_2)]$ or the empty interval $\varnothing$,
check interval branch on $a_1 > a_2$ (checking for proper interval).

7.2.4 Comments on Comparisons and Lattice Operations

For comparisons and lattice operations no shuffling of bounds is needed. All combinations of minimum and maximum of lower and upper bounds do occur.

In IEEE arithmetic the bit string of nonnegative floating-point numbers can be treated as an integer for comparison purposes. Computation of minimum or maximum is done by comparison and selection.

Intersection and checking for an improper interval needs a comparison of the lower and the upper bound of the result interval. In all other cases the lower and the upper bound of the result interval can be computed independently.

7.3 General Circuitry for Interval Operations and Comparisons

7.3.1 Algebraic Operations

We assume that the data are read from and written into a register file or through a memory-access-unit. Each memory cell holds 128 bits for pairs of double precision floating-point numbers holding the lower and upper bound of an interval. Intervals are moved within the unit via busses of 128 bits.

There are three such busses in the interval arithmetic unit, one for the operand $A = [a_1, a_2]$, a second one for the operand $B = [b_1, b_2]$, and a third one for the result $C = [c_1, c_2]$. The interval arithmetic unit has two parts, one to perform the operand selection and the operations, the other to perform comparisons and result selection. See Figure 7.1.

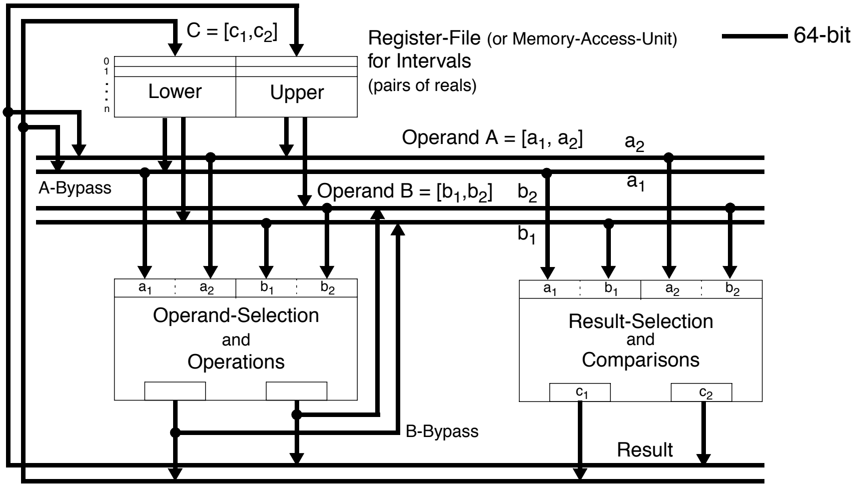


Figure 7.1. General circuitry for interval operations and comparisions.

An operation is performed as follows: The operands $A = [a_1, a_2]$ and $B = [b_1, b_2]$ are read from the memory onto the A-bus and B-bus and forwarded to the Operand-Selection and Operations Unit. Here from the lower and upper bounds of the operands, multiplexers select various combinations of values depending on the operation, and on signs and zeros of all four values. Finally a pair of operations $\triangledown$ and $\triangle$, $\circ \in \{+, -, \cdot, /\}$, is performed on the selected values, one with rounding downwards and one with rounding upwards. See Figure 7.2. In many cases the computed values are already the desired result $C = [c_1, c_2]$. In these cases the result is forwarded to the memory via the C-bus. But there are exceptions, for instance multiplication where both operands contain zero, or division by an interval that contains zero. See the tables in Section 7.2. In these cases the computed values are forwarded to the Comparison and Result-Selection Unit for further processing.

The selector signals o_{a1}, o_{a2}, o_{b1}, and o_{b2} control the multiplexers. The Operand-Selection and Operations Unit performs all arithmetic operations.

In **addition** the lower bounds of A and B are simply added with rounding downwards and the upper bounds with rounding upwards $[a_1 \triangledown b_1, a_2 \triangle b_2]$. The selector signals are set to $o_{a1} = 0$, $o_{a2} = 1$, $o_{b1} = 0$, and $o_{b2} = 1$.

In **subtraction** the bounds of B are exchanged by the operand selection. Then the subtraction is performed, the lower bound being computed with rounding downwards and the upper bound with rounding upwards $[a_1 \triangledown b_2, a_2 \triangle b_1]$. The selector signals are set to $o_{a1} = 0$, $o_{a2} = 1$, $o_{b1} = 1$, and $o_{b2} = 0$.

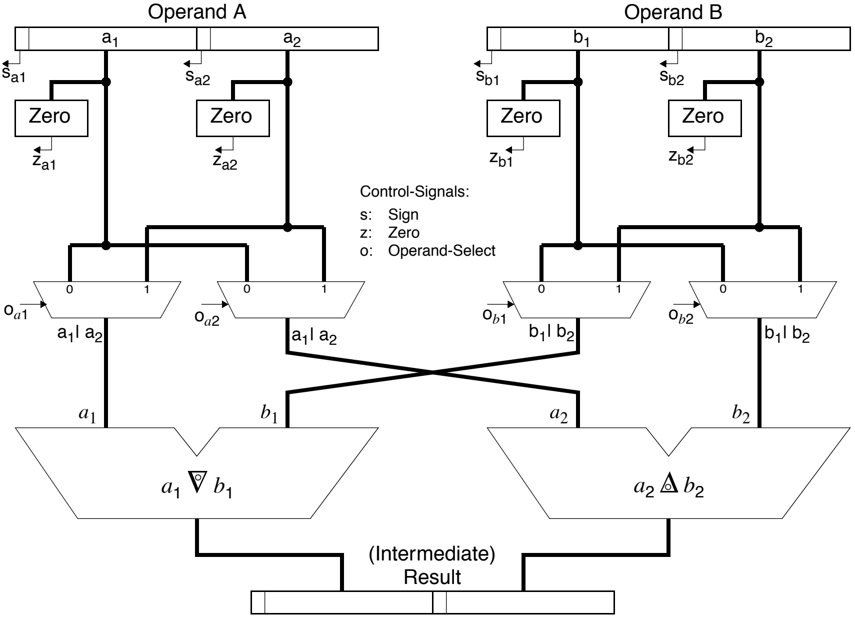


Figure 7.2. Operand-Selection and Operations Unit.

Multiplication is a little more complicated. The various multiplications are performed in the following way:

If neither operand A nor B contains zero $(s_{a1} \cdot \overline{s_{a2}} \cdot s_{b1} \cdot \overline{s_{b2}} = 0)$

then the bounds are shuffled, the result $[a_1 \triangledown b_1, a_2 \triangle b_2]$ is computed with the selected bounds, and delivered to the target via the C-bus.

else (i) The bounds are shuffled for the first multiplication by operand selection, the first partial result $[a_1 \triangledown b_2, a_1 \triangle b_1]$ is computed, and it is forwarded to the Comparison and Result-Selection Unit via the A-bus.

(ii) The bounds are shuffled for the second variation by operand selection, the second partial result $[a_2 \triangledown b_1, a_2 \triangle b_2]$ is computed, and it is forwarded to the Comparison and Result-Selection Unit via the B-bus.

(iii) In the Comparison and Result-Selection Unit the hull of the two multiplications is selected and delivered as the final result to the target via the C-bus.

For multiplication, the multiplexers are controlled by the following selector signals:[1]

$$o_{a1} = s_{b2} + \overline{s_{a1}} \cdot s_{b1} + ms, \qquad o_{a2} = \overline{s_{b1}} + \overline{s_{a1}} \cdot \overline{s_{b2}} + ms,$$

$$o_{b1} = \overline{ms}(s_{a2} + s_{a1} \cdot \overline{s_{b2}}), \qquad o_{b2} = \overline{s_{a1}} + \overline{s_{a2}} \cdot \overline{s_{b1}} + ms.$$

These signals are computed by the signs of the bounds of the interval operands $A = [a_1, a_2]$ and $B = [b_1, b_2]$. In the expressions the signal ms is zero in case 1, and it is one in case 2, of the conditional statement above.

For multiplication, every operand selector signal can be realized by two or three gates!

Division is a little simpler than multiplication, but it is organized in a similar pattern:

If B does not contain zero ($\overline{s_{b1}} \cdot \overline{z_{b1}} + s_{b2} = 1$)

then the bounds are shuffled, the result $[a_1 \triangledown b_1, a_2 \triangle b_2]$ is computed with the selected bounds, and delivered to the target via the C-bus

else (i) the bounds are shuffled, the arithmetic operation $a_1 \triangledown b_1$ and/or $a_2 \triangle b_2$ is computed with the selected bounds, and it is forwarded to the Comparison and Result-Selection Unit together with the selection code for special values

(ii) in the Comparison and Result-Selection Unit the result is generated from arithmetic values and/or special values $(-\infty|\varnothing_1| + \infty|\varnothing_2)$, and it is forwarded to the target via the C-bus. $\varnothing_1$ and $\varnothing_2$ are assumed here to represent particular encodings of the empty set $\varnothing = (\varnothing_1, \varnothing_2)$. (+NaN, -NaN) may be such an encoding.

The operand selection is controlled by the following selector signals:

$$o_{a1} = s_{b2} + s_{a1} \cdot s_{b1}, \qquad o_{a2} = \overline{s_{b1}} + s_{a2} \cdot \overline{s_{b2}},$$

$$o_{b1} = \overline{s_{a1}} + \overline{s_{a2}} \cdot s_{b1}, \qquad o_{b2} = s_{a2} + s_{a1} \cdot s_{b1}.$$

For division, every operand selector signal can be realized by two gates!

For division by an interval which contains zero, the result is an extended interval where at most one bound is a real number or it is an improper interval where the left hand bound is higher than the right hand bound. The instruction *check interval* supplies a test for improper intervals.[2] In Newton's method, for instance, the following operation is then an intersection with a regular interval. It delivers one or two regular intervals.

[1] Note: A negative sign is a 1. A bar upon a logical value means inversion. In the expressions a dot stands for a logical and, and a plus for a logical or.

[2] Within the given framework of existing processors only one interval can be delivered as the result of an operation. Other solutions which use special registers or flags or new exceptions would require an adaption of the operating system.

7.3.2 Comparisons and Result-Selection

In this unit, comparisons and minima and maxima are to be computed. We mention again that in IEEE arithmetic the bit string of nonnegative floating-point numbers can be interpreted as an integer for comparison purposes. Computation of a minimum or maximum comprises comparison and then selection. Thus the arithmetic that has to be done in this unit is relatively simple. Again the computation of the lower bound and the upper bound of the result is done in parallel and simultaneously. No bound shuffling is necessary in this unit. See Figure 7.3.

The comparisons for **equality**, **less than or equal** and **set inclusion** are done by comparing the bounds, combining the results and setting a flag in the state register. For the operation **element of** an interval, $[a, a]$ is built, then a test for inclusion is applied.

The minimum and maximum computations for the operations **higher lower bound**, **lower upper bound**, **interval hull**, and result selection for multiplication where $0 \in A$ and $0 \in B$ are executed by comparing the bounds and selecting the higher and the lower, respectively. Then the result is delivered to the target.

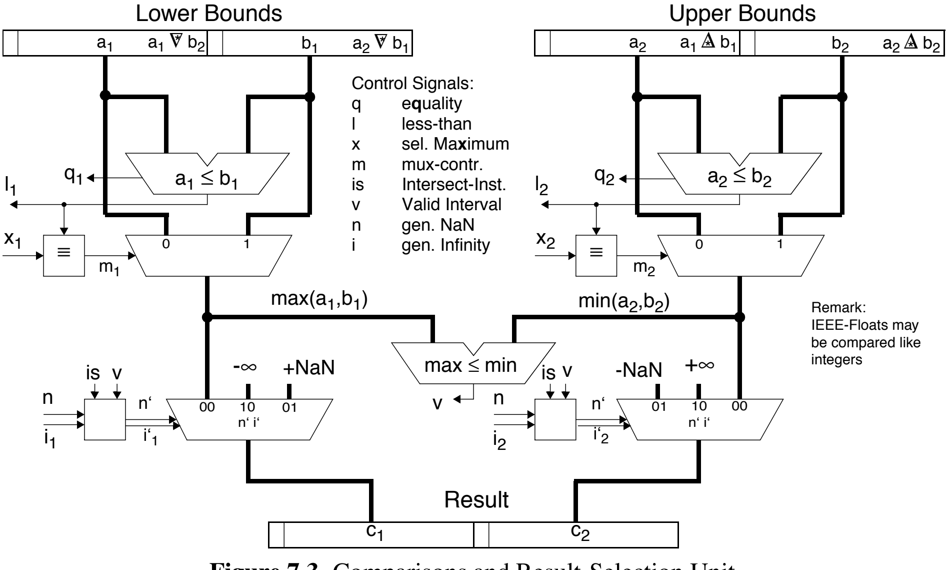


Figure 7.3. Comparisons and Result-Selection Unit.

The computation of the **intersection** is slightly more complicated. First the bounds of the operands are compared and the higher lower bound and the lower upper bound are selected. If $\max(a_1, b_1) \leq \min(a_2, b_2)$, the intersection, which is the interval $[\max(a_1, b_1), \min(a_2, b_2)]$ is delivered to the target, otherwise the empty interval is delivered. In Figure 7.3 it is assumed that the empty set is encoded as $\varnothing = (+\mathtt{NaN}, -\mathtt{NaN})$.

For division by an interval which contains zero, an interval with bounds like +NaN, $-\infty$, -NaN, $+\infty$ has to be delivered. These alternatives are selected in the Comparison and Result-Selection Unit.

Now we define the various selector signals that appear in Figure 7.3.

$$
\begin{aligned}
n = n_1 = n_2 &= \left(\overline{z_{a1}} \cdot \overline{s_{a1}} \cdot \overline{s_{a2}} + s_{a2}\right) \cdot z_{b2} \cdot s_{b1}, && \text{3 Gates,} \\
i_1 &= z_{a1} \cdot z_{b1} + z_{a1} \cdot s_{b1} \cdot \overline{s_{b2}} + s_{a1} \cdot \overline{s_{a2}} \cdot s_{b1} \cdot \overline{s_{b2}} \\
&\quad + \overline{s_{a2}} \cdot \overline{z_{b1}} \cdot z_{b2} + s_{a1} \cdot z_{b1} \cdot \overline{z_{b2}} + s_{a1} \cdot \overline{s_{a2}} \cdot z_{b1}, && \text{7 Gates,} \\
i_2 &= z_{a1} \cdot z_{b1} + z_{a1} \cdot s_{b1} \cdot \overline{s_{b2}} + s_{a1} \cdot \overline{s_{a2}} \cdot s_{b1} \cdot \overline{s_{b2}} \\
&\quad + s_{a1} \cdot \overline{s_{a2}} \cdot z_{b1} + \overline{s_{a2}} \cdot z_{b1} \cdot \overline{z_{b2}} + s_{a2} \cdot \overline{z_{b1}} \cdot z_{a2}, && \text{4 Gates,} \\
i'_1 &= \overline{is} \cdot i_1, \\
i'_2 &= \overline{is} \cdot i_2, \\
n' = n_1 = n_2 &= \overline{is} \cdot n + n \cdot \overline{v}.
\end{aligned}
$$

The expressions for i_1 and i_2 contain common subexpressions.

The logical expressions given in this section were developed from function tables of up to several hundred entries. Then minimization was performed which leads to the equations given. We do not repeat the tables and the minimization here because all this is a standard procedure in circuit design. The function tables contain very many "don't care" entries which allows realization with a very small number of gates. Since the "don't care" entries may be used during minimization in different ways, various designs may end up with different equations of equal complexity.

7.3.3 Alternative Circuitry for Interval Operations and Comparisons

In Figure 7.2 the operand selection and the execution of the interval operations together form the Operand-Selection and Operations Unit. This is justified since the time needed for the operand selection is negligible in comparison to the time needed to perform the arithmetic operations.

It may, however, be useful to separate the operand selection from the operations themselves. Separation into two units would make it easier to use these for other purposes. Examples are ordinary arithmetic or shuffling of parts of the operands. Many processors are being built which already provide several independent arithmetic units. These may then be easily used as part of the interval operations unit. Figure 7.4 shows a circuit for such interval operations and comparisons.

The Operand-Selection Unit and the Operations Unit would then look as shown in Figure 7.5 and Figure 7.6. We will not discuss these circuits in further detail. Their functionality should be clear from what has been said already.

In summary it can be said that a hardware unit for fast interval arithmetic has a very regular structure. Interval arithmetic is just regular arithmetic for pairs of reals with particular roundings, plus operand selection, plus clever control.

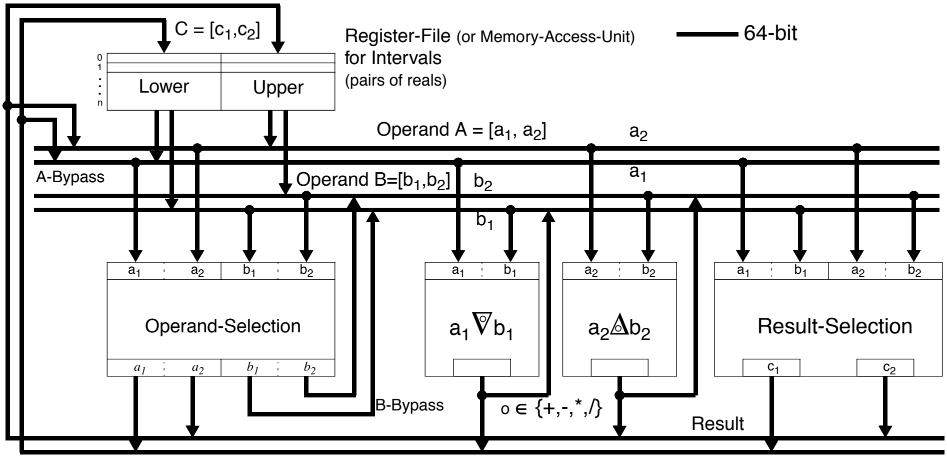


Figure 7.4. General circuitry for interval operations and comparisons.

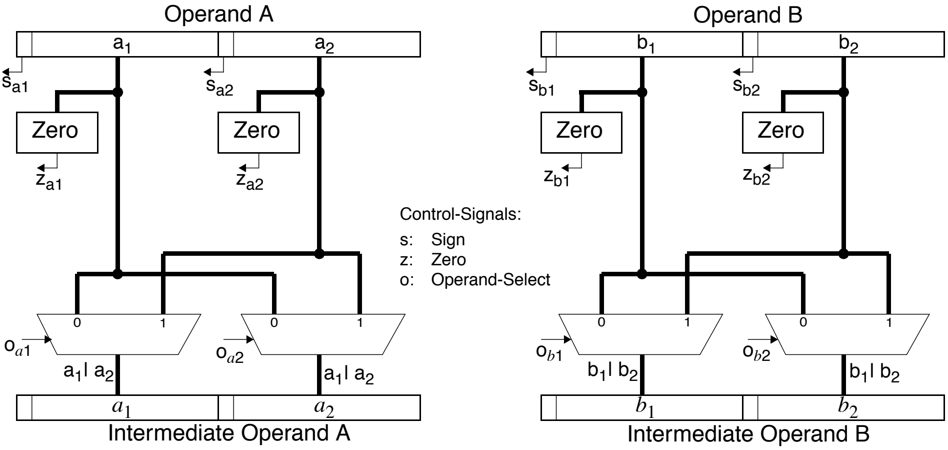


Figure 7.5. Operand Selection Unit.

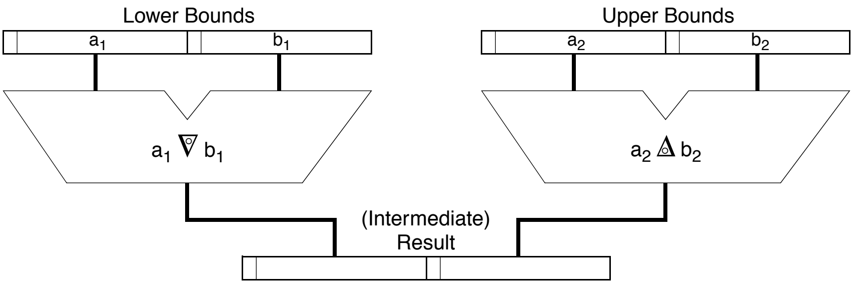


Figure 7.6. Arithmetic Operations Unit.

7.3.4 Hardware Support for Interval Arithmetic on X86-Processors

It is interesting to note that most of what is needed for fast hardware support for interval arithmetic is already available on current x86-processors. Figure 7.7 shows figures from various publications by Intel.

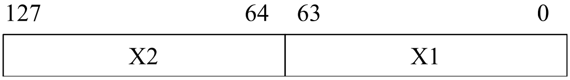


Figure 6. Packed double precision floating-point data type

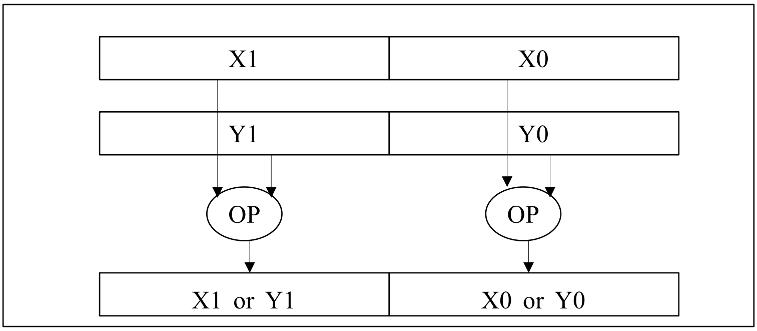


Figure 11-3. Packed Double-Precision Floating-Point Operation

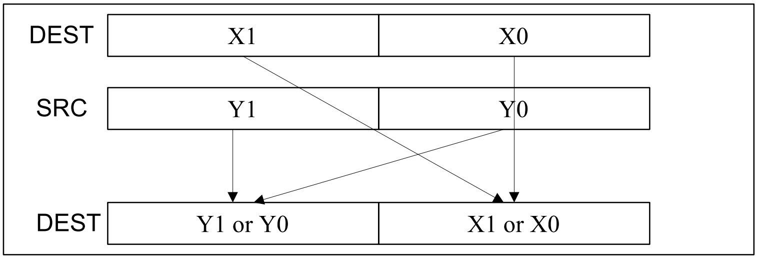


Figure 11-5. SHUFFD Instruction Packed Shuffle Operation

Figure 7.7. Figures from various Intel publications.

On an Intel Pentium 4, for instance, eight registers are available for words of 128 bits (xmm0, xmm1, ..., xmm7). The x86-64 processors even provide 16 such registers. These registers can hold pairs of double precision floating-point numbers. They can be viewed as bounds of intervals. Parallel operations like $+$, $-$, $\cdot$, $/$, min, max, and compare can be performed on these pairs of numbers. What is not available and would be needed is for one of the two operations to be rounded downwards and the other one rounded upwards. The last picture in Figure 7.7 shows that even shuffling of bounds is possible under certain conditions. This is half of operand selection needed for interval arithmetic. So an interval operation would need two such units or to pass this unit twice. Also nearly all of the data paths are available on current x86-processors. Thus full hardware support of interval arithmetic would probably add less than 1%, more likely less than 0.1% to a current Intel or AMD x86 processor chip.

Full hardware support of fast interval arithmetic on RISC processors may cost a little more as these lack pairwise processing. But most of them have two arithmetic units and use them for super scalar processing. What has to be added is some sophisticated control.

7.3.5 Accurate Evaluation of Interval Scalar Products

Chapter 8 deals with exact computation of scalar products of two vectors with floating-point components. This requires computation of the products of corresponding vector components to the full double length and exact accumulation of these products. In Chapter 9 use will often be made of exact scalar products of two vectors with interval components. These can be computed by hardware which is very similar to that developed in this chapter.

If $A = (A_\nu)$ and $B = (B_\nu)$ with $A_\nu = [a_{\nu 1}, a_{\nu 2}]$ and $B_\nu = [b_{\nu 1}, b_{\nu 2}] \in IS$ are two such vectors this requires evaluation of the formulas

$$\boldsymbol{A} \diamondsuit \boldsymbol{B} = \diamondsuit \left(\boxed{\sum_{\nu=1}^{n}} A_\nu \boxdot B_\nu \right) = \left(\diamondsuit \boxed{\sum_{\nu=1}^{n}} A_\nu \boxdot B_\nu \right) \tag{7.3.1}$$

$$= \left[\triangledown \sum_{\nu=1}^{n} \min_{i,j=1,2} (a_{\nu i} b_{\nu j}), \triangle \sum_{\nu=1}^{n} \max_{i,j=1,2} (a_{\nu i} b_{\nu j}) \right]. \tag{7.3.2}$$

Here the products of the bounds of the vector components $a_{\nu i} b_{\nu j}$ are elements of $\mathbb{R}$ but in general not of S. They are to be computed to the full double length. Then the minima and maxima have to be selected. These selections can be done by distinguishing the nine cases shown in Table 4.1. The selected products are then accumulated in $\mathbb{R}$ as an exact scalar product. Finally the sum of products is rounded only once by $\triangledown$ (resp. $\triangle$) from $\mathbb{R}$ into S. The two sums can be computed into two long fixed-point registers by one of the techniques shown in Chapter 8.

The case distinctions (the minimum and maximum selection in (7.3.2)) can be hardware supported as shown in Figure 7.2 or 7.5. However, the succeeding multiplications of these figures have here to be done without rounding. Clearly, forwarding the products to the result selection unit or to the target in Figures 7.1 and 7.4 then requires wider busses. Design of a scalar product unit for vectors with interval components is an interesting task. See Section 8.6.2.

Chapter 8

Scalar Products and Complete Arithmetic

This chapter deals with the implementation of the scalar or dot product of two floating-point vectors, an operation which is occasionally called *multiply and accumulate*. *Basic computer arithmetic* and the mapping principle of semimorphism require that the computed result of the scalar product differs from the exact result by at most one rounding. The circuits developed in this chapter, however, go beyond the concept of semimorphism. *Scalar products are computed exactly.* The gain is simplicity, speed and accuracy. The circuits described here for the computation of such scalar products can be used in all kinds of computers: personal computers, workstations, mainframes, supercomputers, and digital signal processors.

The most natural way to compute a scalar product exactly is to add the products of the vector components into a long fixed-point register on the arithmetic unit. This method has the advantage of being quite simple and very fast. A special resolution technique is used to absorb the carry as soon as it arises. Since fixed-point accumulation is error free it always provides the desired exact answer. So we have the seeming paradox and striking outcome that scalar products of vectors with millions of components can be computed exactly using a relatively small finite register in the arithmetic unit. Fixed-point accumulation is simpler and faster than accumulation in floating-point arithmetic. The circuits developed in this chapter show that there is no way to compute an approximation of a scalar product faster than the exact result. In a very natural pipeline the arithmetic, consisting of multiplication and accumulation, can be performed in the time which is needed to read the data into the arithmetic unit. Pipelining is a basic first step towards parallel computing and high speed. In numerical analysis, the scalar product is ubiquitous.

To make the new capability conveniently available to the user a new data format called *complete* is used together with a few simple arithmetic operations associated with each floating-point format. *Complete arithmetic* computes all scalar products of floating-point vectors exactly. The result of complete arithmetic is always exact; it is complete, not truncated. Not a single bit is lost. A variable of type complete is a fixed-point word wide enough to allow exact accumulation (continued summation) of floating-point numbers and of simple products of such numbers. If register space for the complete format is available complete arithmetic is very very fast. The arithmetic needed to perform complete arithmetic is not much different from that available in a conventional CPU. In the case of the IEEE double precision format a *complete register* consists of about $1/2$ K bytes. Instructions for complete arithmetic for low and high level programming languages are discussed.

8.1 Introduction and Motivation

The number one requirement for computer arithmetic has always been speed. It is the main need that drives the technology. With increased speed larger problems can be attempted. Pipelining of arithmetic operations and vector processing are important techniques to speed up a computation. Particularly suited to pipelining are so-called elementary compound operations like accumulate or multiply and accumulate. The first operation is a particular case of the second one which computes a sum of products, the dot product or scalar product of two vectors. Advanced processors and programming languages offer these operations. Pipelining makes them really fast, especially if there are many summands.

In conventional vector processors the accumulation is done in floating-point arithmetic by the so-called *partial sum technique*. The pipeline for the accumulation consists of p steps. What comes out of the pipeline is wrapped back into the input of the pipeline as a second summand. Thus p sums are built up in floating-point arithmetic. Finally these sums are added together to obtain the scalar product. This partial sum technique changes the sequence of the summands. This is an additional source of error. The conventional error analysis for a floating-point algorithm does not necessarily hold for this kind of evaluation. Difficulties concerning the accuracy of conventional vector processors have been addressed in the literature [202, 489, 656, 657]. This is an unsatisfactory situation. The user should not be obliged to perform an error analysis every time a compound arithmetic operation, implemented by the manufacturer or in the programming language, is employed. If such operations are automatically inserted into a user's program by a vectorizing compiler the user loses complete control of his computation.

Computer users, vendors, and even mathematicians often argue that a more accurate arithmetic is not needed since the underlying model is imprecise, or discretization errors are much larger than rounding errors.

This kind of justification is rather dubious. One source of error does not justify adding another one. Even an imprecise mathematical model or imprecise data will suffer from the use of an imprecise or sloppy arithmetic. The systematic development of a mathematical model requires that the error resulting from the computation can largely be excluded. This requires the best possible arithmetic. What happens outside the computer is the responsibility of the user. As soon as the data are in the computer treating them as exact is a must. The vendor is responsible for the arithmetic that is used in the computer. To be as accurate as possible is also a must. Internal and external error sources must be separately identifiable.

There is another side of the computational coin apart from speed: the accuracy and reliability of the computed result. Progress on this side is also very important, if not essential. *Basic computer arithmetic* requires that all computer approximations of arithmetic operations – in particular those in the usual vector and matrix spaces – differ from the exact result by at most one rounding.

The concept of semimorphism requires that scalar products be computed with but a single rounding. Extensive studies of the topic and a look into computing history (see the next section), however, show that accumulations of products can easily be performed exactly on the computer with no rounding whatsoever. These concepts are discussed in this chapter. This goes beyond the concept of semimorphism. The methods lead to a considerable gain in computing speed. Exception handling is greatly simplified.

In this chapter, scalar product units are developed for different kinds of computer: personal computers, workstations, mainframes, supercomputers and digital signal processors. All these units compute the scalar product exactly. Not a single bit is lost. The products are added into a fixed-point register in the arithmetic unit. Fixed-point addition is error free. It always provides the desired exact answer. Pipelining makes these operations really fast. The differences in the circuits for the different processors are dictated by the speed with which the processor delivers the data into the arithmetic or scalar product unit. The data are the vector components. In all cases the extended computational capability is gained at modest cost. The cost increase is comparable to that from a simple to a fast multiplier, for instance by a Wallace tree, accepted years ago. It is a central result of the development that for all processors mentioned above, circuits can be given for the computation of the exact scalar product with virtually no unoverlapped computing time needed for the execution of the arithmetic. In a pipeline, the arithmetic can be executed in the time the processor needs to read the data into the arithmetic unit. This means that no other method of computing a scalar product can be faster, in particular not a conventional approximate computation of the scalar product in double or quadruple precision floating-point arithmetic. Fixed-point accumulation of the products is simpler than accumulation in floating-point. Many intermediate steps that are executed in a floating-point accumulation, such as normalization and rounding of the products and the intermediate sum, composition into a floating-point number and decomposition into mantissa and exponent for the next operation, do not occur in the fixed-point accumulation of the exact scalar product. Since the result is always exact, no exception handling is needed.

8.2 Historic Remarks

Floating-point arithmetic has been used since the early forties and fifties (Zuse Z3, 1941) [63, 500, 501, 533]. Technology in those days was poor (electromechanical relays, electron tubes). It was complex and expensive. The word size of the Z3 was 24 bits. The storage provided 64 words. The four elementary floating-point operations were all that could be provided. For more complicated calculations error analysis was left to the user.

Before that time, highly sophisticated mechanical computing devices were used. Several very interesting techniques provided the four elementary operations addition, subtraction, multiplication and division. Many of these calculators were able to perform an additional *fifth operation* which was called *Auflaufenlassen* or the *running total*. The input register of such a machine had perhaps 10 or 12 decimal digits. The result register was much wider and had perhaps 30 digits. It was a fixed-point register which could be shifted back and forth relative to the input register. This allowed a continuous accumulation of numbers and of products of numbers into different positions of the result register. Fixed-point accumulation was thus error free. See Figures 8.1(a)–(d).

On all these computers the result register is much wider than the input data register. The two calculators displayed in the Figures 8.1(c) and (d) show more than one long result register. This allowed the simultaneous accumulation of different long sums.

This fifth arithmetic operation was the fastest way to use the computer. It was applied as often as possible. No intermediate results needed to be written down and typed in again for the next operation. No intermediate roundings or normalizations had to be performed. No error analysis was necessary. As long as no underflow or overflow occurred, which would be obvious and visible, the result was always exact. It was independent of the order in which the summands were added. Rounding was only done, if required, at the very end of the accumulation.

This extremely useful and fast fifth arithmetic operation was not built into the early floating-point computers. It was too expensive for the technologies of those days. Later its superior properties had been forgotten. Thus floating-point arithmetic is still comparatively incomplete.

After Zuse, the electronic computers of the late forties and early fifties represented their data as fixed-point numbers. Fixed-point arithmetic was used because of its superior properties. Fixed-point addition and subtraction are exact. Fixed-point arithmetic with a rather limited word size, however, imposed a scaling requirement. Problems had to be preprocessed by the user so that they could be accommodated by this fixed-point number representation. With increasing speed of computers, the problems that could be solved became larger and larger. The necessary preprocessing soon became an enormous burden.

Thus automatic scaling became generally accepted as floating-point arithmetic. It largely eliminated this burden. A scaling factor is appended to each number in floating-point representation. The arithmetic hardware takes care of the scaling. Exponents are added (subtracted) during multiplication (division). It may result in a big change in the value of the exponent. But multiplication and division are relatively sta-

(a) **Burkhardt Arithmometer** Step drum calculating machine by Arthur Burkhardt & Cie, Glashütte, Germany, 1878.

(b) **Brunsviga** Pin wheel calculating machine BRUNSVIGA, system Trinks, by Brunsviga Maschinenwerke Grimme Natalis & Co., Braunschweig, Germany, 1917.

(c) **MADAS**, by H.W. Egli, Zürich, Switzerland (1936)
(**M**ultiplication, **A**utomatic **D**ivision, **A**ddition, **S**ubtraction).

(d) **MONROE**, model MONROMATIC ASMD (1956),
by Monroe Calculating Machine Company, Inc., Orange, New Jersey, USA.

Figure 8.1. Some mechanical computing devices developed between 1878 and 1956.

ble operations in floating-point arithmetic. Addition and subtraction, on the contrary, are troublesome in floating-point. Early floating point arithmetic was done on early fixed-point machines by programming.

The quality of floating-point arithmetic has been improved over the years. The data format was extended to 64 and even more bits and the IEEE arithmetic standard has finally taken the bugs out of various realizations. Floating-point arithmetic has been used very successfully in the past. Very sophisticated and versatile algorithms and libraries have been developed for particular problems. However, in a general application the result of a floating-point computation is often hard to judge. It can be satisfactory, inaccurate or even completely wrong. Neither the computation itself nor the computed result indicate which one of the three cases has occurred. We illustrate the typical shortcomings by three very simple examples. All data in these examples are IEEE double precision floating-point numbers! For these and other examples see [507]:

Examples. 1. Compute the following, theoretically equivalent expressions:

$$\begin{array}{rcrcrcrcrcr}
10^{20} & + & 17 & - & 10 & + & 130 & - & 10^{20}, \\
10^{20} & - & 10 & + & 130 & - & 10^{20} & + & 17, \\
10^{20} & + & 17 & - & 10^{20} & - & 10 & + & 130, \\
10^{20} & - & 10 & - & 10^{20} & + & 130 & + & 17, \\
10^{20} & - & 10^{20} & + & 17 & - & 10 & + & 130, \\
10^{20} & + & 17 & + & 130 & - & 10^{20} & - & 10.
\end{array}$$

A conventional computer using the double-precision data format of the IEEE floating-point arithmetic standard returns the values 0, 17, 120, 147, 137, -10. These errors come about because the floating-point arithmetic is unable to cope with the digit range required with this calculation. Notice that the data cover less than 4% of the digit range of the double precision data format!

2. Compute the solution of a system of two linear equations $Ax = b$, with

$$A = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The solution can be expressed by the formulas

$$x_1 = \frac{a_{22}}{a_{11}a_{22} - a_{12}a_{21}} \quad \text{and} \quad x_2 = \frac{-a_{21}}{a_{11}a_{22} - a_{12}a_{21}}.$$

A workstation using IEEE double precision floating-point arithmetic returns the *approximate* solution

$$\tilde{x}_1 = 102558961 \quad \text{and} \quad \tilde{x}_2 = 41869520.5,$$

while the correct solution is

$$x_1 = 205117922 \quad \text{and} \quad x_2 = 83739041.$$

After only 4 floating-point operations all digits of the computed solution are wrong. A closer look into the problem reveals that the error happens during the computation of the denominator. This is just the kind of expression that will always be computed exactly by the missing fifth operation.

3. Compute the scalar product of the two vectors a and b with five components each:

$$\begin{aligned}
a_1 &= 2.718281828 \cdot 10^{10}, & b_1 &= 1486.2497 \cdot 10^9,\\
a_2 &= -3.141592654 \cdot 10^{10}, & b_2 &= 878366.9879 \cdot 10^9,\\
a_3 &= 1.414213562 \cdot 10^{10}, & b_3 &= -22.37492 \cdot 10^9,\\
a_4 &= 0.5772156649 \cdot 10^{10}, & b_4 &= 4773714.647 \cdot 10^9,\\
a_5 &= 0.3010299957 \cdot 10^{10}, & b_5 &= 0.000185049 \cdot 10^9.
\end{aligned}$$

The correct value of the scalar product is $-1.00657107 \cdot 10^8$. IEEE-double precision arithmetic delivers $+4.328386285 \cdot 10^9$ so even the sign is incorrect. Note that no vector element has more than 10 decimal digits. IEEE arithmetic computes with a word size that corresponds to about 16 decimal digits. Only 9 floating-point operations are used.

Problems that can be solved by computers become larger and larger. Today fast computers are able to execute several billion floating-point operations every second. This number exceeds the imagination of any user. Traditional error analysis of numerical algorithms is based on estimates of the error of each individual arithmetic operation and on the propagation of these errors through a complicated algorithm. It is simply no longer possible to expect that the error of such computations can be controlled by the user. There remains no alternative to further developing the computer's arithmetic to enable it to control and validate the computational process.

Computer technology is extremely powerful today. It allows solutions which even an experienced computer user may be totally unaware of. Floating-point arithmetic which may fail in simple calculations, as illustrated above, is no longer adequate to be used exclusively in computers of such gigantic speed for huge problems. The reintroduction into computers of the fifth arithmetic operation, the exact scalar product, is a step which is long overdue. A central and fundamental operation of numerical analysis which can be executed exactly with only modest technical effort should indeed always be executed exactly and no longer only *approximately*. With the exact scalar product all the valuable properties which have been listed in connection with the old mechanical calculators return to the modern digital computer.

> *The exact scalar product is the fastest way to use the computer. It should be applied as often as possible. No intermediate results need to be stored and*

> *read in again for the next operation. No intermediate roundings and normalizations have to be performed. No intermediate overflow or underflow can occur. No error analysis is necessary. The result is always exact. It is independent of the order in which the summands are added. Rounding is only done, if required, at the very end of the accumulation.*

This chapter pleads for extending floating-point arithmetic by the exact scalar product as the fifth elementary operation. This combines the advantages of floating-point arithmetic (no scaling requirement) with those of fixed-point arithmetic (fast and exact accumulation of numbers and of single products of numbers even for very long sums). The exact scalar product reintegrates the advantages of fixed-point arithmetic into digital computing and floating-point arithmetic. It is obtained by putting a capability into modern computer hardware which was already available on calculators before the electronic computer came on stage.

This chapter claims that, and explains how, scalar products, *the source data of which are floating-point numbers*, can always be exactly computed. In the old days of computing (1950–1980) computers often provided sloppy arithmetic in order to be fast. This was "justified" by explaining that the last bit was incorrect in many cases, due to rounding errors. So why should the arithmetic be slowed down or more hardware be invested by computing the best possible answer of the operations under the assumption that the last bit is correct? Today it is often asked: why do we need an exact scalar product? The last bit of the data is often incorrect and a floating-point computation of the scalar product delivers the best possible answer for problems with perturbed data. With the IEEE arithmetic standard this kind of "justification" has been overcome. This allows problems to be handled exactly where the data are exact and the best possible result of an operation is needed. In mathematics it makes a big difference whether a computation is exact for many or most data or for all data! For problems with perturbed (inexact) data, interval arithmetic is the appropriate tool. The exact scalar product extends the accuracy requirements of the IEEE arithmetic standard to all operations in the usual product spaces of computation, to complex numbers, to vectors, matrices, and their interval extensions. If implemented in hardware it brings an essential speed up of these operations, and it allows an easy and very fast realization of multiple or variable precision arithmetic.

In short: Interval arithmetic brings guarantees and mathematics into computing, while the exact scalar product brings speed, dynamic precision, and accuracy. A combination of both is what is needed from a modern computer.

8.3 The Ubiquity of the Scalar Product in Numerical Analysis

In numerical analysis the scalar or dot product is ubiquitous. It is not merely a fundamental operation in all the product spaces mentioned above. The process of residual or defect correction, or of iterative refinement, is composed of scalar products. There are well-known limitations to these processes in floating-point arithmetic. The question of how many digits of a defect can be guaranteed in single, double or extended precision arithmetic has been carefully investigated. With an exact scalar product the defect can always be computed to full accuracy. It is the exact scalar product which makes residual correction effective. It has a direct and positive influence on all iterative solvers of systems of linear equations.

A simple example may illustrate the advantages of what has been said: Solving a system of linear equations $Ax = b$ is the central task of numerical computing. Large linear systems can only be solved iteratively. Iterative system solvers repeatedly compute the defect d (sometimes called the residual) $d := b - A\tilde{x}$ of an approximation $\tilde{x}$. It is well known that the error e of the approximation $\tilde{x}$ is a solution of the same system with the defect as the right hand side: $Ae = d$. If $\tilde{x}$ is already a good approximation of the solution, the computation of the defect suffers from cancellation in floating-point arithmetic, and if the defect is not computed correctly the computation of the error does not make sense. In the computation of the defect it is essential that, although $\tilde{x}$ is just an approximation of the solution x^*, $\tilde{x}$ is assumed to be an exact input and that the entire expression for the defect $b - A\tilde{x}$ is computed as a single exact scalar product. This procedure delivers the defect to full accuracy and by that also to multiple precision accuracy. Thus the defect can be read to two or three or four fold precision as necessary in the form $d = d_1 + d_2 + d_3 + d_4$ as a long real variable. The computation can then be continued with this quantity. This often has positive influence on the convergence speed of the linear system solver [165, 166, 167, 189]. It is essential to understand that apart from the exact scalar product, all operations are performed in double precision arithmetic and thus are very fast. If the exact scalar product is supported by hardware it is faster than a conventional scalar product in floating-point arithmetic[1].

But also direct solvers of systems of linear equations profit from computing the defect to full accuracy. The step of verifying the correctness of an approximate solution is based on an accurate computation of the defect. If a first verification attempt fails, a good enough approximation can be computed with the exact scalar product. See Section 9.5.

The so-called *Krawczyk-operator* which is used to verify the correctness of an approximate solution is able to solve the problem only in stable situations. Matrices from the so-called Matrix Market usually are very ill conditioned. In such cases the Krawczyk-operator almost never finds a verified answer. Similarly, for instance, in the case of a Hilbert matrix of dimension greater than eleven the Krawczyk-operator always fails to find a verified solution. In all these cases, however, the Krawczyk-operator recognizes that it cannot solve the problem and then automatically calls a

[1] An iterative method which converges to the solution in infinite precision arithmetic often converges more slowly or even diverges in finite precision arithmetic.

more powerful operator, the so-called *Rump-operator* which then in almost all cases solves the problem satisfactorily.

The Krawczyk-operator first computes an approximate inverse R of the matrix A and then iterates with the matrix $I - RA$, where I is the identity matrix. The Rump-operator inverts the product RA in floating-point arithmetic again and then multiplies its approximate inverse by RA. This product is computed with the exact scalar product and from that it can be read to two or three or four fold precision as necessary as a long-real matrix. The iteration then is continued with the matrix $I - (RA)^{-1}RA$. It is essential to understand that even in the *Rump-algorithm* all arithmetic operations except for the (very fast) exact scalar product are performed in double precision arithmetic and thus are very fast. This linear system solver is an essential ingredient of many other problem solving routines with automatic result verification. It is essential to understand how it works.

To be successful interval arithmetic has to be complemented by some easy way to use multiple or variable precision arithmetic. The fast and exact scalar product is the tool to provide this very easily.

With the exact scalar product quadruple or other multiple precision arithmetic can easily be provided on the computer. This enables the use of higher precision operations in numerically critical parts of a computation. It helps to increase software reliability. A multiple precision number is represented as an array of floating-point numbers. The value of this number is the sum of its components. The number can be represented in the long register in the arithmetic unit. Addition and subtraction of multiple precision numbers can easily be performed in this register. Multiplication of two such numbers is simply a sum of products. It can be computed easily and fast by means of the exact scalar product. For instance, using fourfold precision the product of two such numbers $a = (a_1 + a_2 + a_3 + a_4)$ and $b = (b_1 + b_2 + b_3 + b_4)$ is obtained by

$$\begin{aligned} a \cdot b &= (a_1 + a_2 + a_3 + a_4) \cdot (b_1 + b_2 + b_3 + b_4) \\ &= a_1b_1 + a_1b_2 + a_1b_3 + a_1b_4 + a_2b_1 + \ldots + a_4b_3 + a_4b_4 \\ &= \sum_{i=1}^{4} \sum_{j=1}^{4} a_i b_j . \end{aligned}$$

The result is independent of the sequence in which the summands are added.

Several of the XSC-languages, see for instance [288], provide intrinsic data types `l-real`, `l-interval`, etc., which are implemented by the exact scalar product. For details see Section 9.7 of this book and [392]. The value of a variable of type long is the sum of its components. Addition and subtraction of such multiple precision data can easily be performed in the long register. Multiplication of two variables of this type can be computed easily and fast by the exact scalar product. Division is performed iteratively. The multiple precision data type `long` is controlled by a global variable called `stagprec` (staggered precision). If `stagprec` is 1, the long type is identical to its component type. If, for instance, `stagprec` is 4 each variable of this type consists of an array of four variables of its component type. Again its value is the sum of its components. The global variable `stagprec` can be increased or decreased at any place in the program. Thus a loop can be executed starting with precision one and the precision can be increased (or decreased) in each repetition as required by the user. Thus no program changes are necessary to benefit from increased precision. The user or the computer itself can choose the precision which optimally fits the problem. The elementary functions are also available for the types `l-real` and `l-interval` [114, 115, 311, 312, 313]. If `stagprec` is 2, a data type is encountered which is occasionally denoted as `double-double` or quadruple precision.

If one runs out of precision in a certain problem class, one often runs out of quadruple precision very soon as well. It is preferable and simpler, therefore, to provide a high speed basis for enlarging the precision rather than to provide any fixed higher precision or to simulate higher precision in software. A hardware implementation of a full quadruple precision arithmetic is more costly than an implementation of the exact scalar product. The latter only requires fixed-point accumulation of the products. On the computer, there is only one standardised floating-point format that is double precision.

For many applications it is necessary to compute the value of the derivative of a function. Newton's method in one or several variables is a typical example of this. Modern numerical analysis solves this problem by automatic or algorithmic differentiation. The so-called reverse mode is a very fast method of automatic differentiation. It computes the gradient, for instance, with at most five times the number of operations needed to compute the function value. The memory overhead and the spatial complexity of the reverse mode can be significantly reduced by the exact scalar product if this is considered as a single, always correct, basic arithmetic operation in the vector spaces [555, 556]. The very powerful methods of global optimization [492, 494] are impressive applications of these techniques.

Many other applications require that rigorous mathematics can be done with the computer using floating-point arithmetic. As an example, this is essential in simulation runs (eigenfrequencies of a large generator, fusion reactor, simulation of nuclear explosions) or mathematical modelling where the user has to distinguish between computational artefacts and genuine reactions of the model. The model can only be developed systematically if errors resulting from the computation can be excluded.

Nowadays computer applications are of immense variety. Any discussion of where a dot product computed in quadruple or extended precision arithmetic can be used to substitute for the exact scalar product is superfluous. Since the former can fail to produce a correct answer an error analysis is needed for all applications. This can be left to the computer. As the scalar product can always be executed exactly with moderate technical effort it should indeed always be executed exactly. An error analysis thus becomes irrelevant. Furthermore, the same result is always obtained on different computer platforms. An exact scalar product eliminates many rounding errors in numerical computations. It stabilises these computations and speeds them up as well. It is the necessary complement to floating-point arithmetic.

8.4 Implementation Principles

We begin with a brief review of the definition and the notation of floating-point numbers.

A normalized floating-point number x (in signed-magnitude representation) is a real number of the form $x = \circ m\, b^e$. Here $\circ \in \{+, -\}$ is the sign of the number, b is the base of the number system in use and e is the exponent. The base b is an integer greater than unity. The exponent e is an integer between two fixed integer bounds $e1$ and $e2$, and in general $e1 < 0 < e2$. The mantissa (or significand) m is of the form

$$m = \sum_{i=1}^{l} d_i \cdot b^{-i}.$$

The d_i are the digits of the mantissa. They have the property $d_i \in \{0, 1, \ldots, b-1\}$ for all $i = 1(1)l$ and $d_1 \neq 0$. Without this last condition floating-point numbers are said to be unnormalized. The set of normalized floating-point numbers does not contain zero. For a unique representation of zero we assume the mantissa and the exponent to be zero. Thus a floating-point system depends on the four constants $b, l, e1$ and $e2$. We denote it by $R = R(b, l, e1, e2)$. On occasion we shall use the abbreviations sign(x), mant(x) and exp(x) to denote the sign, mantissa and exponent of x respectively.

Now we turn to our task. Let $a = (a_i)$ and $b = (b_i)$, $i = 1(1)n$, be two vectors with n components which are floating-point numbers, i.e.,

$$a_i, b_i \in R(b, l, e1, e2), \text{ for } i = 1(1)n.$$

We are going to compute the two results (scalar products):

$$s := \sum_{i=1}^{n} a_i \cdot b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n,$$

and

$$c := \square \sum_{i=1}^{n} a_i \cdot b_i = \square\,(a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n) = \square\, s,$$

where all additions and multiplications are the operations for real numbers and $\square$ is a rounding symbol representing rounding to nearest, towards zero, upwards or downwards.

We shall discuss circuits for the hardware realization of these operations for processors like personal computers, workstations, mainframes, supercomputers and digital signal processors. The differences in the circuitry for these various processors are dictated by the speed with which the processor delivers the vector components a_i and b_i, $i = 1, 2, \ldots, n$, to the arithmetic or scalar product unit.

Since a_i and b_i are floating-point numbers with a mantissa of l digits, the products $a_i \cdot b_i$ in the sums for s and c are floating-point numbers with a mantissa of $2l$ digits. The exponent range of these numbers doubles also, i.e., $a_i \cdot b_i \in R(b, 2l, 2e1, 2e2)$. All these summands can be taken into a fixed-point register of length $2e2 + 2l + 2|e1|$ without loss of information, see Figure 8.2: If one of the summands has an exponent 0, its mantissa can be taken into in a register of length $2l$. If another summand has exponent 1, it can be treated as having an exponent 0 if the register provides further digits on the left and the mantissa is shifted one place to the left. An exponent -1 in one of the summands requires a corresponding shift to the right. The largest exponents in magnitude that may occur in the summands are $2e2$ and $2|e1|$. So any summand can be treated as having an exponent 0 and be taken into a fixed-point register of length $2e2 + 2l + 2|e1|$ without loss of information.

In the following two subsections we shall detail two principal solutions to the problem. The first solution uses a long adder and a long shift. The second solution uses a short adder and some local memory or register space in the arithmetic unit. At first sight both of these principal solutions seem to lead to relatively slow hardware circuits. However more refined studies will later show that very fast circuits can be devised for both methods and for the various processors mentioned above. A first step in this direction is the provision of the very fast carry resolution scheme described in Section 8.4.4.

Actually it is a central result of this study that, for all processors under consideration, circuits for the computation of the exact scalar product are available for which virtually no unoverlapped computing time for the execution of the arithmetic is needed. In a pipeline, the arithmetic can be done within the time the processor needs to read the data into the arithmetic unit. This means that no other method of computing the scalar product can be faster, in particular, not even a conventional computation of scalar products in floating-point arithmetic which may lead to an incorrect answer. Once more we emphasize the fact that the methods to be discussed here compute the scalar product of two floating-point vectors of arbitrary finite length without loss of information or with only one rounding at the very end of the computation.

8.4.1 Long Adder and Long Shift

If the register shown in Figure 8.2 is built as an accumulator with an adder, all summands could be added in without loss of information. To accommodate possible overflows, it is convenient to provide a few, say k, more digits of base b on the left. With such an accumulator, every such sum or scalar product can be added in without loss of information. As many as b^k overflows may occur and be accommodated without loss of information. In the worst case, presuming every sum causes an overflow, we can accommodate sums with $n \leq b^k$ summands.

A teraflops computer would perform about 10^{20} operations in 10 years. So 20 decimal or about 65 binary digits certainly are a safe and reasonable upper bound for k. Thus, the long accumulator and the long adder consist of $L = k + 2e2 + 2l + 2|e1|$ digits of base b. The summands are shifted to the proper position and added in. See Figure 8.2. Fast carry resolution techniques will be discussed later. The final sums s and c are supposed to be in the single exponent range $e1 \leq e \leq e2$, otherwise c is not representable as a floating-point number and the problem has to be scaled.

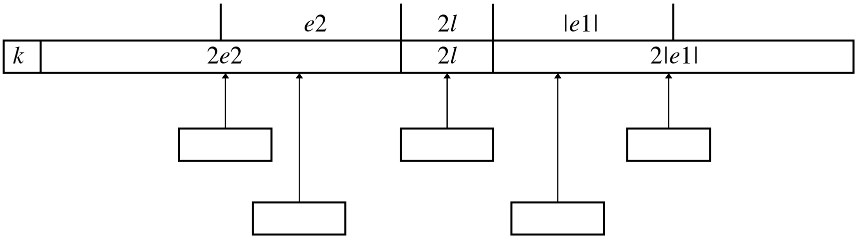


Figure 8.2. Long register with long shift for exact scalar product accumulation.

8.4.2 Short Adder with Local Memory on the Arithmetic Unit

In a scalar product computation the summands are all of length $2l$. So actually the long adder and long accumulator may be replaced by a short adder and a local store (register space) of size L on the arithmetic unit. The local store is organized in words of length l or l', where l' is a power of 2 and slightly larger than l (for instance $l = 53$ bits and $l' = 64$ bits). Since the summands are of length $2l$, they fit into a part of the local store of length $3l'$. This part of the store is determined by the exponent of the summand. We load this part of the store into an accumulator of length $3l'$. The summand mantissa is placed in a shift register and is shifted to the correct position as determined by the exponent. Then the shift register contents are added to the contents of the accumulator. Figure 8.3.

An addition into the accumulator may produce a carry. As a simple method to accommodate carries, we enlarge the accumulator on its left end by a few more digit positions. These positions are filled with the corresponding digits of the local store. If not all of these digits equal $b - 1$ for addition (or zero for subtraction), they will accommodate a possible carry of the addition (or borrow in the case of subtraction). Of course, it is possible that all these additional digits are $b - 1$ (or zero). Then a loop can be provided that takes care of the carry and adds it to (subtracts it from) the next digits of the local store. This loop may need to be traversed several times. Other carry (borrow) handling processes are possible and will be dealt with later. This completes our sketch of the second method for an exact computation of scalar products using a short adder and some local store on the arithmetic unit. See Figure 8.3.

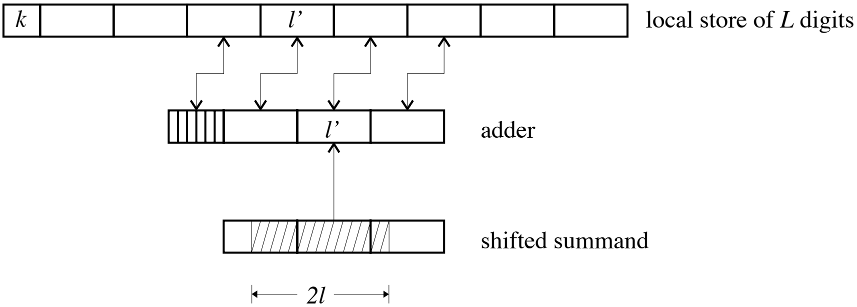


Figure 8.3. Short adder and local store on the arithmetic unit for exact scalar product accumulation.

8.4.3 Remarks

The scalar product is very frequent in scientific computing. The two solutions described in the last two subsections are both simple, straightforward and mature.

Remark 8.1. The purpose of the k digits on the left end of the register in Figure 8.2 and Figure 8.3 is to accommodate possible overflows. The only numbers that are added to this part of the register are plus or minus unity. So this part of the register can just be treated as a counter by an incrementer/decrementer. ■

Remark 8.2. The final result of a scalar product computation is assumed to be a floating-point number with an exponent in the range $e1 \leq e \leq e2$. During the computation, however, summands with an exponent outside of this range may well occur. The remaining computation then has to cancel all the extra digits. However, in a normal scalar product computation, the register space outside the range $e1 \leq e \leq e2$ will be little used. The conclusion should not be drawn from this consideration that the register size can be restricted to the single exponent range in order to save some silicon area. This would require the implementation of complicated exception handling routines which finally require as much silicon but do not solve the problem in principle. ■

Remark 8.3. We emphasize once more that the number of digits, L, needed for the register to compute scalar products of two vectors exactly only depends on the floating-point data format. In particular it is independent of the number n of components of the two vectors to be multiplied.

As samples we calculate the register width L for a few typical and frequently used floating-point data formats:

(a) IEEE-arithmetic single precision:
$b = 2$; word length: 32 bits; sign: 1 bit; exponent: 8 bits; mantissa: $l = 24$ bits; exponent range: $e1 = -126$, $e2 = 127$, binary.
$L = k + 2e2 + 2l + 2|e1| = k + 554$ bits.
With $k = 86$ bits we obtain $L = 640$ bits. This register can be represented by 10 words of 64 bits.

(b) /370 architecture, long data format:
$b = 16$; word length: 64 bits; sign: 1 bit; mantissa: $l = 14$ hex digits; exponent range: $e1 = -64$, $e2 = 63$, hexadecimal.
$L = k + 2e2 + 2l + 2|e1| = k + 282$ digits of base 16.
With $k = 88$ bits we obtain $L = 88 + 4 \cdot 282 = 1216$ bits. This register can be represented by 19 words of 64 bits.

(c) IEEE-arithmetic double precision:
$b = 2$; word length: 64 bits; sign: 1 bit; exponent: 11 bits; mantissa: $l = 53$ bits; exponent range: $e1 = -1022$, $e2 = 1023$, binary.
$L = k + 2e2 + 2l + 2|e1| = k + 4196$ bits.
With $k = 92$ bits we obtain $L = 4288$ bits. This register can be represented by 67 words of 64 bits.

These samples show that the register size (at a time where memory space is measured in gigabits and gigabytes) is modest in all cases. It grows with the exponent range of the data format. If this range should be extremely large, as for instance in the case of an extended precision floating-point format, only an inner part of the register would be supported by hardware. The outer parts which are used very rarely could be simulated in software. The long data format of the /370 architecture covers a range of about 10^{-75} to 10^{75}, which is very modest. This architecture dominated the market for more than 25 years and most problems could conveniently be solved with machines of this architecture within this range of numbers. ■

Remark 8.4. Multiplication is often considered to be more complex than addition. In modern computer technology this is no longer the case. Very fast circuits for multiplication using carry-save-adders (Wallace tree) are available and common. They nearly equalize the time to compute a sum and a product of two floating-point numbers. In a scalar product computation a large number of products is usually to be computed. The multiplier is able to produce these products very quickly. In a balanced scalar product unit the accumulation should be able to absorb a product in about the same time the multiplier takes to produce it. Therefore, measures have to be taken to equalize the speed of both operations. Because of a possible long carry propagation the accumulation seems to be the more complicated process. ■

Remark 8.5. Techniques to implement the exact scalar product on machines which do not provide enough register space on the arithmetic logical unit will be discussed later in this chapter. ■

8.4.4 Fast Carry Resolution

Both the solutions for our problem which have been sketched above seem at first glance to be slow. The first solution requires a long shift which is necessarily slow. The addition over perhaps 4000 bits is slow also, in particular if a long carry propagation is necessary. For the second solution, five steps have to be carried out: 1. read from the local store, 2. perform the shift, 3. add the summand, 4. resolve the carry, possibly by loops, and 5. write the result back into the local store. Again the carry resolution may be very time consuming.

As a first step to speed up both solutions, we discuss a technique which allows a very fast carry resolution. Actually a possible carry can be accommodated while the product, the addition of which might produce a carry, is still being computed.

Both solutions require a long register in which the final sum in a scalar product computation is built up. Henceforth we shall call this register a *complete register* and CR for short.[2] It consists of L digits of base b. The CR is a fixed-point register wherein any sum of floating-point numbers and of simple products of floating-point numbers can be represented without error.

To be more specific we now assume that we are using the double precision data format of the IEEE-arithmetic standard 754. See case (c) of Remark 8.3. As soon as the principles are clear, the technique can easily be applied to other data formats. The mantissa here consists of $l = 53$ bits. We assume additionally that the CR that appears in both solutions is subdivided into words of $l' = 64$ bits. The mantissa of the product $a_i \cdot b_i$ is then 106 bits wide. It touches at most three consecutive 64-bit words of the CR which are determined by the exponent of the product. A shifter then aligns the 106 bit product into the correct position for the subsequent addition into the three consecutive words of the CR. This addition may produce a carry (or a borrow in the case of subtraction). The carry is absorbed by the next more significant 64 bit word of the CR in which not all digits are 1 (or 0 for subtraction). Figure 8.4(a). For fast identification of this word two information bits or flags are appended to each CR word. Figure 8.4(b). One of these bits, the *all bits 1* flag, is set to 1 if all 64 bits of the register word are 1. This means that a carry will propagate through the entire word. The other bit, the *all bits 0* flag, is set to 0, if all 64 bits of the register word are 0. This means that in the case of subtraction a borrow will propagate through the entire word.

During the addition of a product into three consecutive words of the CR, a search is started for the next more significant word of the CR where the *all bits 1* flag is not set.

[2] For this use of *complete* see Section 8.7.

This is the word which will absorb a possible carry. If the addition generates a carry, this word must be incremented by one and all intermediate words must be changed from all bits 1 to all bits 0. The easiest way to do this is simply to switch the flag bits from *all bits 1* to *all bits 0* with the additional semantics that if a flag bit is set, the appropriate constant (all bits 0 or all bits 1) must be generated instead of reading the CR word contents when fetching a word from the CR, Figure 8.4(b). Borrows are handled in an analogous way.

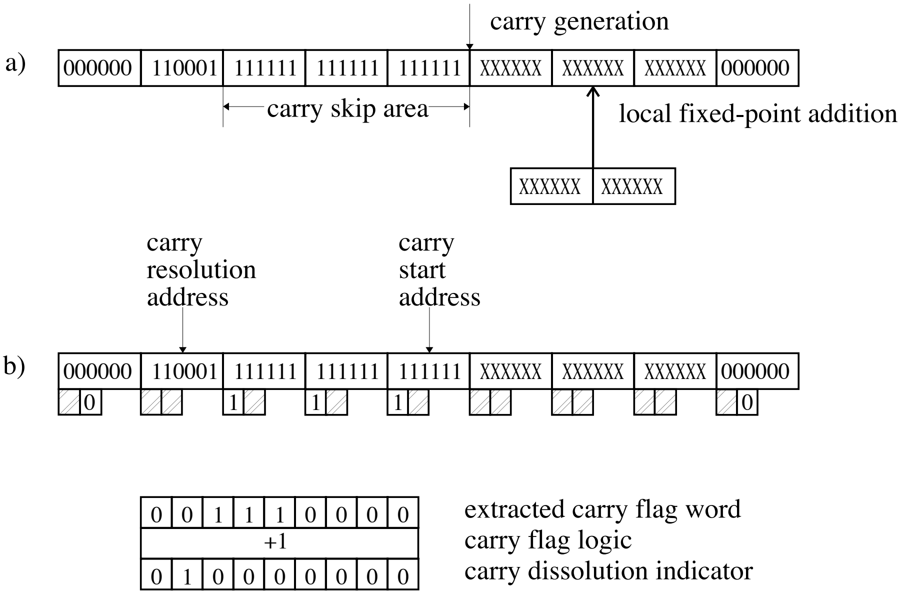

a)
carry generation
000000 110001 111111 111111 111111 XXXXXX XXXXXX XXXXXX 000000
carry skip area
local fixed-point addition
XXXXXX XXXXXX
carry resolution address
carry start address
b)
000000 110001 111111 111111 111111 XXXXXX XXXXXX XXXXXX 000000
0 1 1 1 0
0 0 1 1 1 0 0 0 0
+1
0 1 0 0 0 0 0 0 0
extracted carry flag word
carry flag logic
carry dissolution indicator

Figure 8.4. Fast carry resolution.

This carry handling scheme allows a very fast carry resolution. The generation of the carry resolution address is independent of the addition of the product, so it can be performed in parallel. At the same time, a second set of flags is set up for the case that a carry is generated. In this case the carry is added into the appropriate word and the second set of flags is copied into the former flag word.

Simultaneously with the multiplication of the mantissas of a_i and b_i their exponents are added. This is just an eleven bit addition. The result is available very quickly. It delivers the exponent of the product and the address for its addition. By using the flags, the carry resolution address can be determined and the carry word can be incremented/decremented as soon as the exponent of the product is available. It could be available before the multiplication of the mantissas is finished. If the accumulation of the product then produces a carry, the incremented/decremented carry word is written back into the CR, otherwise nothing is changed.

This very fast carry resolution technique could be used in particular for the computation of short scalar products which occur, for instance, in the computation of the real and imaginary parts of the product of two complex floating-point numbers. A long scalar product, however, is usually performed in a pipeline. Then, during the multiplication, the previous product is added in. It seems to be reasonable, then, to wait with the carry resolution until the addition of the previous product is actually finished.

8.5 Scalar Product Computation Units (SPUs)

Having discussed the two principal solutions for exact scalar product computation as well as a very fast carry handling scheme, we now turn to a more detailed design of scalar product computation units for various processors. These units will be called SPUs, which stands for Scalar Product Units. If not otherwise mentioned we assume throughout this section that the data are stored in the double precision format of the IEEE-arithmetic standard 754. There the floating-point word has 64 bits and the mantissa 53 bits. A central building block for the SPU is the *complete register*, the CR. It is a fixed-point register wherein any sum of floating-point numbers and of simple products of floating-point numbers can be represented without error. The SPU allows the computation of scalar products of two vectors with any finite number of floating-point components exactly or with a single rounding at the very end of the computation. As shown in Remark 8.3(c) of Section 8.4.3, the CR consists of 4288 bits. It can be represented by 67 words of 64 bits.

The scalar product is frequently used in scientific computation so its execution needs to be fast. All circuits to be discussed in this section perform the scalar product in a pipeline which simultaneously executes the following steps:

(a) read the two factors a_i and b_i to perform a product,

(b) compute the product $a_i \cdot b_i$ to the full double length, and

(c) add the product $a_i \cdot b_i$ to the CR.

Step (a) turns out to be the bottleneck of this pipeline. Therefore, we shall develop different circuits for computers which are able to read the two factors a_i and b_i into the SPU in four or two or one portion. The latter case will be discussed in Section 8.8. Step (b) produces a product of 106 bits. It maps onto at most three consecutive words of the CR. The address of these words is determined by the product's exponent. In step (c) the 106 bit product is added to the three consecutive words of the CR.

8.5.1 SPU for Computers with a 32 Bit Data Bus

Here we consider a computer which is able to read the data into the arithmetic logical unit and/or the SPU in portions of 32 bits. An older personal computer (until ca. 2003) is typical of this kind of computer.

The first solution with an adder and a shifter for the full CR of 4288 bits would be expensive. So the SPU for these computers is built upon the second solution (see Figure 8.5).

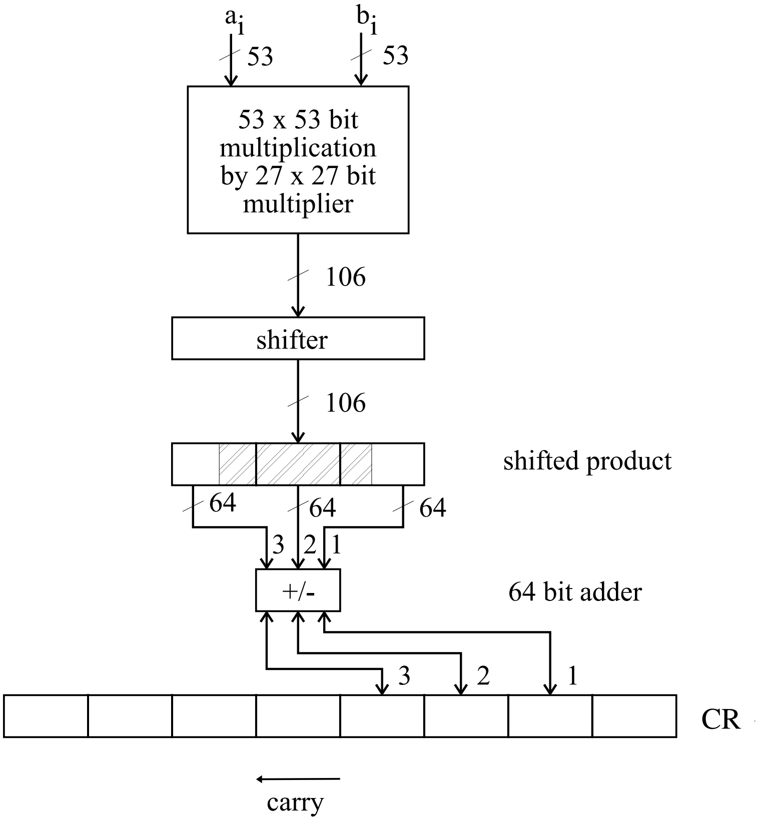


Figure 8.5. Accumulation of a product to the CR by a 64 bit adder.

For the computation of the product $a_i \cdot b_i$ the two factors a_i and b_i are to be read. Both consist of 64 bits. Since the data can only be read in 32 bit portions, the unit has to read four times. We assume that with the necessary decoding this can be done in eight cycles. See Figure 8.6.

This is rather slow and turns out to be the bottleneck for the whole pipeline. In a balanced SPU the multiplier should be able to produce a product and the adder should be able to accumulate the product in about the same time the unit needs to read the data. Therefore, it suffices to provide a 27 × 27 bit multiplier. It computes the 106 bit product of the two 53 bit mantissas of a_i and b_i by 4 partial products. The subsequent addition of the product into the three consecutive words of the CR is performed by an adder of 64 bits. The appropriate three words of the CR are loaded into the adder one

cycle	read	mult/shift	accumulate			
	read a^1_{i-1}					
	read a^2_{i-1}					
	read b^1_{i-1}					
	read b^2_{i-1}					
	read a^1_{i}					
	read a^2_{i}	$c_{i-1} := a_{i-1} \cdot b_{i-1}$				
	read b^1_{i}	$c_{i-1} :=$ shift (c_{i-1})				
	read b^2_{i}					
	read a^1_{i+1}		load1			
			add/sub	load2		
	read a^2_{i+1}	$c_i := a_i \cdot b_i$	store1	add/sub	load3	
				store2	add/sub	load carry
	read b^1_{i+1}	$c_i :=$ shift (c_i)			store3	inc/dec
						store carry
	read b^2_{i+1}					store flags
	read a^1_{i+2}		load1			
			add/sub	load2		
	read a^2_{i+2}	$c_{i+1} := a_{i+1} \cdot b_{i+1}$	store1	add/sub	load3	
				store2	add/sub	load carry
	read b^1_{i+2}	$c_{i+1} :=$ shift (c_{i+1})			store3	inc/dec
						store carry
	read b^2_{i+2}					store flags
	read a^1_{i+3}		load1			
			add/sub	load2		
	read a^2_{i+3}	$c_{i+2} := a_{i+2} \cdot b_{i+2}$	store1	add/sub	load3	
				store2	add/sub	load carry
	read b^1_{i+3}	$c_{i+2} :=$ shift (c_{i+2})			store3	inc/dec
						store carry
	read b^2_{i+3}					store flags

Figure 8.6. Pipeline for the accumulation of scalar products on computers with 32 bit data bus.

after the other and the appropriate portion of the product is added. The sum is written back into the same word of the CR that the portion has been read from. A 64 out of 106 bit shifter must be used to align the product onto the relevant word boundaries. See Figure 8.5. The addition of the three portions of the product into the CR may cause a carry. The carry is absorbed by incrementing (or decrementing in the case of a borrow) a more significant word of the CR as determined by the carry handling scheme.

The pipeline is sketched in Figure 8.6. There, we assume that a dual port RAM is available on the SPU to store the CR. This is usual for register memory. It allows reading from the CR and writing into it simultaneously. Eight machine cycles are needed to read the two 64 bit factors a_i and b_i for a product, including the necessary address decoding. This is also about the time in which the multiplication and the shift can be performed in the second step of the pipeline. The three successive additions and the carry resolution in the third step of the pipeline again can be done in about the same time. See Figure 8.6. Figure 8.7 shows a block diagram for an SPU with 32 bit data bus.

The sum of the exponents of a_i and b_i delivers the exponent of the product $a_i \cdot b_i$. It consists of twelve bits. The six low order (less significant) bits of this sum are used to perform the shift. The more significant bits of the sum deliver the CR address to which the product $a_i \cdot b_i$ has to be added. So the originally very long shift is split into a short shift and an addressing operation. The shifter performs a relatively short shift operation. The addressing selects the three words of the CR for the addition of the product.

The CR RAM needs only one address decoder to find the starting address for an addition. The two more significant parts of the product are added to the contents of the two CR words with the next two addresses. The carry logic determines the word that absorbs the carry. All these address decodings can be hard wired. The result of each of the four additions is written back into the same CR words to which the addition has been executed. The two carry flags appended to each accumulator word are indicated in Figure 8.7. In practice the flags are kept in separate registers.

We stress the fact that in the circuit just discussed virtually no unoverlapped computing time is needed for the arithmetic. In the pipeline the arithmetic is performed in the time that is needed to read the data into the SPU. Here, we assumed this requires eight cycles, allowing both the multiplication and the accumulation to be performed very economically and sequentially by a 27×27 bit multiplier and a 64 bit adder. Both the multiplication and the addition are themselves performed in a pipeline. The arithmetic overlaps with the loading of the data into the SPU.

8.5.2 A Coprocessor Chip for the Exact Scalar Product

A vector arithmetic coprocessor chip XPA 3233 for the PC was developed in a CMOS 0.8 μm VLSI gate array technology at the author's Institute in 1993/94 in collaboration

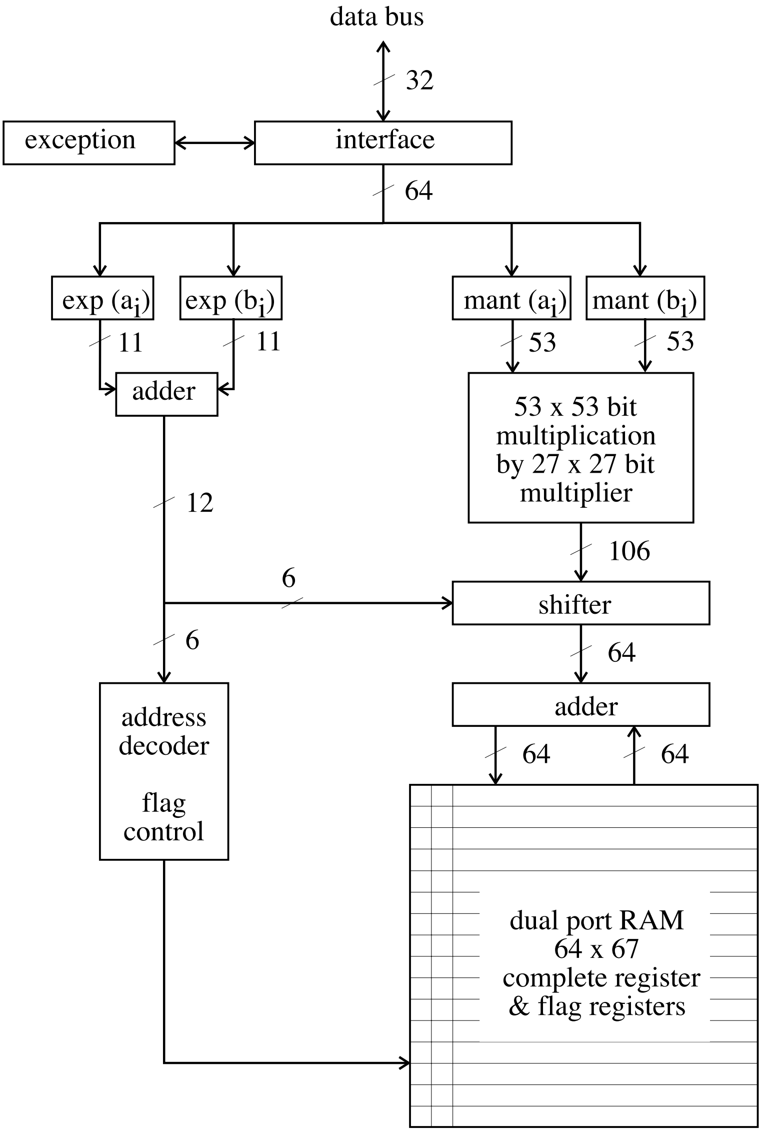


Figure 8.7. Block diagram for an SPU with 32 bit data supply and sequential addition into the CR.

with the Institute for Microelectronics at the Universität Stuttgart and the Institute for Informatics of the Technische Universität Hamburg-Harburg. VHDL and COMPASS design tools were used. For design details see [232, 281, 373] and in particular [61]. The chip is connected with the PC via the PCI-bus. The PCI- and EMC-interfaces are integrated on the chip. With the coprocessor the PC became a vector computer. In its time the chip computed the exact scalar product between two and four times faster than the PC could compute an approximation in floating-point arithmetic. With increasing clock rate of the PC the PCI-bus turned out to be a severe bottleneck. To keep up with the increased speed the SPU must be integrated into the arithmetic logical unit of the processor and interconnected by an internal bus system.

The chip, see Figure 8.8, realizes the SPU that has been discussed in Section 8.5.1 using 207,000 transistors. About 30% of the transistors and the silicon area are used for the CR and the flag registers with the carry resolution logic. The remaining 70% of the silicon area is needed for the PCI/EMC interface and the chip's own multiplier, shifter, adder and rounding unit. All these units would be superfluous if the SPU were integrated into the arithmetic unit of the processor. A multiplier, shifter, adder and rounding unit are already there. Everything just needs to be arranged a little differently. Ultimately the SPU requires fewer transistors and less silicon area than is needed for the exception handling of the IEEE-arithmetic standard. The SPU is logically much more regular and simple. With it a large number of exceptions that can occur in a conventional floating-point computation are avoided. The silicon area would shrink considerably if full custom design were used.

Testing of the coprocessor XPA 3233 was easy. XSC-languages had been available and used since 1980. An identical software simulation of the exact scalar product had also been implemented. Additionally a large number of problem solving routines had been developed and collected in the toolbox volumes [205, 207, 289, 290, 332]. All that had to be done was to change the PASCAL-XSC compiler a little to call the hardware chip instead of its software simulation. Surprisingly, 40% of the chips on the first wafer were correct and, probably due to the high standard of the implementors and their familiarity with the theoretical background, with PASCAL-XSC and the toolbox routines no redesign was necessary. The chips produced results identical to those of the software simulation.

Modern computer technology can provide millions of transistors on a single chip. This allows capabilities to be put into the computer hardware which even an experienced computer user is totally unaware of. Through ignorance of the technology and the design tools and implementation techniques, obvious and easy capabilities are not demanded by mathematicians. The engineer on the other hand, who is familiar with these techniques, is not aware of the consequences for mathematics [232].

There are processors available where the data supply to the arithmetic unit or the SPU is much faster. We discuss the design of a SPU for such processors in the next section and in Section 8.8.

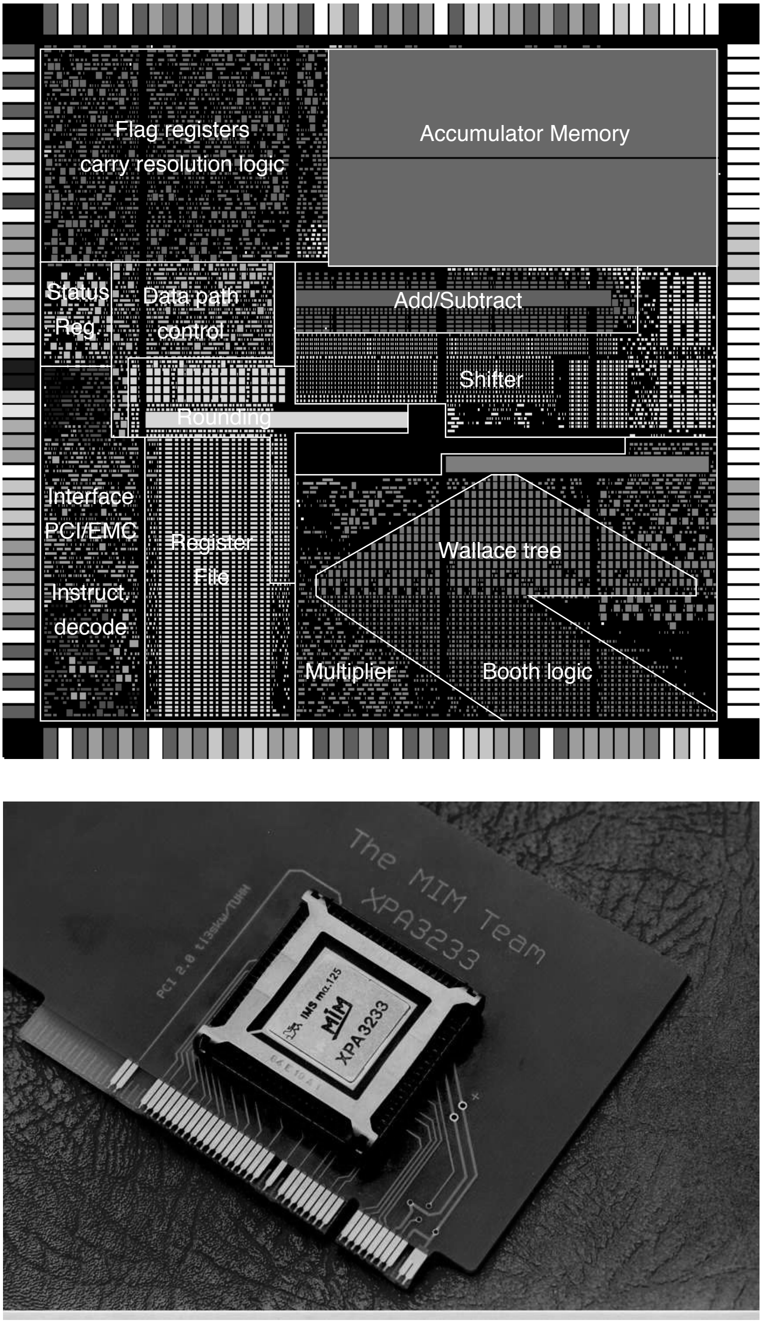


Figure 8.8. Functional units, chip and board of the vector arithmetic coprocessor XPA 3233.

8.5.3 SPU for Computers with a 64 Bit Data Bus

Now we consider a computer which is able to read data into the arithmetic logical unit and/or the SPU in portions of 64 bits. Fast workstations or mainframes are typical of this kind of computer.

Now the time to perform the multiplication and the accumulation overlapped in pipelines as before is no longer available. To keep the execution time for the arithmetic within the time the SPU needs to read its data, we have to invest in more circuitry. For multiplication a 53×53 bit multiplier must now be used. The result is still 106 bits wide and so will go into two or three 64 bit words of the CR. But the addition of the product and the carry resolution now have to be performed in parallel.

If the 106 bit summand spans three consecutive 64 bit words of the CR, a closer look shows that the 22 least significant bits of those three words are never changed by addition of the summand. Thus the adder needs to be 170 bits wide only. Figure 8.9 shows a sketch for the parallel accumulation of a product.

In the circuit a 106 to 170 bit shifter is used. The four additions are to be performed in parallel. So four read/write ports are to be provided for the CR RAM. Sophisticated logic must be used for the generation of the carry resolution address, since this address must be generated very quickly. Again the CR RAM needs only one address decoder to find the starting address for an addition. The more significant parts of the product are added to the contents of the two CR words with the next two addresses. A tree structured carry logic now determines the CR word which absorbs the carry. A very fast hardwired multi-port driver can be designed which allows all 4 CR words to be read into the adder in one cycle.

Figure 8.10 shows the pipeline for this kind of addition. In the figure we assume that two machine cycles are needed to decode and read one 64 bit word into the SPU.

Figure 8.11 shows a block diagram for an SPU with a 64 bit data bus and parallel addition.

We emphasize that again virtually no unoverlapped computing time is needed for the execution of the arithmetic. In a pipeline the arithmetic is performed in the time which is needed to read the data into the SPU. Here, we assume that with the necessary address decoding, this requires four cycles for the two 64 bit factors a_i and b_i for a product. To match the shorter time required to read the data, more circuitry has to be used in the multiplier and the adder.

If the technology is fast enough it may be reasonable to provide a 256 bit adder instead of the 170 bit adder. An adder width of a power of 2 may simplify shifting as well as address decoding. The lower bits of the exponent of the product control the shift operation while the higher bits are directly used as the starting address for the accumulation of the product into the CR.

The two flag registers appended to each CR word are indicated in Figure 8.11 again. In practice the flags are kept in separate registers.

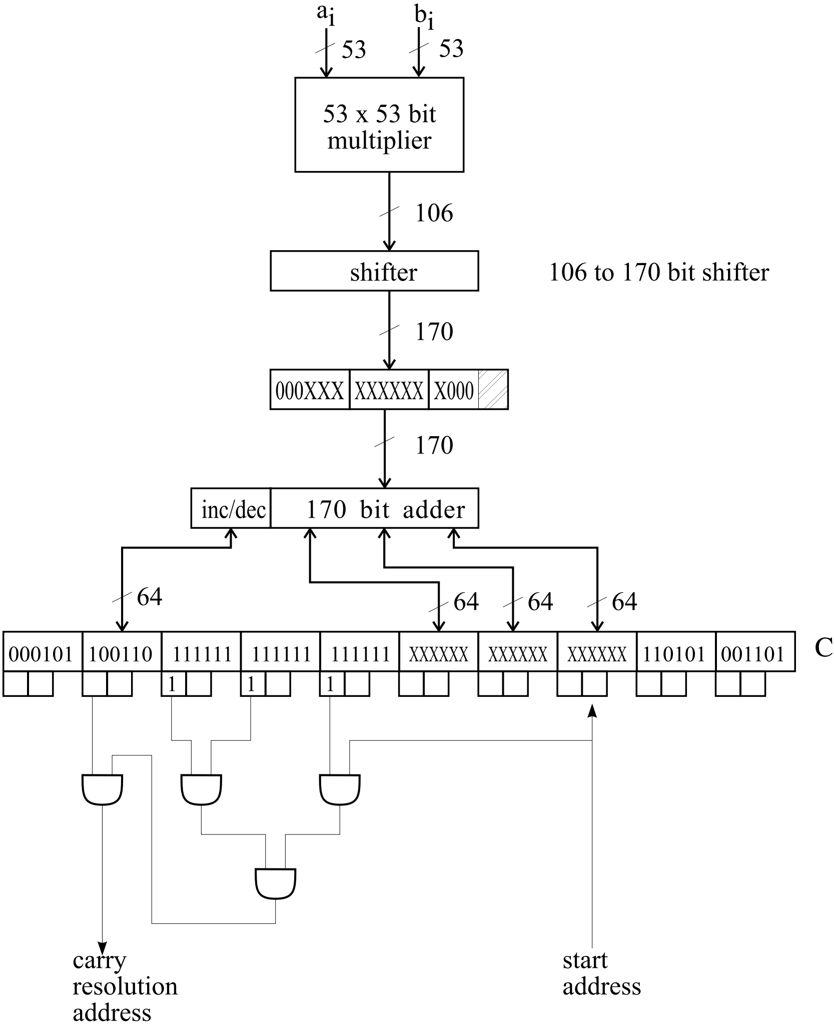


Figure 8.9. Parallel accumulation of a product into the CR.

cycle	read	mult/shift	accumulate
	read a_{i-1}		
	read b_{i-1}		
	read a_i	$c_{i-1} := a_{i-1} \cdot b_{i-1}$	
	read b_i	c_{i-1} := shift (c_{i-1})	
	read a_{i+1}	$c_i := a_i \cdot b_i$	address decoding load
	read b_{i+1}	c_i := shift (c_i)	add/sub c_{i-1} store & store flags
	read a_{i+2}	$c_{i+1} := a_{i+1} \cdot b_{i+1}$	address decoding load
	read b_{i+2}	c_{i+1} := shift (c_{i+1})	add/sub c_i store & store flags
	read a_{i+3}	$c_{i+2} := a_{i+2} \cdot b_{i+2}$	address decoding load
	read b_{i+3}	c_{i+2} := shift (c_{i+2})	add/sub c_{i+1} store & store flags

Figure 8.10. Pipeline for the accumulation of scalar products.

8.6 Comments

8.6.1 Rounding

If the result of an exact scalar product is needed later in a program, the contents of the CR must be put into user memory. How this can be done will be discussed later in this section.

If not processed any further the exact result of a scalar product computation usually has to be rounded into a floating-point number or a floating-point interval. The flag bits that are used for the fast carry resolution can be used for the rounding of the CR contents also. By looking at the flag bits, the leading result word in the CR can easily be identified. This and the next CR word are needed to compose the mantissa of the result. This 128 bit quantity must then be shifted to form a normalized mantissa of an IEEE-arithmetic double precision number. The shift length can be extracted by looking at the leading result word in the CR with the same procedure which identified it by looking at the flag bit word.

For the correct rounding downwards (or upwards) it is necessary to check whether any of the discarded bits is a one. This is done by testing the remaining bits of the 128 bit quantity in the shifter and by looking at the *all bits 0* flags of the following CR words. This information is then used to control the rounding.

Rounding is only needed at the very end of a scalar product computation. If a large number of products has been accumulated the contribution of the rounding to the

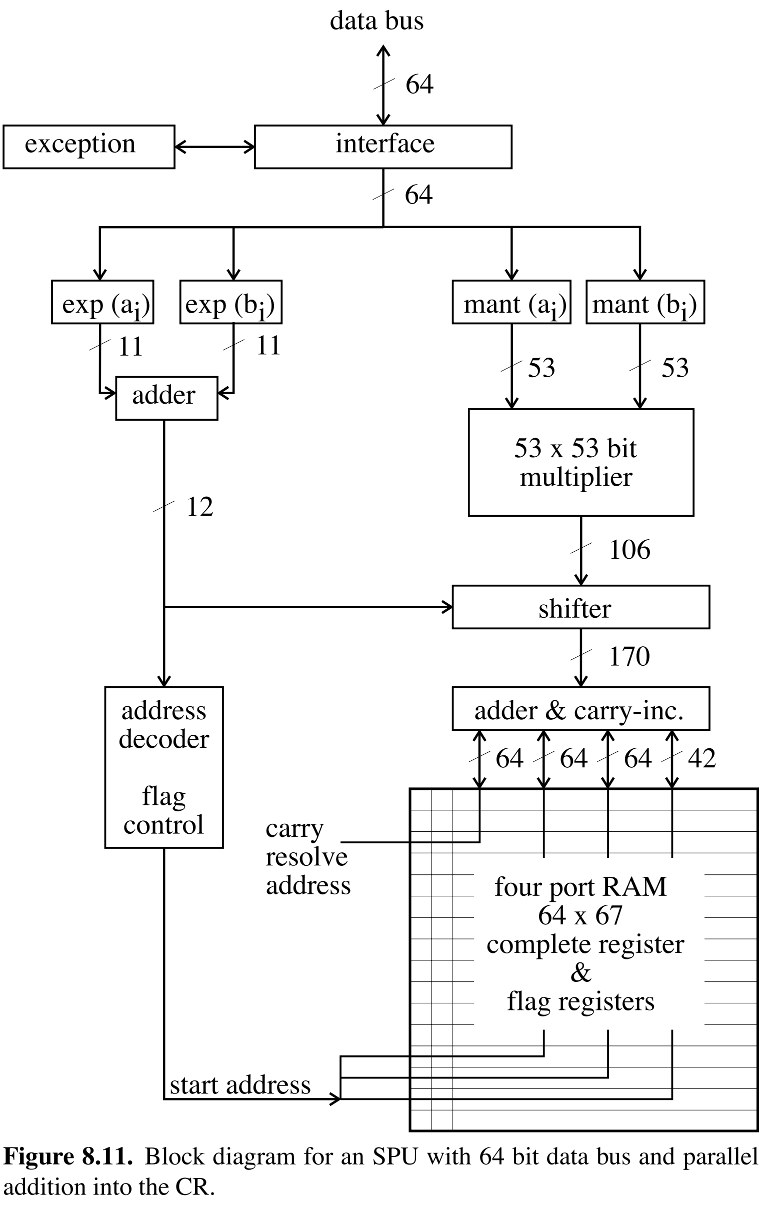


Figure 8.11. Block diagram for an SPU with 64 bit data bus and parallel addition into the CR.

computing time is negligible. However, if a short scalar product or a single floating-point addition or subtraction has to be carried out by the SPU, a very fast rounding procedure is essential for the speed of the overall application.

The rounding speed depends heavily on the speed with which the leading one digit of the CR can be detected. A pointer to this digit, carried along with the computation, would immediately identify this digit. The pointer logic requires additional hardware and its benefit decreases if lengthy scalar products are to be computed.

For short scalar products or single floating-point operations leading zero anticipation (LZA) would be more useful. The final result of a scalar product computation is supposed to lie in the exponent range between $e1$ and $e2$ of the CR. Otherwise the problem has to be scaled. So hardware support for LZA is only needed for this part of the CR. A comparison of the exponents of the summands identifies the CR word for which the LZA should be activated. LZA involves fast computation of a provisional sum which differs from the correct sum by at most one leading zero. With this information the leading zeros and the shift width for the two CR words in question can be detected easily and quickly [577].

8.6.2 How Much Local Memory Should be Provided on an SPU?

There are applications which make it desirable to provide more than one CR on the SPU. If, for instance, the components of the two vectors $a = (a_i)$ and $b = (b_i)$ are complex floating-point numbers, the scalar product $a \cdot b$ is also a complex floating-point number. It is obtained by accumulating the real and imaginary parts of the product of two complex floating-point numbers. The formula for the product of two complex floating-point numbers

$$\begin{aligned} \big(x &= x_1 + ix_2,\; y = y_1 + iy_2 \\ \Rightarrow x \cdot y &= (x_1 \cdot y_1 - x_2 \cdot y_2) \;+\; i\,(x_1 \cdot y_2 + x_2 \cdot y_1)\big) \end{aligned}$$

shows that the real and imaginary parts of a_i and b_i are both needed for the computation of both the real part of the product $a_i \cdot b_i$ as well as the imaginary part.

Access to user memory is usually slower than access to register memory. To obtain high computing speed it is desirable, therefore, to bring the real and imaginary parts of the vector components in only once and to compute the real and imaginary parts of the products simultaneously in two CRs on the SPU instead of reading the data twice and performing the two accumulations sequentially. Interestingly, the old calculators shown in Figures 8.1(c) and (d) on page 249 had two long registers.

Very similar considerations show that a high speed computation of the scalar product of two vectors with interval components makes two CRs desirable as well.

We briefly sketch here a scalar product unit for two vectors with interval components. If $\boldsymbol{A} = (A_\nu)$ and $\boldsymbol{B} = (B_\nu)$ with $A_\nu = [a_{\nu 1}, a_{\nu 2}]$ and $B_\nu = [b_{\nu 1}, b_{\nu 2}] \in IS$ are two such vectors this requires evaluation of the formulas

$$\boldsymbol{A} \diamondsuit \boldsymbol{B} = ([a_{\nu 1}, a_{\nu 2}]) \diamondsuit ([b_{\nu 1}, b_{\nu 2}]) \tag{8.6.1}$$

$$= \left[\triangledown \sum_{\nu=1}^{n} \min_{i,j=1,2} (a_{\nu i} b_{\nu j}), \triangle \sum_{\nu=1}^{n} \max_{i,j=1,2} (a_{\nu i} b_{\nu j}) \right]. \tag{8.6.2}$$

Here the products of the bounds of the vector components $a_{\nu i} b_{\nu j}$ are to be computed to the full double length. Then the minima and maxima have to be selected. These selections can be done by distinguishing the nine cases shown in Table 4.1. This can be hardware supported as shown in Figure 7.2 or 7.5. The selected products are then accumulated in $\mathbb{R}$ as an exact scalar product. Clearly, forwarding the products then requires wider busses. The two sums are computed in two complete registers by one of the techniques shown in this chapter. Finally the sum of products is rounded only once by $\triangledown$ (resp. $\triangle$) from $\mathbb{R}$ into S. See Figure 8.12. After multiplication in the figure the path through minimum and maximum selection respectively is only activated, if both intervals $[a_{\nu 1}, a_{\nu 2}]$ and $[b_{\nu 1}, b_{\nu 2}]$ contain zero.

The unit shown in Figure 8.12 can also be used to compute the scalar product of two vectors the components of which are complex floating-point numbers.

For vectors with complex interval components even four CRs would be useful.

There might be other reasons to provide local memory space for more than one CR on the SPU. A program with higher priority may interrupt the computation of a scalar product and require a CR. The easiest way to solve this problem is to open a new CR for the program with higher priority. Of course, this can happen several times which raises the question how much local memory for how many CRs should be provided on an SPU. Three might be a good number to solve this problem. If a further interrupt requires another CR, the CR with the lowest priority could be mapped into the main memory by some kind of stack mechanism. This technique would not limit the number of interrupts that may occur during a scalar product computation. These problems and questions must be addressed as part of the operating system's responsibility.

For a time sharing environment memory space for more than one CR on the SPU may also be useful.

However the contents of the last two paragraphs are of a somewhat hypothetical nature. The author is of the opinion that the scalar product is a fundamental and basic operation which should not be, and never needs to be, interrupted.

8.7 The Data Format Complete and Complete Arithmetic

We have seen in the previous sections that scalar or dot products can be computed exactly and without any exceptions in a register of $L = k + 2e2 + 2l + 2|e1|$ digits of base b. In numerical analysis the scalar product is ubiquitous. Most of the advanced

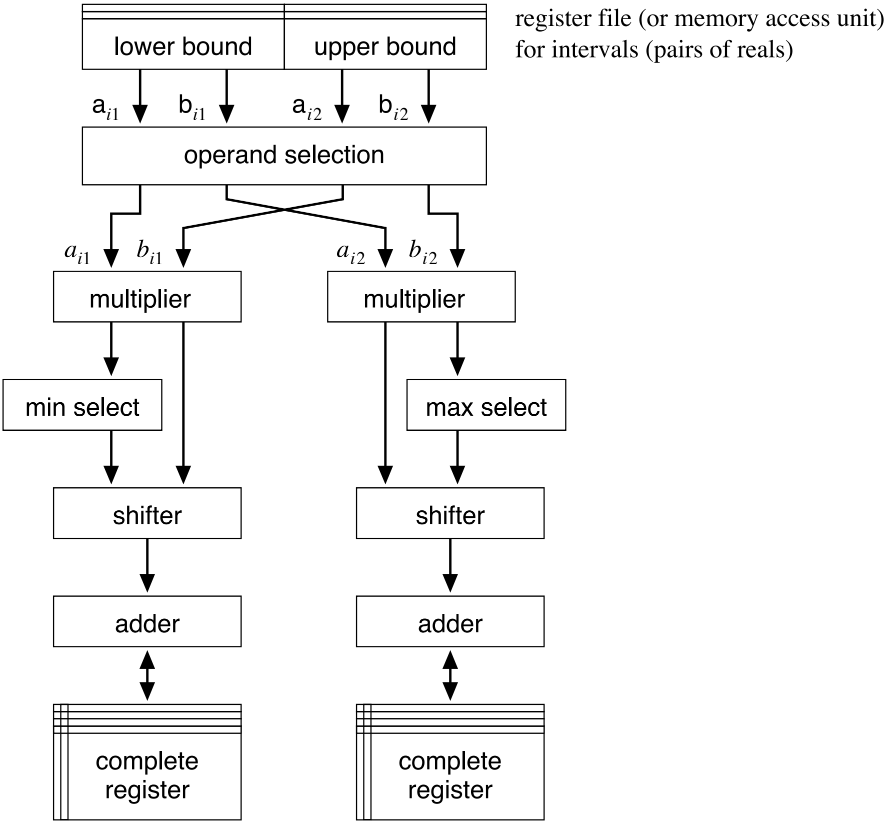


Figure 8.12. SPU for vectors with interval components.

applications discussed in Chapter 9 of this book are based on applications of the exact scalar product. With it advanced computer arithmetic surpasses elementary floating-point arithmetic as well as basic computer arithmetic. The basic components for the computation of the exact scalar product, therefore, should be given into the hands of the users.

We do this by a new data format which is called *complete* and a limited set of arithmetic operations defined for it. *Complete arithmetic* is performed in what is called a *complete register*, CR for short. Complete arithmetic suffices to compute all scalar products of floating-point vectors exactly. The result of complete arithmetic is always exact, it is complete, and no intermediate roundings or truncations are performed. Not a single bit is lost.

A datum of the type complete is a fixed-point word of the size of the complete register. It consists of $L = k + 2e2 + 2l + 2|e1|$ digits of base b. See Figure 8.13. It covers the full range of the floating-point system $R = R(b, 2l, 2e2, 2e1)$ and a few additional digits. It suffices to allow exact accumulation (continued summation) of products of floating-point numbers. In the complete register the b-ary point is sitting immediately to the right of the $(k + 2e2)$th digit of base b.

k	$2e2$	$2l$	$2\|e1\|$

Figure 8.13. Complete Register CR.

Complete arithmetic can be provided for every specific floating-point format.

The following two sections give programming instructions for complete arithmetic for low and high level programming languages respectively.

8.7.1 Low Level Instructions for Complete Arithmetic

For complete arithmetic the following ten low level instructions are recommended. They are the most natural and are necessary for application of complete arithmetic. These low level capabilities support the high level instructions developed in the next section, and are based on experience with these in the XSC-languages since 1980. Very similar instructions were provided by the processors developed in [585], and [61]. Practically identical instructions were used in [650] to support ACRITH and ACRITH-XSC [649, 651, 653]. These IBM program products were developed at the author's institute in collaboration with IBM.

The ten low level instructions for complete arithmetic are:

1. clear the CR,

2. add a product to the CR,

3. add a floating-point number to the CR,

4. subtract a product from the CR,
5. subtract a floating-point number from the CR,
6. read the CR and round its contents to the destination format,
7. store the contents of the CR in memory,
8. load the CR from memory,
9. add stored CR to CR,
10. subtract stored CR from CR.

The clear instruction can be performed by setting all *all bits 0* flags to 0. Also the *all bits 1* flags should be set to 0. The load and store instructions are performed by using the load/store instructions of the processor. For the add, subtract and round instructions the following denotations are used here. The prefix *sp* identifies SPU instructions. *ln* denotes the floating-point format that is used and will be *db* for IEEE double. In all these instructions, the CR is an implicit source and destination operand. The numbering of the instructions above is repeated in the following.

2. spadd *ln* src1, src2
   multiply the numbers in the given registers and add the product to the CR.
3. spadd *ln* src
   add the number in the given register to the CR.
4. spsub *ln* src1, src2
   multiply the numbers in the given registers and subtract the product from the CR.
5. spsub *ln* src
   subtract the number in the given register from the CR.
6. spstore *ln.rd* dest
   get CR contents and put the rounded value into the destination register.
   Here *rd* controls the rounding mode that is used when the CR contents are stored in a floating-point register. It is one of the following:

rn	rounding to nearest,
rz	rounding toward zero,
rp	rounding upwards, i.e., toward positive ∞,
rn	rounding downwards, i.e., toward negative ∞.

7. spstore dest
   get CR contents and put them into the destination memory operand.
8. spload src
   load the *complete* number from the given memory operand into the CR.

9. spadd src
   the *complete* number at the location src is added to the contents of the CR in the processor.

10. spsub src
    the *complete* number at the location src is subtracted from the contents of the CR in the processor.

If multiple CRs are provided on the SPU additional instructions for exchanging CR contents may be useful.

8.7.2 Complete Arithmetic in High Level Programming Languages

Advances in computer technology allow the quality and high accuracy of the basic floating-point operations of addition, subtraction, multiplication and division to be extended to the arithmetic operations in the linear spaces and their interval extensions which are most commonly used in computation. The scalar product serves to provide these operations. It can be produced by an instruction *multiply and accumulate*. The products are accumulated in a complete register CR which has enough digit positions to contain the exact sum in its entirety. Only a single rounding error of at most one unit in the last place is introduced when the completed scalar product (often also called dot product) is returned to one of the floating-point registers.

By operator overloading in modern programming languages matrix and vector operations can be provided with highest accuracy and in a simple notation, if the exact scalar product is available. However, many scalar products that occur in a computation do not appear as vector or matrix operations in the program. A vectorizing compiler is certainly a good tool for detecting such masked scalar products in a program. Since the hardware-supported exact scalar product is very fast this would increase both the accuracy and the speed of the computation.

In the computer, the scalar product is produced by the elementary computer instructions shown in the last section. Programming and the detection of scalar products in a program can be simplified a great deal if some of these computer instructions are put into the hands of the user and incorporated into high level programming languages. This has been done with great success since 1980 in the so-called XSC-languages (eXtended Scientific Computation) [79, 257, 288, 289, 290, 291, 333, 355, 356, 647, 648, 649, 653] that have been developed at the author's institute. All these languages provide an exact scalar product implemented in software based on integer arithmetic. If a computer is equipped with the hardware unit XPA 3233 (see Section 8.5.2) the hardware unit is called instead. A large number of problem solving routines with automatic result validation has been implemented in the XSC-languages for practically all standard problems of numerical analysis [206, 207, 332, 368, 651, 653]. These routines have been very successfully applied in the sciences.

We mention a few of these constructs and demonstrate their usefulness. Central to this is the idea of allowing variables of the size of the CR to be defined in a user's program. For this purpose a new data type called `complete` is introduced. A variable of the type `complete` is a fixed-point variable with $L = k + 2e2 + 2l + 2|e1|$ digits of base b. See Figure 8.13. As has been shown earlier, every finite sum of floating-point products $\sum_{i=1}^{n} a_i \cdot b_i$ can be represented as a variable of type `complete`. Moreover, every such sum can be computed in a complete register (CR) without loss of information. Along with the type `complete` the following constructs serve as primitives for developing expressions in a program which can easily be evaluated with the low level instructions introduced in the last section:

`complete`	new data type
`:=`	assignment from `complete` to `complete` or to `real` with rounding to nearest or to `interval` with roundings downwards and upwards depending on the type on the left hand side of the `:=` operator.

For variables of type `complete` so-called *complete expressions* are permitted. As every expression a complete expression is composed of operands and operations. Only three kinds of operands are permitted:

(a) a constant or variable of type `real`,

(b) an exact product of two such objects, and

(c) another datum of type `complete`.

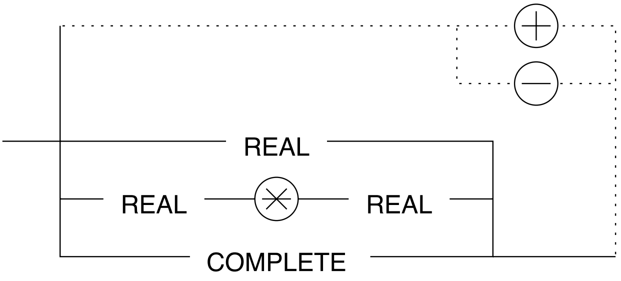


Figure 8.14. Syntax diagram for COMPLETE EXPRESSION.

Such operands can be added or subtracted in an arbitrary order. All operations (multiplication, addition, and subtraction) are to be exact. The result is a datum of type `complete`. It is always exact. No truncation or rounding is performed during execution of a complete expression. The result of *complete arithmetic* reflects the complete information as given by the input data and the operations in the expression.

Figure 8.14 shows a syntax diagram for COMPLETE EXPRESSION. In the diagram solid lines are to be traversed from left to right and from top to bottom. Dotted lines are to be traversed oppositely.

Compete arithmetic is a necessary complement to floating-point arithmetic. Although it looks very simple it is a very powerful tool. A central task of numerical analysis is the solution of systems of linear equations. The so-called verification step for a system of linear equations is solely performed by complete arithmetic. It computes close bounds for the solution. *Complete arithmetic* also is a fundamental tool for many other applications discussed in Chapter 9.

We now illustrate its use by a few simple algorithms. For instance, let `x` be a variable of type `complete` and `y` and `z` variables of type `real`. Then in the assignment

```
x := x + y * z
```

the double length product of `y` and `z` is added to the variable `x` of type `complete` and its new value is assigned to `x`.

The scalar product of two vectors `a=(a[i])` and `b=(b[i])` is now easily implemented with a variable `x` of type `complete` as follows:

```
x := 0;
for i := 1 to n do x := x + a[i] * b[i];
y := x;
```

The last statement `y := x` rounds the value of the variable `x` of type `complete` into the variable `y` of type `real` by applying the standard rounding of the computer. Variable `y` then has the value of the scalar product $a \boxdot b$ which is within a single rounding error of the exact scalar product $a \cdot b$.

For example, the method of defect correction or iterative refinement requires highly accurate computation of expressions of the form

```
a * b − c * d
```

with vectors `a`, `b`, `c`, and `d`. Employing a variable `x` of type `complete`, this expression can now be programmed as follows:

```
x := 0;
for i := 1 to n do x := x + a[i] * b[i];
for i := 1 to n do x := x − c[i] * d[i];
y := x;
```

or by

```
x := 0;
for i := 1 to n do x := x + a[i] * b[i] − c[i] * d[i];
y := x;
```

The result `y`, involving $2n$ exact multiplications and $2n - 1$ exact additions, is produced with a single rounding operation.

In the last two examples `y` could have been defined to be of type `interval`. Then the last statement `y := x` would produce an interval with a lower bound which is obtained by rounding the `complete` value of `x` downwards and an upper bound by rounding it upwards. Thus, the bounds of `y` will be either the same or two adjacent floating-point numbers.

In the XSC-languages the functionality of the `complete` type and expression is available also for complex data as well as for interval and complex interval data. Corresponding types would be called `ccomplete`, `icomplete`, and `cicomplete` respectively. High speed dot products for these types would require two complete registers in the first two cases and four in the latter.

Complete arithmetic and expressions can also be defined for higher dimensional data types like vectors and matrices of the four basic types mentioned in the last paragraph. In the corresponding syntax diagrams the product would be the vector matrix product and the matrix product respectively. On scalar processors these expressions would be executed componentwise using the basic complete arithmetic. Since the execution for the different components is independent of each other, parallel execution of several components is possible if the processor allows this.

The data type `complete` is the appropriate generalization of the long result registers that have been used in the old calculators shown in the Figures 8.1(c) and (d).

8.8 Top Speed Scalar Product Units

A top-performance computer is able to read two data x and y to produce the product $x \cdot y$ into the arithmetic logical unit and/or the SPU simultaneously in one cycle. Supercomputers and vector processors are typical of this kind of computer. Usually the floating-point word has 64 bits and the data bus is 128 or even more bits wide. However, digital signal processors with a word size of 32 bits can also belong in this class if two 32 bit words are read into the ALU and/or SPU in one cycle. For such computers both the solutions sketched in Sections 8.4.1 and 8.4.2 make sense and will be considered in the following. The higher the speed of the system the more circuitry has to be used for the accumulation of the products. The more complex and expensive solution seems to be best suited to reveal the basic ideas. So we begin with the solution which uses a long fixed-point adder and a 64-bit data format.

8.8.1 SPU with Long Adder for 64 Bit Data Word

In [365] the basic ideas for a general data format have been developed. However, to be very specific we discuss here a circuit for the double precision format of the IEEE-arithmetic standard 754. The word size is 64 bits. The mantissa has 53 bits and the exponent 11 bits. The exponent covers a range from -1022 to $+1023$. The CR has

4288 bits. We assume again that the scalar product computation can be divided into the independent steps:

(a) read a_i and b_i,

(b) compute the product $a_i \cdot b_i$,

(c) add the product to the CR.

Now by assumption the SPU can read the two factors a_i and b_i entirely and simultaneously. We call the time that is needed for this a *cycle*. Then, in a balanced design, steps b) and c) should take about the same time. Using well-known fast multiplication techniques like Booth-Recoding and Wallace-tree this certainly is possible for step b). Here, the two 53 bit mantissas are multiplied. The product has 106 bits. The main difficulty seems to appear in step c). There, we have to add a summand of 106 bits to the CR in every *cycle*.

Following the idea that has been discussed in Section 8.4.1 the addition is performed by a long adder and a long shift, both of $L = 4288$ bits. An adder and a shift of this size are necessarily slow, certainly too slow to process one summand of 106 bits in a single cycle. Therefore, measures have to be taken to speed up the addition as well as the shift. As a first step we subdivide the long adder into shorter segments. The width of these segments is chosen in such a way that a parallel addition can be performed in a single cycle. We assume here that the segments consist of 64 bits.[3] A 64 bit adder certainly is faster than a 4288 bit adder. Now each of the 64 bit adders may produce a carry. We write these carries into carry registers between adjacent adders. See Figure 8.15.

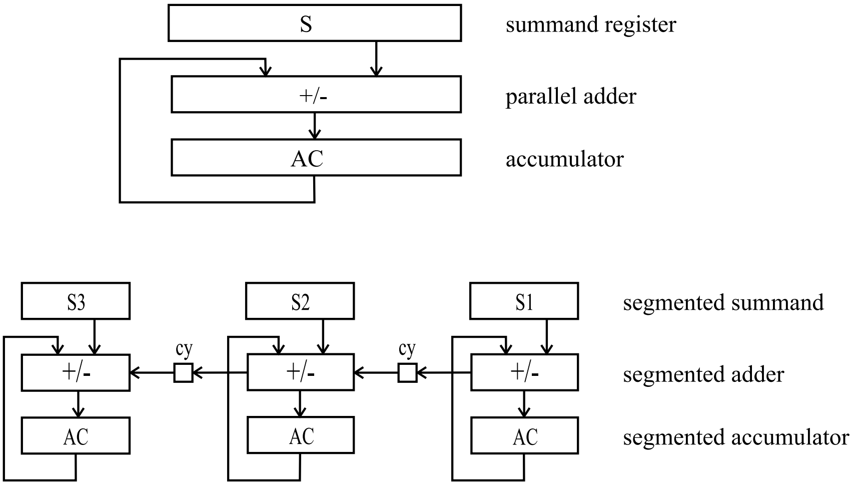


Figure 8.15. Parallel and segmented parallel adder.

[3] Other segment sizes are possible; see [284, 285].

If a single addition has to be performed these carries have to be propagated straight away. In a scalar product computation, however, this is not necessary. We assume many summands have to be added up. In the next machine cycle the carries are added to the next more significant adder, possibly together with another summand. Only at the very end of the accumulation, when no more summands are coming, carries may have to be eliminated. However, every summand is relatively short. It consists of 106 bits only. So during the addition of a summand, carries are only produced in a small part of the 4288 bit adder. The carry elimination, on the other hand, takes place during each step of the addition wherever a carry is left. So in an average case there will only be very few carries left at the end of the accumulation and a few additional cycles will suffice to absorb the remaining carries. Thus, segmenting the adder enables it to keep up with steps a) and b) and to read and process a summand in each cycle.

The long shift of the 106 bit summand is slow also. It is speeded up by a matrix shaped arrangement of the adders. Only a few, let us assume here four, of the partial adders are placed in a row. We begin with the four least significant adders. The four next more significant adders are placed directly beneath of them and so on. The most significant adders form the last row. The rows are connected as shown in Figure 8.16.

In our example, where we have 67 adders of 64 bits, 17 rows suffice to arrange the entire summing matrix. Now the long shift is performed as follows: The summand of 106 bits carries an exponent. In a fast shifter of 106 to 256 bits the summand is shifted into a position where its most significant digit is placed directly above the position in the long adder which carries the same exponent identification E. The remaining digits of the summand are placed immediately to its right. Now the summing matrix reads this summand into the S-registers (summand registers) of every row. The addition is done in the row where the exponent identification coincides with that of the summand.

It may happen that the most significant digit of the summand has to be shifted so far to the right that the remaining digits would hang over at the right end of the shifter. These digits then are reinserted at the left end of the shifter by a ring shift. If now the more significant part of the summand is added in row r, its less significant part will be added in row $r - 1$.

By this matrix shaped arrangement of the adders, the unit can perform both a shift and an addition in a single cycle. The long shift is reduced to a short shift of 106 to 256 bits, which is fast. The remaining shift happens automatically by the row selection for the addition in the summing matrix.

Every summand carries an exponent which in our example consists of 12 bits. The lower part of the exponent, i.e., the 8 least significant digits, determine the shift width and with it the selection of the columns in the summing matrix. The row selection is specified by the 4 most significant bits of the exponent. This corresponds roughly to the selection of the adding position in two steps by the process of Figure 8.3. The shift width and the row selection for the addition of a product $a_i \cdot b_i$ to the CR are known as soon as the exponent of the product has been computed. Since the exponents of a_i

and b_i consist of 11 bits only, the result of their addition is available very quickly. So while the mantissas are being multiplied the shifter can be switched and the addresses of the CR words for the accumulation of the product $a_i \cdot b_i$ can be selected.

The summing matrix just described must be able to process positive and negative summands. In the latter case borrows may occur. They also have to be processed, possibly over several cycles. The 106 bit summand maps onto at most three consecutive words of the CR. The summand is added by these three partial adders. Each of these adders can produce a carry. The carry of the leftmost of these partial adders can with high probability be absorbed, if the addition always is executed over four adders and the fourth adder then is the next more significant one. This can reduce the number of carries that have to be resolved during future steps of the accumulation and in particular at the end.

In each step of the accumulation an addition only has to be activated in the selected row of adders and in those adders where a non zero carry is waiting to be absorbed. This adder selection can reduce the power consumption for the accumulation step significantly.

The carry resolution method that has been discussed so far is quite natural. It is simple and does not require special hardware support. If long scalar products are being computed it works very well. At the end of the accumulation, only if no more summands are coming, a few additional cycles may be required to absorb the remaining carries. Then a rounding can be done. However, the additional cycles for the carry resolution at the end of the accumulation, although few in general, depend on the data and are unpredictable. For short scalar products the time needed for these additional cycles may be disproportionately high and indeed exceed the addition time.

With the fast carry resolution mechanism that has been discussed in Section 8.4.4 these difficulties can be overcome. At the cost of some additional hardware all carries can be absorbed immediately at each step of the accumulation. The method is shown in Figure 8.16 also. Two flag registers for the *all bits 0* and the *all bits 1* flags are shown at the left end of each partial accumulator word in the figure. The 106 bit products are added by three consecutive partial adders. Each of these adders can produce a carry. The carries between adjacent adders can be avoided, if all partial adders are built as carry-select-adders. This increases the hardware costs only moderately. The carry registers between adjacent adders are then no longer necessary.[4] The flags indicate which one of the more significant CR words will absorb the leftmost carry. During the addition of a product only these 4 CR words are changed and only these 4 adders need to be activated. The addresses of these 4 words are available as soon as the exponent of the summand $a_i \cdot b_i$ has been computed. During the addition step the carry word can be incremented (decremented) while the product is being added. If the addition produces a carry the incremented word will be written back into this

[4] This is the case in Figure 8.17 where a similar situation is discussed. There all adders are supposed to be carry-select-adders.

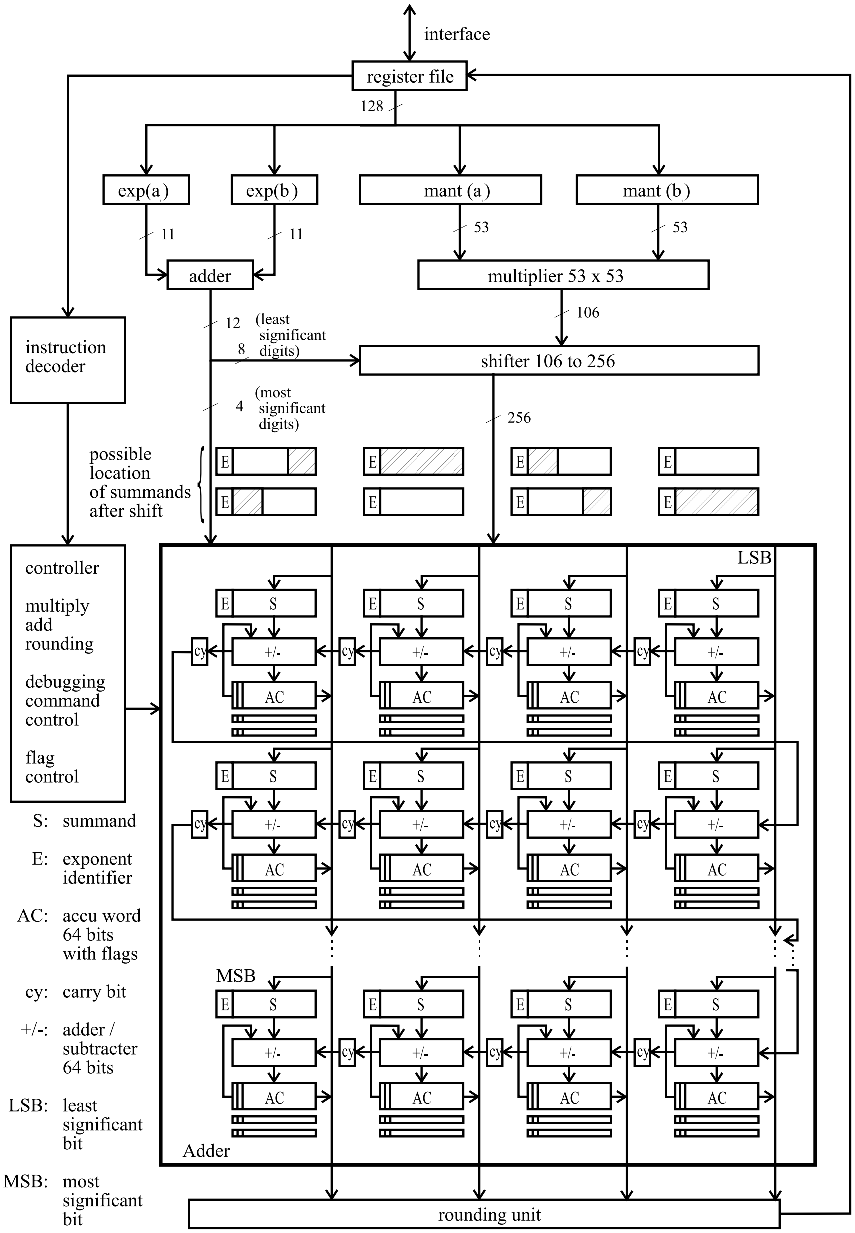


Figure 8.16. Block diagram of an SPU with long adder for a 64 bit data word and 128 bit data bus.

CR. If the addition does not produce a carry, the CR word remains unchanged. Since we have assumed that all partial adders are built as carry-select-adders this final carry resolution scheme requires no additional hardware. Simultaneously with the incrementing/decrementing of the carry word a second set of flags is set up for the case that a carry is generated. In this case the second set of flags is copied into the former flag word.

The accumulators that belong to partial adders in Figure 8.16 are denoted by AC. Beneath them a small memory is indicated in the figure. It can be used to save the CR contents very quickly should a program with higher priority interrupt the computation of a scalar product and require the unit for itself. However, the author is of the opinion that the scalar product is a fundamental and basic arithmetic operation which should never be interrupted.

In Section 8.6.2 we have discussed applications like complex arithmetic or interval arithmetic which make it desirable to provide more than one CR on the SPU. The local memory on the SPU shown in Figure 8.16 can be used for fast execution of scalar products in complex and interval arithmetic.

In Figure 8.16 the registers for the summands carry an exponent identification denoted by E. This is very useful for the final rounding. The usefulness of the flags for the final rounding has already been discussed. They also serve for fast clearing of the accumulator.

The SPU which has been discussed in this section seems to be costly. However, it consists of a large number of identical parts and it is very regular. This allows a highly compact design. Furthermore the entire unit is simple. No particular exception handling techniques are to be dealt with by the hardware. The result is exact. Vector computers are the most expensive. A compact and simple solution, though expensive, is justified for these systems.

8.8.2 SPU with Long Adder for 32 Bit Data Word

In this section we consider a computer which uses a 32 bit floating-point word and which is able to read two such words into the ALU and/or SPU simultaneously in one portion. Digital signal processors are like this. Real time computing requires very high computing speed and high accuracy in the result. As in the last section we call the time that is needed to read the two 32 bit floating-point words a cycle.

We first develop circuitry for the SPU using a long adder and a long shift. To be very specific we assume that the data are given as single precision floating-point numbers conforming to the IEEE-arithmetic standard 754. There the mantissa consists of 24 bits and the exponent has 8 bits. The exponent covers a range from -126 to $+127$ (in binary). As discussed in Remark 8.3(a) of Section 8.4.3, 640 bits is a reasonable choice for the CR. It can be represented by ten words of 64 bits.

Again the scalar product is computed by independent steps like

(a) read a_i and b_i,

(b) compute the product $a_i \cdot b_i$,

(c) add the product to the CR.

Each of the mantissas of a_i and b_i has 24 bits. Their product has 48 bits. It can be computed very fast by a 24 × 24 bit multiplier using standard techniques like Booth-Recoding and Wallace-tree. The addition of the two 8 bit exponents of a_i and b_i delivers the exponent of the product consisting of 9 bits.

The CR consists of 10 words of 64 bits. The 48 bit mantissa of the product maps onto at most two of these words. The product is added by the corresponding two consecutive partial adders. Each of these two adders can produce a carry. The carry between the two adjacent adders can immediately be absorbed if all partial adders are built as carry-select-adders again. The carry of the more significant of the two adders will be absorbed by one of the more significant 64 bit words of the CR. The flag mechanism (see Section 8.4.4) indicates which one of the CR words will absorb a possible carry. So during an addition of a summand the contents of at most 3 CR words are changed and only these three partial adders need to be activated. The addresses of these words are available as soon as the exponent of the summand $a_i \cdot b_i$ has been computed. The carry word can be incremented (decremented) while the product is being added in. If the addition produces a carry the incremented word will be written back into the CR. If the addition does not produce a carry, this CR word remains unchanged. Since all partial adders are built as carry select adders no additional hardware is needed for the carry resolution. While the carry word is being incremented/decremented, a second set of flags is set up in case a carry is generated. In this case the second set of flags is copied into the former flag word.

Details of the circuitry just discussed are summarized in Figure 8.17. The figure is highly similar to Figure 8.16 of the previous section. To avoid the long shift, the long adder is designed as a summing matrix consisting of two adders of 64 bits in each row. For simplicity in the figure only three rows of the five needed to represent the full CR are shown.

In a fast shifter of 48 to 128 bits the 48 bit product is shifted into a position where its most significant digit is placed directly above the position in the long adder which carries the same exponent identification E. The remaining digits of the summand are placed immediately to its right. If they hang over at the right end of the shifter, they are reinserted at the left end by a ring shift. Above the summing matrix in Figure 8.17 two possible positions of summands after the shift are indicated.

The summing matrix now reads the summand into its S-registers. The addition is done by those adders where the exponent identification coincides with that of the summand. The exponent of the summand has nine bits. The lower part, i.e., the seven least significant bits, determines the shift width. The selection of the two adders which perform the addition is determined by the two most significant bits of the exponent.

In Figure 8.17 some local memory is indicated for each part of the CR. It can be used to save the CR contents very quickly should a program with higher priority

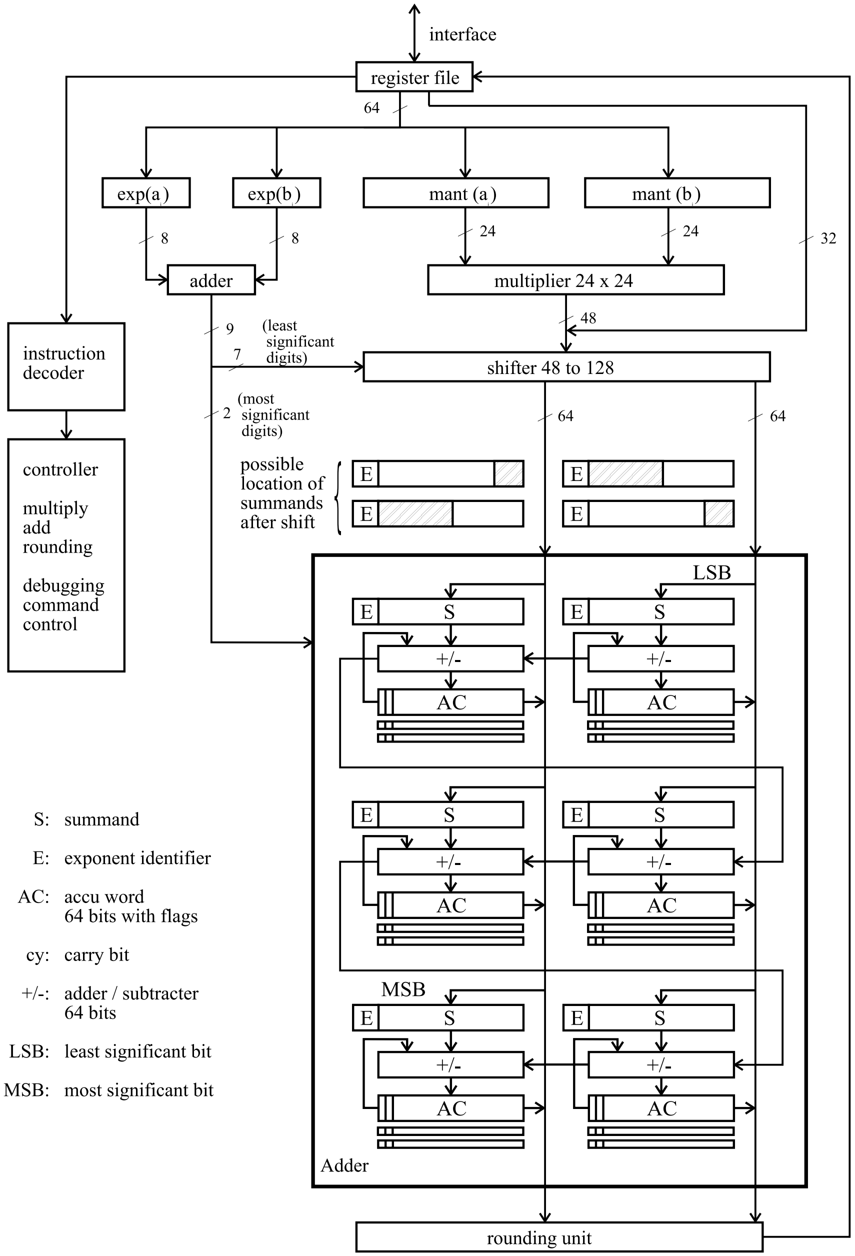


Figure 8.17. Block diagram of an SPU with long adder for a 32 bit data word and 64 bit data bus.

interrupt the computation of a scalar product and require the unit for itself. The local memory on the SPU also can be used for fast execution of scalar products in complex and interval arithmetic.

Compared to Figure 8.16, Figure 8.17 shows an additional 32 bit data path directly from the input register to the fast shifter. This data path is used to allow very fast execution of the operation *multiply and add fused*, $rnd(a \cdot b + c)$, which is provided by some conventional floating-point processors. While the product $a \cdot b$ is being computed by the multiplier, the summand c is added to the CR.

The SPU which has been discussed in this section seems to be costly at first glance. While a single floating-point addition can be done conveniently with one 64 bit adder, here 640 full adders (ten 64-bit adders) have been used in carry-select-adder mode. However, the advantages of this design are tremendous. While a conventional floating-point addition can produce a completely wrong result with only two or three additions, the new unit never delivers a wrong answer, even if millions of floating-point numbers or single products of such numbers are added. An error analysis is never necessary for these operations. The result is exact. The unit consists of a large number of identical parts and it is very regular. This allows a very compact design. No particular hardware has to be included to deal with rare exceptions. Although an increase in adder equipment by a factor of 10, compared with a conventional floating-point adder, might seem to be high, the number of full adders used for the circuitry is not extraordinary. We stress the fact that for a Wallace tree in the case of a standard 53 × 53 bit multiplier about the same number of full adders is used. For fast conventional computers this has been the state of the art multiplication for many years and nobody complains about high cost.

8.8.3 A FPGA Coprocessor for the Exact Scalar Product

A hardware unit which realizes the SPU as discussed in Section 8.8.2 has been built in Field Programmable Gate Array (FPGA) technology at the author's institute in 2001/2002 [73]. The XILINX XCV800 processor was used. The unit has been designed in such a way that it can also be used to compute the exact scalar product for double precision floating-point numbers within the single exponent range, i.e., if no under- and no overflow occurs.

The design of an SPU in FPGA technology, of course, is simpler than that of the unit discussed in Section 8.5.2 where a gate array technology or full custom design was used. No expensive hardware equipment and reproduction facilities are needed in the FPGA technology. Everything can be done in a mathematics institute. However, the board that finally carries the FPGA chip is more complicated than the one shown in Figure 8.8. Several additional support chips are needed to reach the full functionality. The main disadvantage of this solution however, was the speed. Fixed-point accumulation of the exact scalar product is supposed to be about four times as fast as a computation of the scalar product in floating-point arithmetic. This theoretical gain in speed could not keep up with the loss of speed caused by the much lower clock rate of the FPGA chip in comparison with the clock rate of the most advanced processors. The situation may change in the future if the clock rate of an FPGA processor comes closer to that of its main processor. Nevertheless a coprocessor for the exact scalar product can only be a temporary solution. Eventually the exact scalar product must be incorporated into the ALU of the processor itself.

8.8.4 SPU with Short Adder and Complete Register

In the circuits discussed in Sections 8.8.1 and 8.8.2 adder circuitry was provided for the full width of the CR. The long adder was segmented into partial adders of 64 bits. In Section 8.8.1 67 and in Section 8.8.2 10 such units were used. During the addition of a summand, however, in Section 8.8.1 only four, and in Section 8.8.2 only three, of these units are activated. This raises the question of whether adder circuitry is really needed for the full width of the CR and whether the accumulation can be done with only three or four adders in accordance with the solution of Section 8.4.2. There the CR is kept as local memory on the arithmetic unit.

In this section we develop such a solution for the 64-bit data format. A solution in principle using a short adder and local memory on the arithmetic unit was discussed in Section 8.5.3. There the data a_i and b_i to perform a product $a_i \cdot b_i$ are read into the SPU successively in two portions of 64 bits. This leaves four machine *cycles* to perform the accumulation in the pipeline.

Now we assume that the two data a_i and b_i for a product $a_i \cdot b_i$ are read into the SPU simultaneously in one portion of 128 bits. Again we call the time that is needed for this a cycle. In accordance with the solution shown in Figure 8.16 and Section 8.8.1 we assume that the multiplication and the shift can also be done in one such read cycle. In a balanced pipeline, then, the circuit for the accumulation must be able to read and process one summand in each (read) cycle also. The circuit in Figure 8.18 displays a solution. Closely following the summing matrix in Figure 8.16 we assume there that the local memory is organized in 17 rows of four 64 bit words.

In each cycle the multiplier supplies a product (summand) to be added in the accumulation unit. Every such summand carries an exponent which in our example is 12 bits long. The eight lower (least significant) bits of the exponent determine the shift width. The row selection of the CR is given by the four most significant bits of the exponent. This roughly corresponds to the selection of the adding position in two steps by the process described in the context of Figure 8.3. The shift width and the row selection for the addition of the product to the local memory are known as soon as the exponent of the product has been computed. Since the exponents of a_i and b_i consist of 11 bits only, the result of their addition is available very quickly. So while the mantissas are still being multiplied the shifter can be switched and the addresses for the CR words for the accumulation of the product $a_i \cdot b_i$ can be selected.

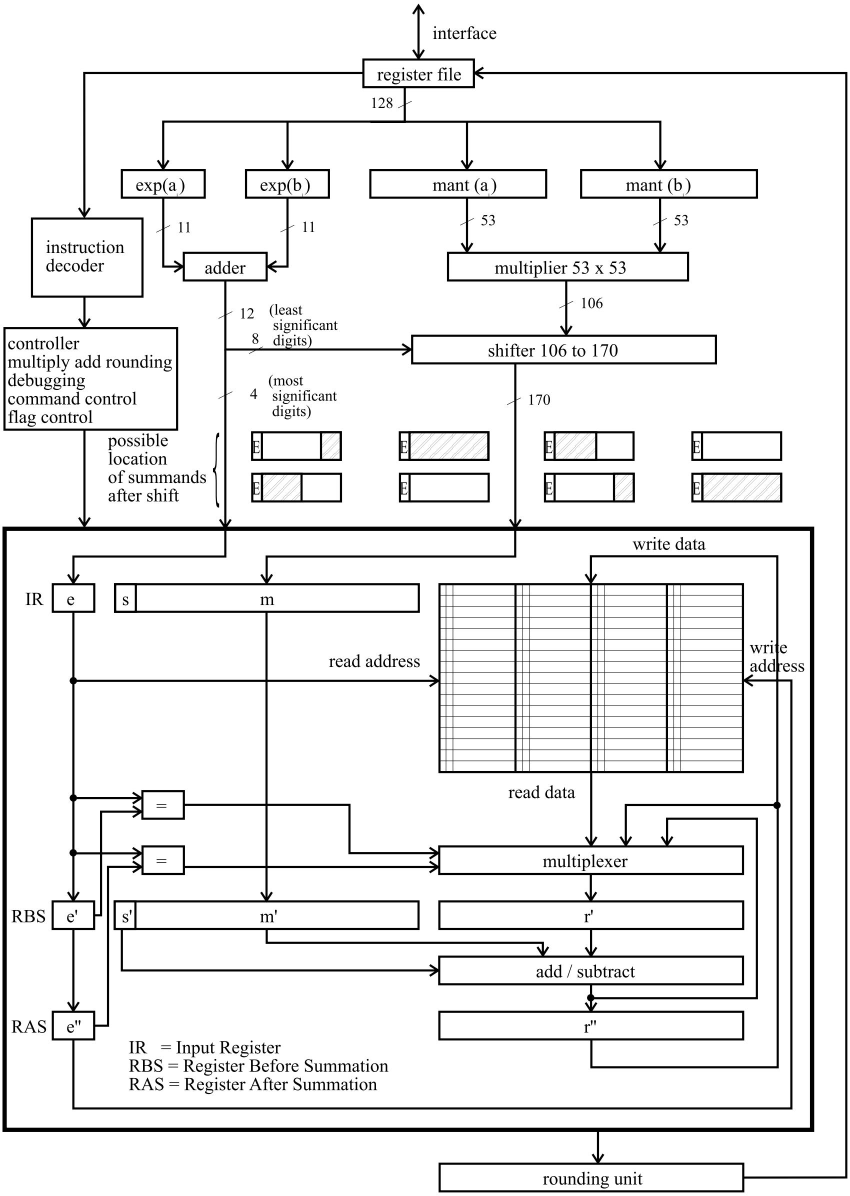


Figure 8.18. Block diagram of an SPU with short adder and local store for a 64 bit data word and 128 bit data bus.

After being shifted the summand reaches the accumulation unit. It is read into the input register IR of this unit. The shifted summand now consists of an exponent e, a sign s, and a mantissa m. The mantissa maps onto three consecutive words of the CR, while the exponent is reduced to the four most significant bits of the original exponent of the product.

Now the addition of the summand is performed in the accumulation unit by the following three steps:

(i) The local memory is addressed by the exponent e. The contents of the addressed part of the CR including the word which resolves the carry are transferred to the register before summation (RBS). This transfer moves four words of 64 bits. The summand is also transferred from IR to the corresponding section of RBS. In Figure 8.18 this part of the RBS is denoted by e', s' and m' respectively.

(ii) In the next cycle the addition or subtraction is executed in the add/subtract unit according to the sign. The result is transferred to the register after summation (RAS). The adder/subtracter consists of four parallel adders of 64 bits which are working in carry select mode. The summand maps onto three of these adders. Each of these three adders can produce a carry. The carries between adjacent adders are absorbed by the carry select addition. The fourth word is the carry word. It is selected by the flag mechanism. During the addition step a one is added to or subtracted from this word. If the addition produces a carry the incremented/decremented word will be selected. If the addition does not produce a carry this word remains unchanged. While the carry word is being incremented/decremented, a second set of flags is set up which is copied into the flag word if a carry is generated. In Figure 8.18 two possible locations of the summand after the shift are indicated. The carry word is always the most significant word. Incrementing or decrementing this word never produces a carry. Thus the adder/subtracter in Figure 8.18 can simply be built as a parallel carry-select-adder.

(iii) In the next cycle the computed sum is written back into the same four memory cells of the CR to which the addition has been executed. Thus only one address decoding is necessary for the read and write step. A different bus called *write data* in Figure 8.18 is used for this purpose.

Figure 8.20 shows a more detailed block diagram for an SPU with short adder and local store.

In summary the addition consists of the typical three steps: 1. read the summand, 2. perform the addition, and 3. write the sum back into the (local) memory. Since a summand is delivered from the multiplier in each cycle, all three phases must be active simultaneously, i.e., the addition itself must be performed in a pipeline. This means that it must be possible to read from the memory and to write into the memory in each cycle simultaneously. So two different data paths have to be provided. This, however, is usual for register memory.

The pipeline for the addition has three steps. Pipeline conflicts are quite possible. A pipeline conflict occurs if an incoming summand needs to be added to a partner from the CR which is still being computed and not yet available in the local memory. These situations can be detected by comparing the exponents e, e' and e'' of three successively incoming summands. In principle all pipeline conflicts can be resolved by the hardware. Here we discuss the resolution of the two pipeline conflicts which are by far the most likely.

One conflict situation occurs if two consecutive products carry the same exponent e. In this case the two summands map onto the same three words of the CR. Then the second summand is unable to read its partner for the addition from the local memory because it is not yet available. This situation is checked by the hardware where the exponents e and e' of two consecutive summands are compared. If they are identical, the multiplexer blocks off the process of reading from the local memory. Instead the sum which is just being computed is written back directly into the register before summation (RBS) via the multiplexer so that the second summand can be added immediately without memory involvement.

Another possible pipeline conflict occurs if from three successively incoming summands the first one and the third one carry the same exponent. Since the pipeline consists of three steps, the partner for the addition of the third one is not yet in the local memory but is still in the register after summation (RAS). This situation is checked by the hardware also, see Figure 8.18. There the two exponents e and e'' of the two summands are compared. In the case of coincidence the multiplexer again suppresses the reading from the local memory. Instead, the result of the former addition, which is still in the RAS, is directly written back into the RBS via the multiplexer. So this pipeline conflict can also be resolved by the hardware without memory involvement.

The case $e = e' = e''$ is also possible. It would cause a reading conflict in the multiplexer. The situation can be avoided by writing a dummy exponent into e'' or by reading from the add/subtract unit with higher priority.

The product that arrives at the accumulation unit maps onto three consecutive words of the CR. A more significant fourth word absorbs the possible carry. The solution for the two pipeline conflicts just described works well if this fourth word is the next more significant word. A carry is not absorbed by the fourth word if all its bits are one, or are all zero in case of a subtraction. The probability that this is the case is $1 : 2^{64} < 10^{-19}$. In the vast majority of instances this will not be the case.

If it is the case the word which absorbs the carry is selected by the flag mechanism and read into the most significant word of the RBS. The addition step then again works well including the carry resolution. But difficulties occur in both cases of pipeline conflict. Figure 8.19 displays a certain part of the CR. The three words to which the addition is executed are denoted by 1, 2 and 3. The next more significant word is denoted by 4 and the word which absorbs the carry by 5.

For a pipeline conflict with $e = e'$ or $e = e''$ the following addition again touches the words 1, 2 and 3. Now the carry is absorbed either by word 4 or by word 5. Word 4 absorbs the carry if an addition is followed by an addition or a subtraction by a

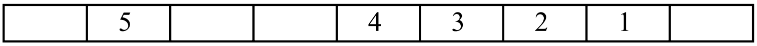


Figure 8.19. Carry propagation for pipeline conflict.

subtraction. Word 5 absorbs the carry if an addition is followed by a subtraction or vice versa. So the hardware has to take care that either word 4 or 5 is read into the most significant word of the RBS depending on the operation which follows. The case that word 5 is the carry word again needs no particular care. Word 5 is already in the most significant position of the RBS. It is simply treated the same way as the words 1, 2 and 3. In the other case word 4 has to be read from the CR into the RBS, simultaneously with the words 1, 2 and 3 from the add/subtract unit or from the RAS into the RBS. In this case word 5 is written into the local memory via the normal write path.

So far certain solutions for the possible pipeline conflicts $e = e'$ and $e = e''$ have been discussed. These are the most frequent but not the only conflicts that may occur. Similar difficulties appear if two or three successive incoming summands overlap only partially. In this case the exponents e and e' and/or e'' differ by 1 or 2 so that these situations can be also detected by comparison of the exponents. Another pipeline conflict appears if one of the two following summands overlaps with a carry word. In these cases summands have to be built up in parts from the adder/subtracter or from the RAS and the CR. Thus hardware solutions for these situations are more complicated and costly. We leave a detailed study of these situations to the reader/designer and offer the following alternative. The accumulation pipeline consists of three steps only. Instead of investing in a lot of hardware logic for rare cases of pipeline conflicts it may be simpler and less expensive to stall the pipeline and delay the accumulation by one or two cycles as needed. It should be mentioned that other details, for instance the width of the adder that is used, can also greatly change the design aspects. A 128 bit instead of the 64 bit adder width which was assumed here could simplify several details.

It was already mentioned that the probability for the carry to run further than the fourth word is less than 10^{-19}. However, a particular situation where this happens occurs if the sum changes its sign. This can happen frequently. To avoid a complicated carry handling procedure in this case a small carry counter of perhaps three bits could be appended to each 64 bit word of the CR. If these counters are not zero at the end of the accumulation their contents have to be added to the CR. For further details see [285, 286].

As was pointed out in connection with the unit discussed in Section 8.5.3, the addition of the summand actually can be carried out over 170 bits only. Thus the shifter that is shown in Figure 8.18 can be reduced to a 106 to 170 bits shifter and the data path from the shifter to the input register IR as well as the one to the RBS also need to be 170 bits wide only.

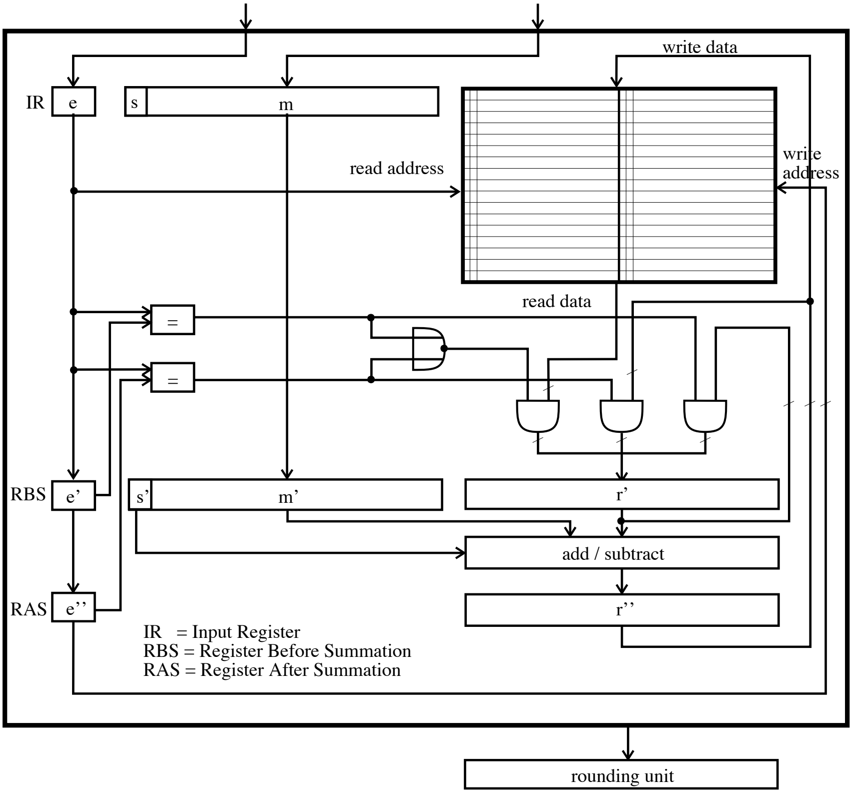


Figure 8.20. Block diagram for an SPU with short adder and local store for a 32 bit data word and 64 bit data bus.

8.8.5 Carry-Free Accumulation of Products in Redundant Arithmetic

So far in this chapter various scalar product units have been studied. The flag mechanism discussed in Section 8.4.4 was a basic technique for fast carry resolution. Its implementation requires logic and hardware which uses some silicon area on the chip. See, for instance, the chip displayed in Figure 8.8. Extra logic and circuitry is needed to deal with carry resolution for the pipeline conflicts of the scalar product unit discussed in the previous subsection.

Instead of using more and more logic and hardware to deal with complicated situations, an alternative approach to the problem may be simpler and more attractive. *Redundant number representation* and arithmetic, binary signed-digit arithmetic in particular, would allow a carry-free accumulation of the products into the complete register.

Two bits are needed to represent a binary digit in binary signed-digit number representation. So the complete register would double in size. On the other hand, the carry resolution logic would disappear.

Binary signed-digit arithmetic is a little more complicated than conventional binary arithmetic. Nevertheless, eliminating the carry propagation makes it very fast. The addition time is independent of the word length of the operands in binary signed-digit arithmetic.

Addition of a product to the CR in signed-digit arithmetic would affect the CR only locally. Thus a short adder would suffice to accumulate the products into the CR. At the end of the accumulation a conventional binary subtraction of two CR contents may be necessary to obtain a conventional binary number. This long subtraction is about of the same complexity as a single multiplication.

Very fast multipliers using binary signed-digit number representation and arithmetic have been designed and used successfully. See [579, 580]. The subject is well studied. See, for instance, [309] and the literature cited there.

Thus a general use of binary signed-digit number representation and arithmetic in the SPU seems to be very attractive.

8.9 Hardware Complete Register Window

So far it has been assumed in this chapter that the SPU is incorporated as an integral part of the arithmetic unit of the processor. Now we discuss the question of what can be done if this is not the case and if not enough register space for the CR is available on the processor.

The final result of a scalar product computation is assumed to be a floating-point number with an exponent in the range $e1 \leq e \leq e2$. If this is not the case, the problem has to be scaled. During the computation of the scalar product, however, summands with an exponent outside of this range may occur. The remaining computation then has to cancel all the digits outside of the range $e1 \leq e \leq e2$. So in a normal scalar product computation, the register space outside this range will be used less frequently. It was mentioned earlier in this book that the conclusion should not be drawn from this consideration that the register size can be restricted to the single exponent range in order to save some silicon area. This would require the use of complicated exception handling routines in software or in hardware. The latter may finally require as much silicon. A software solution certainly is much slower. The hardware requirement for the CR with standard arithmetic is modest and the necessary register space really should be invested.

However, the memory space for the CR on the arithmetic unit grows with the exponent range of the data format. If this range is extremely large, as for instance for an extended precision floating-point format, then only an inner part of the CR can be supported by hardware. We call this part of the CR a *Hardware Complete Register Window* (HCRW). See Figure 8.21. The outer parts of this window must then be handled in software. Probably they would seldom be used.

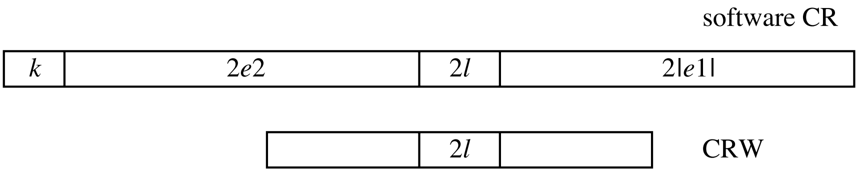


Figure 8.21. Hardware Complete Register Window (HCRW).

There are still other reasons that support the development of techniques for the computation of the exact scalar product using an HCRW. Many conventional computers on the market do not provide enough register space to represent the full CR on the CPU. Then an HCRW is one choice which allows a fast and exact computation of the scalar product in many cases.

Another possibility is to place the CR in the user memory, i.e., in the data cache. In this case only the starting address of the CR and the flag bits are put into (fixed) registers of the general purpose register set of the computer. This solution has the advantage that only a few registers are needed and that a longer CR window or even the full CR can be provided. This reduces the need to handle exceptions. The disadvantage of this solution is that for each accumulation step, four memory words must be read and written in addition to the two operand loads. So the scalar product computation speed is limited by the data cache to processor transfer bandwidth and speed. If the full CR is provided this is a very natural solution. It has been realized on several IBM, SIEMENS and HITACHI computers of the /370 architecture in the 1980s [650, 651, 653, 663].

A faster solution certainly is obtained for many applications with an HCRW in the general purpose register set of the processor. Here only a part of the CR is present in hardware. Overflows and underflows of this window have to be handled by software. A full CR for the double precision data format of the IEEE arithmetic standard 754 requires 4288 bits or 67 words of 64 bits. We assume here that only ten of these words are located in the general purpose register set.

Such a window covers the full CR that is needed for a scalar product computation in case of the single precision data format of the IEEE arithmetic standard 754. It also allows exact computation of scalar products in the case of the long data format of the /370 architecture as long as no under- or overflows occur. In this case $64+28+63 = 155$ hexadecimal digits or 620 bits are required. With an HCRW of 640 bits all scalar products that do not cause an under- or overflow could have been correctly computed on these machines. This architecture was successfully used and even dominated the market for more than 20 years. This example shows that even if an HCRW of only 640 bits is available, the vast majority of scalar products will execute on fast hardware.

Of course, even if only an HCRW is available, all scalar products should be computed exactly. Any operation that over- or underflows the HCRW must be completed in software. This requires a complete software implementation of the CR, i.e., a variable of type `complete`. All additions that do not fit into the HCRW must be executed in software into this `complete` variable.

There are three situations where the HCRW can not accumulate the product exactly:

- The exponent of the product is so high that the product does not (completely) fit into the HCRW. Then the product is added in software to the `complete` variable.
- The exponent of the product is so low that the product does not (completely) fit into the HCRW. Then the product is added in software to the `complete` variable.
- The product fits into the HCRW, but its accumulation causes a carry to be propagated outside the range of the HCRW. In this case the product is added into the HCRW. The carry must be added in software to the `complete` variable.

If at the end of the accumulation the contents of the software CR are not zero, the contents of the HCRW must be added to the software CR to obtain the correct value of the scalar product. Then a rounding can be performed if required. If at the end of the accumulation the contents of the software CR are zero, the HCRW contains the exact value of the scalar product and a rounded value can be obtained from it.

Thus, in general, a software controlled full CR supplements an HCRW. The software routines must be able to perform the following functions:

- clear the software CR. This routine must be called during the initialization of the HCRW. Ideally, this routine only sets a flag. The actual clearing is only done if the software CR is needed.
- add or subtract a product to/from the software CR.

- add or subtract a carry or borrow to/from the software CR at the appropriate digit position.
- add the HCRW to the software CR. This is required to produce the final result when both the HCRW and the software CR were used. Then a rounding can be performed.
- round the software CR to a floating-point number.

With this software support scalar products can be computed exactly using an HCRW at the cost of a substantial software overhead and a considerable time penalty for products that fall outside the range of the HCRW. The software overhead caused by the reduction of the full width of the CR to an HCRW represents the trade off between hardware expenditure and runtime.

A much weaker alternative solution to the HCRW-software environment just described is to discard the products that underflow the HCRW. A counter variable is used to count the number of discarded products. If a number of products were discarded, the last bits of the HCRW must be considered invalid. A valid rounded result can be generated by hardware if these bits are not needed. If this procedure fails to produce a useful answer the whole accumulation is repeated in software using a full CR.

A 640 bit HCRW seems to be the shortest satisfactory hardware window. If this much register space is not available, a software implementation probably is the best solution.

If a shorter HCRW must be implemented, then it should be a movable window. This can be represented by an exponent register associated with the hardware window. At the beginning of an accumulation, the exponent register is set so that the window covers the least significant portion of the CR. Whenever a product would cause the window to overflow, its exponent tag is adjusted, i.e., the window moves to the left, so that the product fits into the window. Products that would cause an underflow are counted and otherwise ignored. The rounding instruction checks whether enough significant digits are left to produce a correctly rounded result or whether too much cancellation did occur. In the latter case it is up to the user to accept the inexact result or to repeat the whole accumulation in software using a full CR.

Using this technique an HCRW as short as 256 bits could be used to perform rounded scalar product computation and quadruple precision arithmetic. However, it would not be possible to perform many other nice and useful applications of the exact scalar product with this type of scalar product hardware such as, for instance, complete arithmetic, a *long real* arithmetic or a *long interval* arithmetic.

To allow fast execution of a number of multiple precision arithmetics the HCRW should not be too small.

Part III

Principles of Verified Computing

Chapter 9
Sample Applications

Floating-point arithmetic is the fast way to perform scientific and engineering calculations. Today, *individual* floating-point operations are maximally accurate in general. However, after as few as two operations, the computed result might be completely wrong. Computers can now carry out up to 10^{14} or 10^{15} floating-point operations in a second. Therefore particular attention must be paid to the reliability of the computed results. Conventional error analysis is impossible for computation at such speed.

Numerical analysts have devised algorithms and techniques which make it possible for the computer itself to verify the correctness of computed results, and even to establish the existence and uniqueness of a solution within computed close bounds, for numerous problems and applications. Such techniques are termed *automatic result verification.*

Advanced computer arithmetic as treated in this book provides the basis for results to be easily and automatically verified on the computer. Interval arithmetic brings the continuum to the computer. Fixed-point accumulation of products makes it easy and quite natural to ensure the accuracy of a computation. A verified solution of an ordinary differential equation, for instance, is exact just as is a solution obtained by a computer algebra system, which as a rule still requires a valid formula evaluation.

In this chapter we go over a few sample applications showing the use of advanced computer arithmetic. We sketch what automatic result verification means and how it works in the case of these examples.

In the first section we develop some basic aspects of interval mathematics. Interval arithmetic is introduced as a shorthand notation and an automatic calculus to deal with inequalities. Interval operations are also interpreted as special power set operations. The inclusion isotony and the inclusion property are central and important consequences of this property. They can be used to enclose the range of a function's values. Advanced techniques for such enclosure by centered forms or by subdivision are also discussed. These techniques are used, for instance, for global optimization.

In combination with automatic differentiation, interval arithmetic allows computation of enclosures of derivatives, of Taylor coefficients, of gradients, of Jacobian or Hessian matrices.

Evaluation of a function for an interval X delivers a superset of the function's values over X. This overestimation tends to zero with the width of the interval X. Thus for small intervals interval evaluation of a function practically delivers the function's values. Many numerical methods proceed in small steps. So this property together with differentiation arithmetic to compute enclosures of derivatives is the key technique for validated numerical computation of integrals, for solution of ordinary differential equations, of refined techniques for global optimization and for many other applications.

Solving systems of linear equations is basic to numerical computation. We show how highly accurate guaranteed bounds for the solution can be obtained, based on an approximate solution, and how the existence and uniqueness of the solution within these bounds can be proved by the computer. It may well happen that an attempt to verify the correctness of a computed result fails to produce a correct answer. This is detected by the computer, and in this case the computer itself can choose an alternative algorithm or repeat the computation using higher precision. We show that by using the exact scalar product a highly accurate enclosure of the solution can be obtained even in very ill conditioned cases without using higher precision arithmetic.

Newton's method is considered in two sections of this chapter. In conventional numerical analysis it is the crucial method for nonlinear problems. Traditionally Newton's method is used to compute an approximate solution of a nonlinear real function. It is well known that the method converges quadratically to the solution if the function is twice continuously differentiable and the starting value is sufficiently close to the solution. If these conditions do not hold, and in other cases, the method may well fail in finite as well as in infinite precision arithmetic even if there is only a single solution in a given interval. The method is only locally convergent. In contrast to this, the interval version of Newton's method is globally convergent. Beginning with a rough enclosure of the solution it computes a nested sequence of enclosures. The bounds finally converge quadratically to the solution. The interval version of Newton's method never diverges or fails otherwise, not even in rounded arithmetic.

Newton's method reaches its ultimate elegance and strength in the *extended interval Newton method.* It separates different solutions, and it encloses all single zeros in a given domain. It is globally convergent and it never fails. The method is locally quadratically convergent.

Arithmetic expressions are basic ingredients of all numerical computing. In Section 9.6 a method is developed which evaluates an arithmetic expression with guaranteed high accuracy. The method uses fast interval arithmetic and a fast and exact scalar product. If these operations are hardware supported the computing time is of the same order as that for a conventional evaluation of the expression in floating-point arithmetic. If the floating-point evaluation fails completely additional computing time and storage are needed.

With a fast and exact multiply and accumulate operation or an exact scalar product fast quadruple or multiple precision arithmetics can easily be provided on the computer for real as well as for interval data. These operations are developed in the last section of this chapter.

9.1 Basic Properties of Interval Mathematics

9.1.1 Interval Arithmetic, a Powerful Calculus to Deal with Inequalities

Problems in technology and science are often described by an equation or by a system of equations. Mathematics is used to manipulate these equations in order to obtain a solution. The Gauss algorithm, for instance, is used to compute the solution of a system of linear equations by adding, subtracting, multiplying and dividing equations in a systematic manner. Newton's method is used to compute approximately the location of a zero of a nonlinear function or of a system of such functions.

Data are often given by bounds rather than by simple numbers. Bounds are expressed by inequalities. To compute bounds for the solution to a problem requires a systematic calculus to deal with inequalities. Interval arithmetic provides this calculus. It supplies the basic rules for how to add, subtract, multiply, divide, and otherwise manipulate inequalities in a systematic manner: Let bounds for two real numbers a and b be given by the inequalities $a_1 \leq a \leq a_2$ and $b_1 \leq b \leq b_2$. Addition of these inequalities leads to bounds for the sum $a + b$:

$$a_1 + b_1 \leq a + b \leq a_2 + b_2.$$

The inequality for b is reversed by multiplication with -1: $-b_2 \leq -b \leq -b_1$. Addition to the inequality for a then delivers the rule for the subtraction of one inequality from another:

$$a_1 - b_2 \leq a - b \leq a_2 - b_1.$$

Interval arithmetic provides a shorthand notation for these rules by suppressing the $\leq$ symbols. We simply identify the inequality $a_1 \leq a \leq a_2$ with the closed and bounded real interval $[a_1, a_2]$. The rules for addition and subtraction for two such intervals now read:

$$[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2], \qquad (9.1.1)$$

$$[a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1]. \qquad (9.1.2)$$

The rule for multiplication of two intervals is more complicated. Nine cases are to be distinguished depending on whether a_1, a_2, b_1, b_2, are less or greater than zero. For division the situation is similar. In conventional interval arithmetic division A/B is not defined if $0 \in B$.

As a result of these rules it can be stated that for real intervals the result of an interval operation $A \circ B$, for all $\circ \in \{+, -, \cdot, /\}$, can be expressed in terms of the bounds of the interval operands (with the A/B exception above). In order to get each of these bounds, typically only one real operation is necessary. Only in one case of multiplication, $0 \in \mathring{A}$ and $0 \in \mathring{B}$, two products have to be calculated and compared. Here $\mathring{A}$ denotes the interior of A, i.e., $c \in \mathring{A}$ means $a_1 < c < a_2$.

We illustrate the efficiency of this calculus for inequalities by a simple example. See [13]. Let $x = Ax + b$ be a system of linear equations in fixed-point form with a contracting real matrix A and a real vector b, and let the interval vector X be a rough initial enclosure of the solution $x^* \in X$. For this interval vector X we can now formally write down the Jacobi method, the Gauss-Seidel method, a relaxation method or some other iterative scheme for the solution of the linear system. Doing so we obtain a number of iterative methods for the computation of enclosures of linear systems of equations. Further iterative schemes then can be obtained by taking the intersection of two successive approximations. If we now decompose all these methods in formulas for the bounds of the intervals we obtain a major number of methods for the computation of bounds for the solution of linear systems which have been derived by well-known mathematicians painstakingly about 40 years ago, see [28, 135]. The calculus of interval arithmetic reproduces these and other methods in the simplest way. The user does not have to take care of the many case distinctions occurring in the matrix vector multiplications. The computer executes them automatically by the preprogrammed calculus. Also the rounding errors are enclosed. The calculus evolves its own dynamics.

9.1.2 Interval Arithmetic as Executable Set Operations

The rules for interval operations can also be interpreted as arithmetic operations for sets. As such they are special cases of general set operations. Further important properties of interval arithmetic can immediately be obtained via set operations. Let M be any set with a dyadic operation $\circ : M \times M \to M$ defined for its elements. The power set $\mathbb{P}M$ of M is defined as the set of all subsets of M. The operation $\circ$ in M can be extended to the power set $\mathbb{P}M$ by the following definition

$$A \circ B := \{a \circ b \mid a \in A \wedge b \in B\} \text{ for all } A, B \in \mathbb{P}M. \tag{9.1.3}$$

The least element in $\mathbb{P}M$ with respect to set inclusion as an order relation is the empty set $\varnothing$. The greatest element is the set M. The empty set is a subset of any set. Any arithmetic operation on the empty set produces the empty set.

The following properties are obvious and immediate consequences of (9.1.3):

$$A \subseteq B \wedge C \subseteq D \Rightarrow A \circ C \subseteq B \circ D \text{ for all } A, B, C, D \in \mathbb{P}M, \tag{9.1.4}$$

and in particular

$$a \in A \wedge b \in B \Rightarrow a \circ b \in A \circ B \text{ for all } A, B \in \mathbb{P}M. \tag{9.1.5}$$

Property (9.1.4) is called the *inclusion isotony* (or *inclusion monotony*). (9.1.5) is called the *inclusion property*.

By use of parentheses these rules can immediately be extended to expressions with more than one arithmetic operation, e.g.,

$$A \subseteq B \wedge C \subseteq D \wedge E \subseteq F \Rightarrow A \circ C \subseteq B \circ D \Rightarrow (A \circ C) \circ E \subseteq (B \circ D) \circ F,$$

and so on. Moreover, if more than one operation is defined in M this chain of conclusions also remains valid for expressions containing several different operations.

If we now replace the general set M by the set of real numbers, (9.1.3), (9.1.4), and (9.1.5) hold in particular for the power set $\mathbb{PR}$ of the real numbers $\mathbb{R}$. This is the case for all operations $\circ \in \{+, -, \cdot, /\}$, if we assume that in case of division 0 is not an element of the divisor, for instance, $0 \notin B$ in (9.1.3).

The set $I\mathbb{R}$ of closed and bounded intervals over $\mathbb{R}$ is a subset of $\mathbb{PR}$. Thus (9.1.3), (9.1.4), and (9.1.5) are also valid for elements of $I\mathbb{R}$. The set $I\mathbb{R}$ with the operations (9.1.3), $\circ \in \{+, -, \cdot, /\}$, is an algebraically closed[1] subset within $\mathbb{PR}$. That is, if (9.1.3) is performed for two intervals $A, B \in I\mathbb{R}$ the result is always an interval again. This holds for all operations $\circ \in \{+, -, \cdot, /\}$ with $0 \notin B$ in the case of division. This property is a simple consequence of the fact that for all arithmetic operations $\circ \in \{+, -, \cdot, /\}$, $a \circ b$ is a continuous function of both variables. $A \circ B$ is the range of this function over the product set $A \times B$. Since A and B are closed intervals, $A \times B$ is a simply connected, bounded and closed subset of $\mathbb{R}^2$. In such a region the continuous function $a \circ b$ takes a maximum and a minimum as well as all values in between. Therefore

$$A \circ B = [\min_{a \in A, b \in B}(a \circ b), \max_{a \in A, b \in B}(a \circ b)], \text{ for all } \circ \in \{+, -, \cdot, /\},$$

provided that $0 \notin B$ in the case of division.

Consideration of (9.1.3), (9.1.4) and (9.1.5) for intervals of $I\mathbb{R}$ leads to the crucial properties of all applications of interval arithmetic. Because of the great importance of these properties we repeat them here. Thus we obtain for all operations $\circ \in \{+, -, \cdot, /\}$:

The *set definition* of interval arithmetic:

$$A \circ B := \{a \circ b \mid a \in A \wedge b \in B\} \quad \text{for all } A, B \in I\mathbb{R}, \; 0 \notin B \text{ in case of division,} \tag{9.1.6}$$

the *inclusion isotony* (or *inclusion monotony*):

$$A \subseteq B \wedge C \subseteq D \Rightarrow A \circ C \subseteq B \circ D \quad \text{for all } A, B, C, D \in I\mathbb{R}, \; 0 \notin C, D \text{ in case of division,} \tag{9.1.7}$$

[1] As the integers are within the reals for $\circ \in \{+, -, \cdot\}$.

and in particular the *inclusion property*:

$$a \in A \wedge b \in B \Rightarrow a \circ b \in A \circ B \quad \text{for all } A, B \in I\mathbb{R}, \atop \qquad\qquad\qquad\qquad\qquad 0 \notin B \text{ in case of division.} \tag{9.1.8}$$

If for $M = \mathbb{R}$ in (9.1.3) the number of elements in A or B is infinite, the operations are effectively not executable because infinitely many real operations would have to be performed. If A and B are intervals of $I\mathbb{R}$, however, the situation is different. In general A or B or both will again contain infinitely many real numbers. The result of the operation (9.1.6), however, can now be performed by a finite number of operations with real numbers, with the bounds of A and B. For all operations $\circ \in \{+, -, \cdot, /\}$ the result is obtained by the explicit formulas derived in Chapter 4 of this book or at [353, 367].

For intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$ the formulas for the interval operations can be summarized by

$$A \circ B = [\min_{i,j=1,2}(a_i \circ b_j), \max_{i,j=1,2}(a_i \circ b_j)] \text{ for all } \circ \in \{+, -, \cdot, /\}, \tag{9.1.9}$$

with $0 \notin B$ in the case of division. (9.1.9), however, should not be interpreted as recommendation for the computation of the result of interval operations.

Since interval operations are particular power set operations, the inclusion isotony and the inclusion property also hold for expressions with more than one arithmetic operation.

In programming languages the concept of an arithmetic expression is usually defined to be a little more general. Besides constants and variables, elementary functions (sometimes called standard functions) like sqr, sqrt, sin, cos, exp, log, tan may also be elementary ingredients. All these are put together with arithmetic operators and parentheses into the general concept of an arithmetic expression. This construct is illustrated by the syntax diagram of Figure 9.1. Therein solid lines are to be traversed from left to right and from top to bottom. Dotted lines are to be traversed oppositely, i.e., from right to left and from bottom to top. In Figure 9.1 the syntax variable REAL FUNCTION merely represents a real arithmetic expression hidden in a subroutine.

Now we define the general concept of an arithmetic expression for the data type `interval` by exchanging the data type `real` in Figure 9.1 for the data type `interval`. This results in the syntax diagram for INTERVAL EXPRESSION shown in Figure 9.2. In Figure 9.2 the syntax variable INTERVAL FUNCTION represents an interval expression hidden in a subroutine.

In the syntax diagram for INTERVAL EXPRESSION in Figure 9.2 the concept of an interval elementary function is not yet defined. We simply define it as the range of the function's values taken over an interval (within the domain of definition $D(f)$ of the function). In case of a real function f we denote the range of values over the interval $[a_1, a_2]$ by

$$f([a_1, a_2]) := \{f(a) \mid a \in [a_1, a_2]\}, \text{ with } [a_1, a_2] \in D(f).$$

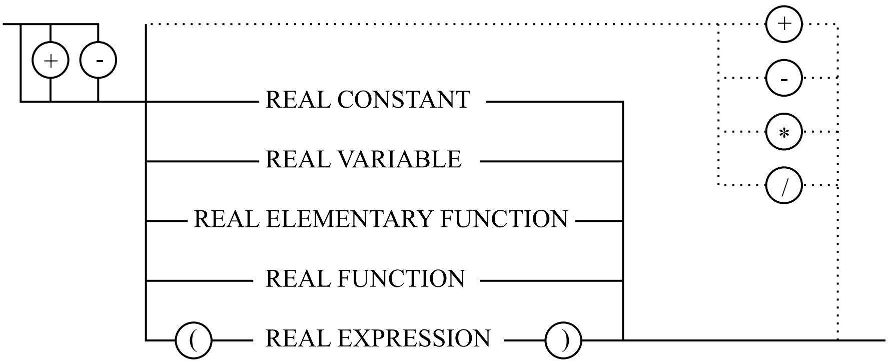


Figure 9.1. Syntax diagram for REAL EXPRESSION.

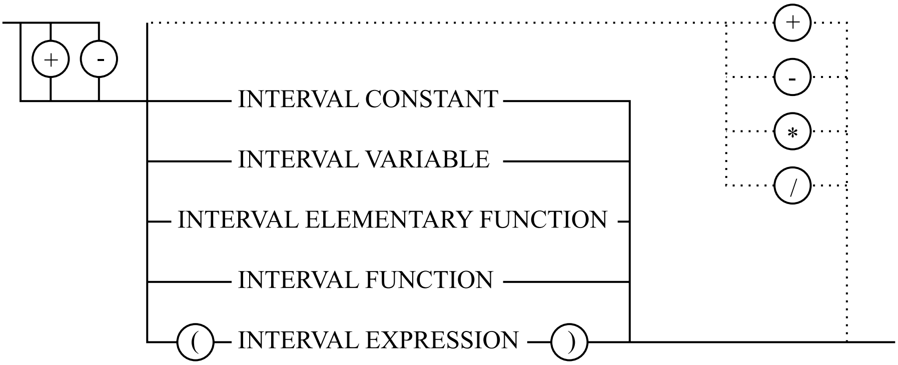


Figure 9.2. Syntax diagram for INTERVAL EXPRESSION.

For instance:

$$\begin{aligned}
e^{[a_1,a_2]} &= [e^{a_1}, e^{a_2}],\\
[a_1,a_2]^{2n} &= \begin{cases} [\min(a_1^{2n}, a_2^{2n}), \max(a_1^{2n}, a_2^{2n})] & \text{for } 0 \notin [a_1,a_2],\\ [0, \max(a_1^{2n}, a_2^{2n})] & \text{for } 0 \in [a_1,a_2], \end{cases}\\
\sin[-\pi/4, \pi/4] &= [-\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2}],\\
\cos[0, \pi/2] &= [0,1].
\end{aligned}$$

For non-monotonic functions the computation of the range of values over an interval $[a_1,a_2]$ requires the determination of the global minimum and maximum of the function in the interval $[a_1,a_2]$. For the usual elementary functions, however, these are known. With this definition of elementary functions for intervals, the key properties of interval arithmetic, the inclusion monotony (9.1.5) and the inclusion property (9.1.6) extend immediately to elementary functions and with this to interval expressions as defined in Figure 9.2:

$$A \subseteq B \Rightarrow f(A) \subseteq f(B), \text{ with } A, B \in I\mathbb{R} \qquad \text{(inclusion isotone)},$$

and in particular for $a \in \mathbb{R}$ and $A \in I\mathbb{R}$:

$$a \in A \Rightarrow f(a) \in f(A) \qquad \text{(inclusion property)}.$$

We summarize the development so far by stating that interval arithmetic expressions are generally inclusion isotone and that the inclusion property holds. These are the key properties of interval arithmetic. They give interval arithmetic its raison d'être. To start with, they provide the possibility of enclosing imprecise data within bounds and then continuing the computation with these bounds. This always results in guaranteed enclosures.

As the next step we define a (computable) real function simply by a real arithmetic expression. We need the concept of an *interval evaluation of a real function.* It is defined as follows: In the arithmetic expression for the function all operands are replaced by intervals and all operations by interval operations (where all intervals must be within the domain of definition of the real operands). This is just the step from Figure 9.1 to Figure 9.2. What is obtained is an interval expression. Then all arithmetic operations are performed in interval arithmetic. For a real function $f(a)$ we denote the interval evaluation over the interval A by $F(A)$.

With this definition we can immediately conclude that interval evaluations of (computable) real functions are inclusion isotone and that the inclusion property holds in particular:

$$A \subseteq B \Rightarrow F(A) \subseteq F(B) \qquad \text{(inclusion isotone)}, \qquad (9.1.10)$$

$$a \in A \Rightarrow f(a) \in F(A) \qquad \text{(inclusion property)}. \qquad (9.1.11)$$

These concepts immediately extend in a natural way to functions of several real variables. In this case in (9.1.11) a is an n-tuple, $a = (a_1, a_2, \ldots, a_n)$, and A and B are higher dimensional intervals, e.g., $A = (A_1, A_2, \ldots, A_n)$, with $A_i \in I\mathbb{R}$ for all $i = 1(1)n$.

Remark 9.1. Two different real arithmetic expressions can define equivalent real functions, for instance:

$$f(x) = x(x-1) \quad \text{and} \quad g(x) = x^2 - x.$$

Evaluation of the two expressions for a real number always leads to the same real function value. In contrast to this, interval evaluation of the two expressions may lead to different intervals. In the example we obtain for the interval $A = [1,2]$:

$$\begin{aligned} F(A) &= [1,2]([1,2] + [-1,-1]) & G(A) &= [1,2][1,2] - [1,2] \\ &= [1,2][0,1] = [0,2], & &= [1,4] - [1,2] = [-1,3]. \quad \blacksquare \end{aligned}$$

A look at the syntax diagram for INTERVAL EXPRESSION, Figure 9.2, shows that operator overloading in a programming language does not suffice to support interval arithmetic effectively. Also, the elementary functions must be provided for interval arguments and the concept of a function subroutine must not be restricted to the data types `integer` and `real`. It must be extended to the data type `interval`.

Implementing the elementary functions for the data type `interval` is a great challenge for the mathematician. For an elementary function of the data type `real` the computer provides a result, the accuracy of which cannot easily be judged by the user. This is no longer the case when the elementary functions are provided for interval arguments. Then, if called for a point interval (where the lower and upper bound coincide), a comparison of the lower and upper bound of the result of the interval evaluation of the function reveals immediately the accuracy with which the elementary function has been implemented. Extremely careful implementation of the elementary functions is needed to get such results, and since interval versions of the elementary functions have been provided on a large scale [288, 289, 290, 291, 355, 356, 653] the conventional real elementary functions on computers also had to be and have been improved step by step by the manufacturers. A highly advanced programming environment in this respect is a decimal version of PASCAL-XSC [79, 80] where, besides the usual 24 elementary functions, about the same number of special functions are provided for real and interval arguments with highest accuracy.

Ease of programming is essential for any sophisticated use of interval arithmetic. If it is not so, coding difficulties absorb all the attention and capacity of users and prevent them from developing deeper mathematical ideas and insight. It is a matter of fact that a great many of the existing and established interval methods and algorithms have originally been developed in PASCAL-XSC even if they have been coded afterwards in other languages. Programming ease is essential indeed.

We summarize this discussion by stating that it does not suffice for an adequate use of interval arithmetic on computers that only the four basic arithmetic operations $+, -, \cdot$ and $/$ for intervals are somehow supported by the computer hardware. An appropriate language support is absolutely necessary. Two things seem to be necessary for a major breakthrough in interval mathematics. A leading vendor has to provide the necessary hardware and software support and the body of numerical analysts must acquire a broader insight and skills to use this support.

9.1.3 Enclosing the Range of Function Values

The interval evaluation of a real function f over the interval A was denoted by $F(A)$. We now compare it with the range of function values over the interval A which was denoted by

$$f(A) := \{f(a) \mid a \in A\}. \tag{9.1.12}$$

We have observed that interval evaluation of an arithmetic expression and of real functions is inclusion isotone (9.1.7), (9.1.10) and that the inclusion property (9.1.8), (9.1.11) holds. Since (9.1.8) and (9.1.11) hold for all $a \in A$ we can immediately state that

$$f(A) \subseteq F(A), \tag{9.1.13}$$

i.e., that the interval evaluation of a real function over an interval delivers a superset of the range of function values over that interval. If A is a point interval $[a, a]$ this reduces to:

$$f(a) \in F([a, a]). \tag{9.1.14}$$

These are basic properties of interval arithmetic. Computing with inequalities always aims for bounds for function values, or for bounds for the range of function values. Interval arithmetic provides for this computation in principle.

Many applications need the range of function values over an interval. Its computation is a very difficult task. It is equivalent to the computation of the global minimum and maximum of the function in that interval. On the other hand, interval evaluation of an arithmetic expression is very easy to perform. It requires about twice as many real arithmetic operations as an evaluation of the expression in real arithmetic. Thus interval arithmetic provides an easy means to compute upper and lower bounds for the range of values of an arithmetic expression.

In essence, any complicated algorithm performs a sequence of arithmetic expressions. Thus interval evaluation of such an algorithm would compute bounds for the result from given bounds for the data. In practice however, the sizes of the intervals grow very fast and for lengthy algorithms the bounds quickly become meaningless, especially if the bounds for the starting data are already wide. This raises the question of whether measures can be taken to keep the sizes of the intervals from growing too much. Interval mathematics has developed such measures and we are going to sketch these now.

If an enclosure for a function value is computed by (9.1.14), the quality of the computed result $F([a,a])$ can be judged by the diameter of the interval $F([a,a])$. This ability to easily judge the quality of a computed result is not available with (9.1.13). Even if $F(A)$ is a large interval, it can be a good approximation for the range of function values $f(A)$ if the latter is large also. So some means to measure the deviation between $f(A)$ and $F(A)$ in (9.1.13) is desirable.

It is well known that the set $I\mathbb{R}$ of real intervals becomes a metric space with the so-called Hausdorff metric, where the distance q between two intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$ is defined by

$$q(A,B) := \max\{|a_1 - b_1|, |a_2 - b_2|\}. \tag{9.1.15}$$

See, for instance, [28, 29].

With this distance function q the following relation can be proved to hold under natural assumptions about f:

$$q(f(A), F(A)) \leq \alpha \cdot d(A), \text{ with a constant } \alpha \geq 0. \tag{9.1.16}$$

Here $d(A)$ denotes the diameter of the interval A:

$$d(A) := |a_2 - a_1|. \tag{9.1.17}$$

In case of functions of several real variables the maximum of the diameters $d(A_i)$ appears on the right hand side of (9.1.16).

The relation (9.1.16) shows that the distance between the range of values of the function f over the interval A and the interval evaluation of the expression for f tends to zero linearly with the diameter of the interval A. So the overestimation of $f(A)$ by $F(A)$ decreases with the diameter of A and in the limit $d(A) = 0$ and the overestimation vanishes.

Because of this result subdivision of the interval A into subintervals A_i, $i = 1(1)n$, with $A = \bigcup_{i=1}^{n} A_i$ is a frequently applied technique to obtain better approximations for the range of function values. Then (9.1.16) holds for each subinterval:

$$q(f(A_i), F(A_i)) \leq \alpha_i \cdot d(A_i), \text{ with } \alpha_i \geq 0 \text{ and } i = 1(1)n,$$

and, in general, the union of the interval evaluations over all subintervals

$$\bigcup_{i=1}^{n} F(A_i)$$

is a much better approximation for the range $f(A)$ than is $F(A)$.

There are yet other methods for better enclosing the range of function values $f(A)$. We have already observed that the interval evaluation $F(A)$ of a function f depends on the expression used for the representation of f. So by choosing appropriate representations for f the overestimation of $f(A)$ by the interval evaluation $F(A)$ can often be reduced. Indeed, if f allows a representation of the form

$$f(x) = f(c) + (x - c) \cdot h(x), \text{ with } c \in A, \tag{9.1.18}$$

then under natural assumptions on h

$$q(f(A), F(A)) \leq \beta \cdot (d(A))^2, \text{ with a constant } \beta \geq 0. \tag{9.1.19}$$

(9.1.18) is called a centered form of f. In (9.1.18) c is not necessarily the center of A although it is often chosen as the center. (9.1.19) shows that the distance between the range of values of the function f over the interval A and the interval evaluation of a centered form of f tends toward zero quadratically with the diameter of the interval A. In practice, this means that for small intervals the interval evaluation of the centered form leads to a very good approximation of the range of function values over an interval A. Again, subdivision is a method that can be applied in the case of a large interval A. It should be clear, however, that in general the bound in (9.1.19) is only better than in (9.1.16) for small intervals.

The reduced overestimation of the range of function values by the interval evaluation of the function with the diameter of the interval A, and the method of subdivision, are reasons why interval arithmetic can successfully be used in many applications. Numerical methods often proceed in small steps. This is the case, for instance, with numerical quadrature or cubature, or with numerical integration of ordinary differential equations. In all these cases an interval evaluation of the remainder term of the integration formula (using differentiation arithmetic) controls the step size of the integration, and anyhow because of the small steps, overestimation is practically negligible.

We now mention briefly how centered forms can be obtained. Usually a centered form is derived via the mean-value theorem. If f is differentiable in its domain D, then $f(x) = f(c) + f'(\xi)(x - c)$ for fixed $c \in D$ and some ξ between x and c. If x and c are elements within the interval $A \subseteq D$, then also $\xi \in A$. Therefore

$$f(x) \in F(A) := f(c) + F'(A)(A - c), \text{ for all } x \in A.$$

Here $F'(A)$ is an interval evaluation of $f'(x)$ in A.

In (9.1.18) the slope

$$h(x) = \frac{f(x) - f(c)}{x - c}$$

can be used instead of the derivative for the representation of $f(x)$. Slopes often lead to better enclosures for $f(A)$ than do derivatives. For details see [37, 337, 492].

Derivatives and enclosures of derivatives can be computed by a process which is called automatic differentiation or differentiation arithmetic. Slopes and enclosures of slopes can be computed by another process which is very similar to automatic differentiation. In both cases the computation of the derivative or slope or enclosures of these is done together with the computation of the function value. For these processes only the expression or algorithm for the function is required. No explicit formulas for the derivative or slope are needed. The computer interprets the arithmetic operations in the expression by differentiation or slope arithmetic. The arithmetic is hidden in the runtime system of the compiler. It is activated by type specification of the operands. For details see [37, 39, 205, 206, 337, 492, 494], and Section 9.2 on differentiation arithmetic. Thus the computer is able to produce and enclose the centered form via the derivative or slope automatically.

Without going into further details, we mention once more that none of these considerations is restricted to functions of a single real variable. Subdivision in higher dimensions, however, is a difficult task which requires additional tools and strategies. Typical of such problems are the computation of the bounds of the solution of a system of nonlinear equations, and global optimization or numerical integration of functions of more than one real variable. In all these and other cases, zero finding is a central task. Here the extended interval Newton method plays an extraordinary role so we review this method in Section 9.3.

9.1.4 Nonzero Property of a Function, Global Optimization

We first consider the question of whether a given function, defined by an arithmetic expression, has zeros in a given interval $X = [a, b]$, see Figure 9.3. This question cannot be answered with mathematical certainty if only floating-point arithmetic is available. All one can do is to evaluate the function at say 1000 points in the interval X. If all computed function values are positive, it is very likely that the function does not have any zeros in X. However, this conclusion is certainly not reliable. Because of rounding errors, a positive result could be computed for a negative function value. The function could also descend to a negative value between adjacent evaluation points with positive values (see the lower part of Figure 9.3). The question can be answered much more simply if interval arithmetic is available. A single interval evaluation of the function may suffice to solve the problem with complete mathematical certainty. We evaluate the function only once in interval arithmetic for the interval X. This delivers a superset $Y = F(X)$ of the range $f(X)$ of all function values over X. If this superset does not contain zero, as in our example, the range, which is a subset, does not contain zero. As a consequence, the function does not have any zeros in the interval X,

$$0 \notin Y = F(X) \Rightarrow 0 \notin f(X) \subseteq F(X) = Y.$$

Even this simple example shows that interval arithmetic is not necessarily more costly than floating-point arithmetic. It is a useful extension of floating-point arithmetic. The interval evaluation of the function is only twice as costly as a single floating-point evaluation, not to mention the 1000 floating-point evaluations cited above.

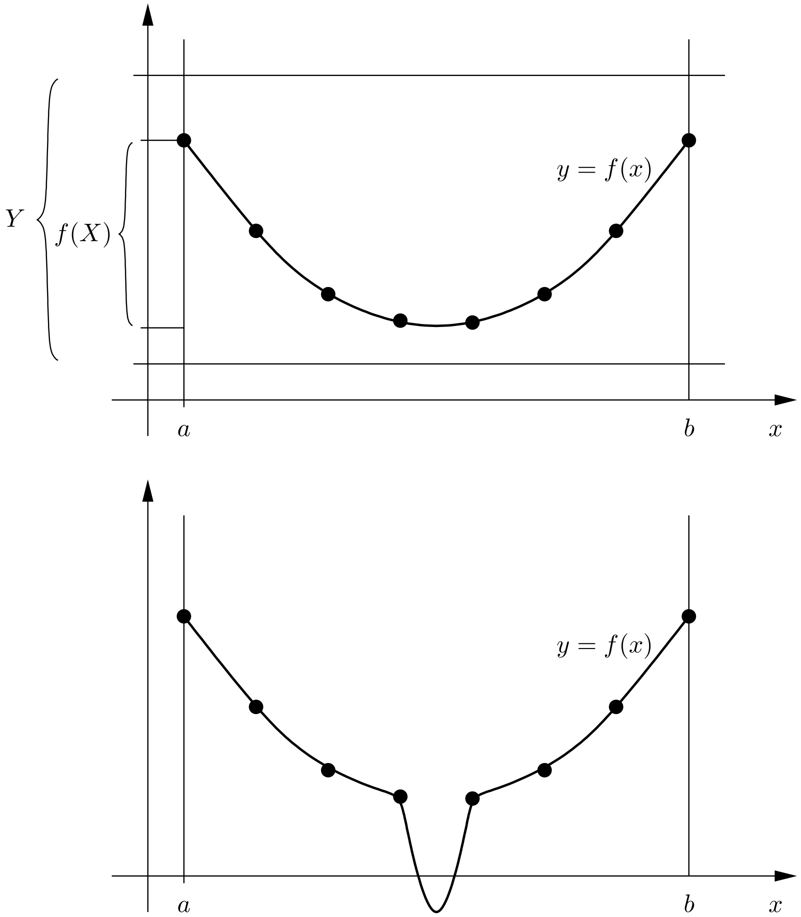


Figure 9.3. Non zero property of a function.

As the next example we consider the task of determining the global minimum of a function in a given domain. Again we restrict the discussion to the most simple, the one dimensional case, see Figure 9.4. The given domain is divided into a number of subintervals. In each subinterval, we now evaluate the function in interval arithmetic. This delivers a superset of the range of values of the function in each subinterval. Now we select the subinterval with the lowest lower bound for the range of function values. For an inner point of this subinterval we evaluate the function in interval arithmetic. This delivers a guaranteed upper bound of a function value. Those subintervals, the lower bound of which is greater than this function value can now be eliminated from further treatment, because they cannot contain the global minimum. The remaining subintervals (two in the example in Figure 9.4) now are further subdivided and treated by the same technique.

The method also works and succeeds very well for higher dimensions up to a certain limit, since whole continua can quickly be excluded from the search. In the literature [211, 212, 213, 491, 492, 494] more refined methods are used. Frequently, safe bounds for the global minimum can be computed faster than an approximation delivered by conventional techniques, the quality of which is still uncertain.

These methods, of course, can also be used for a fast and highly accurate computation of the range of values of a function in a given domain.

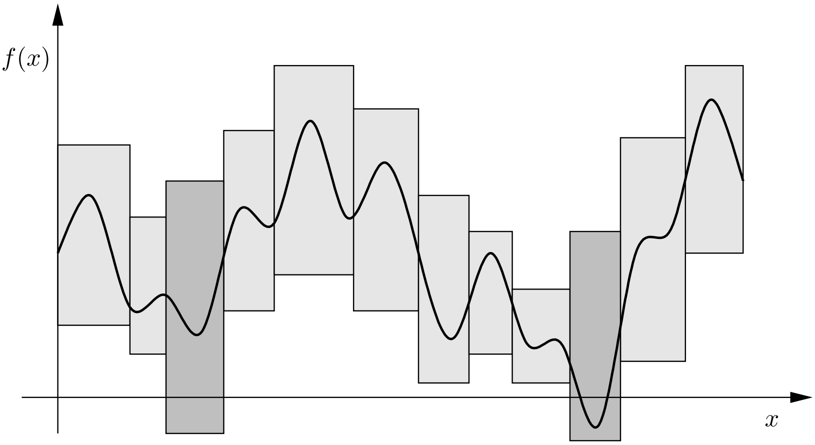


Figure 9.4. Global optimization.

9.2 Differentiation Arithmetic, Enclosures of Derivatives

For many applications in scientific computing the value of the derivative of a function is needed. The interval Newton method requires the computation of an enclosure of the first derivative of the function over an interval. The typical “school method” first computes a formal expression for the derivative of the function by applying well-known rules of differentiation. Then this expression is evaluated for a point or an interval. Differentiation arithmetic avoids the computation of a formal expression for the derivative. It computes derivatives or enclosures of derivatives just by computing with numbers or intervals. We are now going to sketch this method for the simplest case where the value of the first derivative is to be computed. If $u(x)$ and $v(x)$ are differentiable functions then the following rules for the computation of the derivative of the sum, difference, product, and quotient of the functions are well known:

$$\begin{array}{rclcl}(u(x) & + & v(x))' & = & u'(x)+v'(x),\\ (u(x) & - & v(x))' & = & u'(x)-v'(x),\\ (u(x) & \cdot & v(x))' & = & u'(x)\cdot v(x)+u(x)\cdot v'(x),\\ (u(x) & / & v(x))' & = & \frac{1}{v^2(x)}(u'(x)v(x)-u(x)v'(x))\\ & & & = & \frac{1}{v(x)}(u'(x)-\frac{u(x)}{v(x)}v'(x)).\end{array} \tag{9.2.1}$$

These rules can be used to define an arithmetic for ordered pairs of numbers, similar to complex arithmetic or interval arithmetic. The first component of the pair consists of a function value $u(x_0)$ at a point x_0. The second component consists of the value of the derivative $u'(x_0)$ of the function at the point x_0. For brevity we simply write (u,u') for the pair of numbers. Then the arithmetic for pairs follows immediately from (9.2.1):

$$\begin{array}{rclcl}(u,u') & + & (v,v') & = & (u+v,u'+v'),\\ (u,u') & - & (v,v') & = & (u-v,u'-v'),\\ (u,u') & \cdot & (v,v') & = & (u\cdot v,u'v+uv'),\\ (u,u') & / & (v,v') & = & (u/v,\frac{1}{v}(u'-(\frac{u}{v})v')),\quad v\neq 0.\end{array} \tag{9.2.2}$$

The set of rules (9.2.2) is called differentiation arithmetic. It is an arithmetic which deals just with numbers. The rules (9.2.2) are easily programmable and are executable by a computer. These rules are now used to compute simultaneously the value of a real function and of its derivative at a point x_0. For brevity we call these values the function-derivative-value-pair. Why and how can this computation be done?

A computable real function can be defined by an arithmetic expression in the manner that arithmetic expressions are usually defined in a programming language. Apart from the arithmetic operators $+$, $-$, $\cdot$, and $/$, arithmetic expressions contain only three kinds of operands as basic components. These are constants, variables and certain differentiable elementary functions such as exp, log, sin, cos, or sqr. The derivatives of these functions are well known.

If for a function $f(x)$ a function-derivative-value-pair is to be computed at a point x_0, all basic components of the arithmetic expression of the function are replaced by their particular function-derivative-value-pair by the following rules:

$$\begin{array}{lllll}\text{a constant:} & c & \longrightarrow & (c,0), \\ \text{the variable:} & x_0 & \longrightarrow & (x_0,1), \\ \text{the elementary functions:} & \exp(x_0) & \longrightarrow & (\exp(x_0),\exp(x_0)), \\ & \log(x_0) & \longrightarrow & (\log(x_0),1/x_0), \\ & \sin(x_0) & \longrightarrow & (\sin(x_0),\cos(x_0)), \\ & \cos(x_0) & \longrightarrow & (\cos(x_0),-\sin(x_0)), \\ & \mathrm{sqr}(x_0) & \longrightarrow & (\mathrm{sqr}(x_0),2x_0), \\ & \text{and so on.} & & \end{array} \tag{9.2.3}$$

Now the operations in the expression are applied following the rules (9.2.2) of differentiation arithmetic. The result is the function-derivative-value-pair $(f(x_0), f'(x_0))$ of the function f at the point x_0.

Example 9.2. For the function $f(x) = 25(x-1)/(x^2+1)$ the function value and the value of the first derivative are to be computed at the point $x_0 = 2$. Applying the substitutions (9.2.3) and the rules (9.2.2) we obtain

$$(f(2), f'(2)) = \frac{(25,0)((2,1)-(1,0))}{(2,1)(2,1)+(1,0)} = \frac{(25,0)(1,1)}{(4,4)+(1,0)} = \frac{(25,25)}{(5,4)} = (5,1).$$

Thus $f(2) = 5$ and $f'(2) = 1$.

If in the arithmetic expression for the function $f(x)$ elementary functions occur in composed form, the chain rule has to be applied, for instance

$$\begin{array}{rcl} \exp(u(x_0)) & \longrightarrow & (\exp(u(x_0)), \exp(u(x_0)) \cdot u'(x_0)) = (\exp u, u' \exp u), \\ \sin(u(x_0)) & \longrightarrow & (\sin(u(x_0)), \cos(u(x_0)) \cdot u'(x_0)) = (\sin u, u' \cos u), \end{array}$$

and so on.

Example 9.3. For the function $f(x) = \exp(\sin(x))$ the value and the value of the first derivative are to be computed for $x_0 = \pi$. Applying the above rules we obtain

$$\begin{aligned} (f(\pi), f'(\pi)) &= (\exp(\sin(\pi)), \exp(\sin(\pi)) \cdot \cos(\pi)) \\ &= (\exp(0), -\exp(0)) = (1, -1). \end{aligned}$$

Thus $f(\pi) = 1$ and $f'(\pi) = -1$.

Differentiation arithmetic is often called *automatic differentiation* or *algorithmic differentiation*. All operations are performed on numbers. A computer can easily and safely execute these operations though people cannot.

Automatic differentiation is not restricted to real functions which are defined by an arithmetic expression. Any real algorithm in essence evaluates a real expression or the value of one or several real functions. Substituting for all constants, variables and elementary functions their function-derivative-value-pair, and performing all arithmetic operations by differentiation arithmetic, will compute simultaneously the function-derivative-value-pair of the result. Large program packages have been developed which do just this, in particular for problems in higher dimensions.

Automatic differentiation or differentiation arithmetic simply uses the arithmetic expression or the algorithm for the function. A formal arithmetic expression or algorithm for the derivative does not explicitly occur. Of course an arithmetic expression or algorithm for the derivative is evaluated indirectly. However, this expression remains hidden. It is evaluated by the rules of differentiation arithmetic. Similarly if differentiation arithmetic is performed for a fixed interval X_0 instead of for a real point x_0, an enclosure of the range of function values and an enclosure of the range of values of the derivative over that interval X_0 are computed simultaneously. Thus, for instance, neither the Newton method nor the interval Newton method requires that the user provides a formal expression for the derivatives. The derivative or an enclosure for it is computed just by use of the expression for the function itself.

Automatic differentiation allows many generalizations which altogether would fill a thick book. We mention only a few of these.

If the value or an enclosure of the second derivative is needed one would use triples instead of pairs and extend the rules (9.2.2) for the third component by corresponding rules: $u'' + v'', u'' - v'', uv'' + 2u'v' + vu''$, and so on. In the arithmetic expression a constant c would now have to be replaced by the triple $(c, 0, 0)$, the variable x by $(x, 1, 0)$ and the elementary functions also by a triple with the second derivative as the third component.

Another generalization is *Taylor arithmetic*. It works with tuples where the first component represents the function value and the following components represent the successive Taylor coefficients. The remainder term of an integration routine for a definite integral or for an initial value problem of an ordinary differential equation usually contains a derivative of higher order. *Interval Taylor arithmetic* can be used to compute a safe enclosure of the remainder term over an interval. This enclosure can serve as an indicator for automatic step size control.

The following formulas describe the trapezoidal rule process if a constant step size is used:

$$\begin{aligned}
\int_{x_i}^{x_{i+1}} f(x)dx &= \frac{h}{2}(f(x_i) + f(x_{i+1})) - \frac{h^3}{12} f''(\xi_i), \quad \xi_i \in [x_i, x_{i+1}]. \\
\int_a^b f(x)dx &= h\sum_{i=0}^{n}{}' f(x_i) - \frac{h^3}{12}\sum_{i=0}^{n-1} f''(\xi_i), \quad \xi_i \in [x_i, x_{i+1}]. \\
\int_a^b f(x)dx &\in h\sum_{i=0}^{n}{}' f(x_i) - \frac{h^3}{12}\sum_{i=0}^{n-1} f''([x_i, x_{i+1}]).
\end{aligned}$$

In these formulas a dash after the sigma symbol indicates as usual that the first and the last summand have to be halved.

The following two definite integrals have been computed with Romberg's extrapolation method:

$$f_1(x) = 2xe^{x^2} \sin(e^{x^2}),$$

$$I_1 = \int_0^2 f_1(x)dx,$$

and

$$
\begin{aligned}
f_2(x) &= \frac{1}{0.1^2+(3x-1)^2} - \frac{1}{0.1^2+(3x-4)^2} \\
&\quad + \frac{1}{0.1^2+(3x-7)^2} - \frac{1}{0.1^2+(3x-10)^2}, \\
I_2 &= \int_0^4 f_2(x)dx.
\end{aligned}
$$

In this case the error term is expressed by the Euler–MacLaurin sum formula. An adaptive step size control by the error term concentrates the work to those areas where the function is steep or where it oscillates severely. This reduces the execution time.

The following bounds are obtained in double precision floating-point arithmetic:

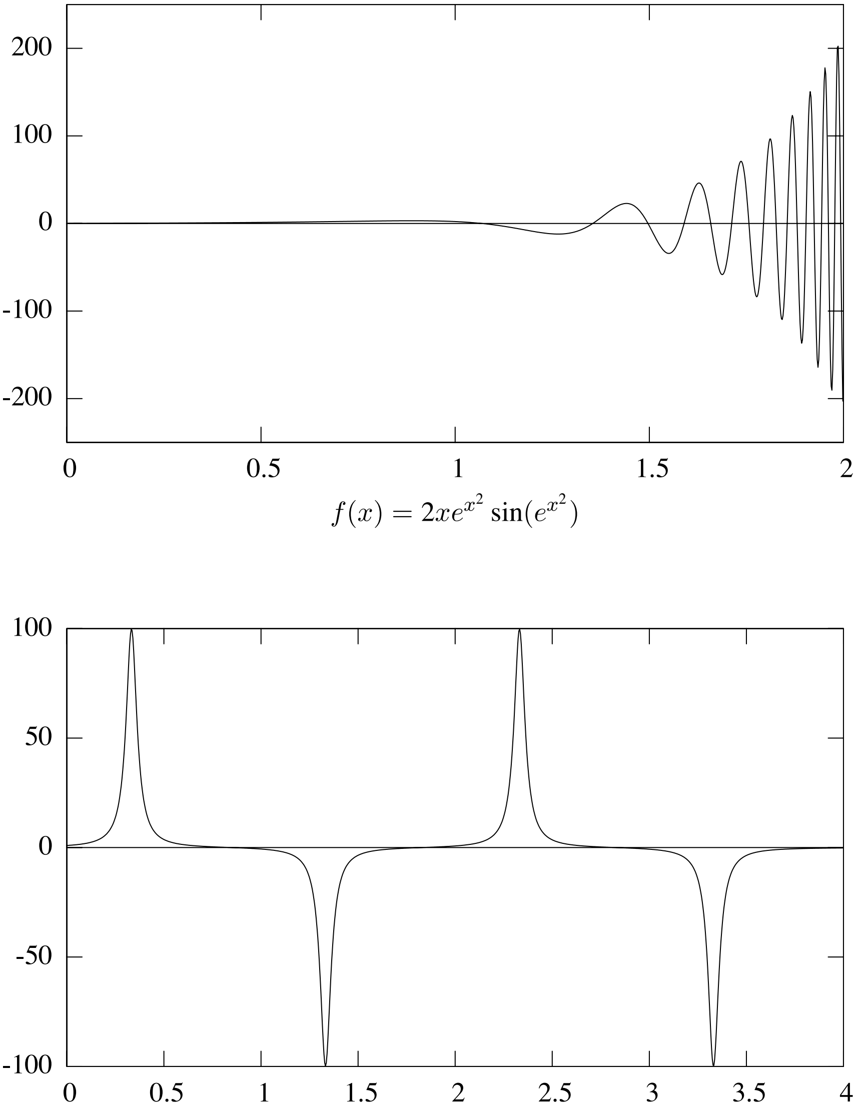
$$
\begin{aligned}
I_1 &\in \ 0.910964039265932^9_8, \\
I_2 &\in -0.151963942232930^6_5.
\end{aligned}
$$

See [5, 277, 332, 573, 651, 653].

Both integrals are difficult to compute with conventional approximate methods. Estimation of the error term for automatic step size control is difficult with traditional floating-point arithmetic. More details can be found in the literature [5, 170, 171, 172, 173, 277, 278, 279, 280, 476, 477, 478, 479, 480, 482, 570, 571, 572, 573].

In [388] R. Lohner has developed a program package AWA which computes continuous upper and lower bounds for the solution of initial value problems of nonlinear systems of differential equations. The package carries its own step size control. It can be applied to boundary and eigenvalue problems of ordinary differential equations as well. In these cases the existence and uniqueness of the solution are not known a priori. These properties are automatically verified by the enclosure algorithm, i.e., by the computer. We briefly sketch the method:

It considers nonlinear systems of ordinary differential equations of the first order. The right hand side of the differential equation is developed into a Taylor polynomial with remainder term by automatic differentiation. Now the remainder term is evaluated for an appropriate step size. Since an enclosure of the solution has to be computed, the computation has to be done in interval arithmetic. In contrast to numerical quadrature, for a differential equation, the remainder term contains the unknown function as well as the independent variable. To compute an enclosure of the remainder term in interval arithmetic, therefore, one first needs a safe enclosure of the unknown function for the particular step of integration. This seems to be a vicious circle. Yet the circle can be opened. First a rough enclosure of the unknown function in the integration interval is computed. For that purpose the differential equation is transformed into an equivalent integral equation. Now the integrand contains the unknown function. In order to enclose the integral within safe bounds by interval arithmetic one



$f(x) = 2xe^{x^2}\sin(e^{x^2})$

$f(x) = \frac{1}{a^2+(3x-1)^2} - \frac{1}{a^2+(3x-4)^2} + \frac{1}{a^2+(3x-7)^2} - \frac{1}{a^2+(3x-10)^2}, \quad a = 0.1$

Figure 9.5. Verified Romberg integration.

again needs an enclosure of the unknown function in the whole integration interval. In the simplest case such an enclosure is obtained by estimation. Since the differential equation fulfils a Lipschitz condition there is always such an enclosure. To keep this estimated enclosure small it may be that the step size has to be reduced. This first, still estimated initial enclosure is now made safe by Banach's fixed-point principle. This means that with this first initial enclosure the right hand side of the integral equation is evaluated in interval arithmetic and it is checked for the expected enclosure. If the check fails the step size has to be reduced and/or the estimated initial enclosure has to be changed. As soon as an enclosure is obtained one has a safe initial enclosure of the unknown solution in the integration interval. With it the remainder term of the Taylor polynomial can be evaluated. The remainder term is now small of higher order. So a much better enclosure of the unknown solution in the integration interval is obtained by a polynomial with interval coefficients. Continuation of this method over several such steps causes the well-known wrapping effect. It is controlled by an intricate use of local coordinates.

Boundary and eigenvalue problems of ordinary differential equations are transformed into initial value problems by shooting methods. Then the algorithm also proves existence and uniqueness of the solution.

These methods for validated computation of the solution of ordinary differential equations are, in general, more expensive in computing time than ordinary approximating methods. However, the additional cost is justified and often more than compensated for overall. A single run on the computer suffices to find a safe solution. Typical trial and error runs to verify the solution are not needed. In particular, critical situations can more easily be detected and analysed. The method has successfully been used to detect and enclose periodic solutions of differential equations, and to prove the existence and uniqueness of the solution. It even has been successfully applied to chaotic solutions of differential equations.

Lohner's AWA program package can be obtained from:
`http://webserver.iam.uni-karlsruhe.de/awa/`.
See also [158, 332, 387, 388, 391, 437].

Another problem class where automatic differentiation has been very successfully applied is global optimization. Global optimization has already been considered in Section 9.1.4. The basic strategy there was subdivision of the given domain into subintervals and elimination of those subintervals which cannot contain the global minimum. Interval evaluation of the expression for the function for an interval delivers a superset of the range of function values in that interval. Those subintervals for which the lower bound of the interval evaluation of the function is greater than a safe function value can be ignored as they cannot contain the global minimum.

If the given function is continuously twice differentiable, interval evaluations of derivatives also can be used to eliminate subintervals from further treatment. Again, we consider here only the one dimensional case.

If the first derivative of the function is positive or negative across the entire interval, the function is monotone in that interval and it cannot have a global minimum within that interval. Automatic differentiation of the function for any subinterval delivers an enclosure (an overestimation) of the range of values of the derivative of the function in that subinterval. If this enclosure does not contain zero, the function is necessarily monotone in that subinterval, which can therefore be ignored.

Also enclosures of the second derivative of the function in a subinterval can be used to exclude subintervals from further treatment. An enclosure of the second derivative of the function in an interval also can easily be obtained by automatic differentiation. If the second derivative of the function is negative across the entire interval, the function cannot have a minimum in that interval.

So if the interval evaluation of the second derivative of the function in a subinterval delivers a negative interval, the function cannot have a minimum in that subinterval, which can therefore be ignored.

Without going into further detail, we simply aver that Newton's method can also be used to exclude subintervals from futher treatment. Corresponding methods, algorithms and programs are also available for problems of more than one dimension.

Basic to all these methods is the fundamental property of interval arithmetic: Evaluation of a function or derivative over an interval delivers an enclosure of its values over a continuum of points, even of points which are not representable on the computer.

In complex, rational, matrix or vector, interval, differentiation and Taylor arithmetic and the like, the arithmetic itself is predefined and can be hidden in the runtime system of the compiler. The user calls the arithmetic operations by the usual operator symbols. The desired arithmetic is activated by type specification of the operands.

As an example the PASCAL-XSC program shown in Figure 9.6 computes and prints enclosures of the 36th and the 40th Taylor coefficient of the function

$$f(x) = \exp\left(\frac{5000}{\sin(11 + (x/100)^2) + 30}\right)$$

over the interval $a = [1.001, 1.005]$.

First the interval a is read. Then it is expanded into the 41-tuple of its Taylor coefficients $(a, 1, 0, 0, \ldots, 0)$ which is kept in b. Then the expression for $f(x)$ is evaluated in interval Taylor arithmetic and enclosures of the 36th and the 40th Taylor coefficient over the interval a are printed.

Automatic differentiation develops its full power in differentiable functions of several real variables. For instance, values or enclosures of the gradient

$$\text{grad} f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n})$$

of a function $f : \mathbb{R}^n \to \mathbb{R}$ or the Jacobian or Hessian matrix can be computed directly from the expression for the function f. No formal expressions for the derivatives are

```
program sample;
use itaylor;
function f(x:  itaylor):  itaylor[lb(x)..ub(x)];
begin f := exp(5000/(sin(11+sqr(x/100))+30));
end;
var a:  interval; b, fb:  itaylor[0..40];
begin
     read(a);
     expand(a,b);
     fb := f(b);
     writeln ('36th Taylor coefficient:  ', fb[36]);
     writeln ('40th Taylor coefficient:  ', fb[40]);
end.
Test results for a = [1.001, 1.005]
36th Taylor coefficient: [-2.4139E+002, -2.4137E+002]
40th Taylor coefficient: [ 1.0759E-006, 1.0760E-006]
```

Figure 9.6. Computation of enclosures of Taylor coefficients.

needed. A special mode, the so-called reverse mode, allows a considerable acceleration for many algorithms of automatic differentiation. In the particular case of the computation of the gradient the following inequality can be shown to hold:

$$A(f, \bigtriangledown f) \leq 5A(f).$$

Here $A(f, \bigtriangledown f)$ denotes the number of operations for the computation of the gradient including the function evaluation, and $A(f)$ the number of operations for the function evaluation, i.e., the number $A(f, \bigtriangledown f)$ of operations for the computation of the gradient and the function differs from the number $A(f)$ of operations for the function evaluation only by a constant factor 5. It is independent of n! For more details see [170, 171, 172, 173, 174, 476, 477, 478, 479, 480, 481].

9.3 The Interval Newton Method

Traditionally Newton's method is used to compute an approximation of a zero of a nonlinear real function $f(x)$, i.e., to compute a solution of the equation

$$f(x) = 0. \tag{9.3.1}$$

The method approximates the function $f(x)$ in the neighborhood of an initial value x_0 by the linear function (the tangent)

$$t(x) = f(x_0) + f'(x_0)(x - x_0) \tag{9.3.2}$$

the zero of which can easily be calculated by

$$x_1 := x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{9.3.3}$$

x_1 is used as new approximation for the zero of (9.3.1). Continuation of this method leads to the general iteration scheme:

$$x_{\nu+1} := x_\nu - \frac{f(x_\nu)}{f'(x_\nu)}, \quad \nu = 0, 1, 2, \dots . \tag{9.3.4}$$

It is well known that if $f(x)$ has a single zero x^* in an interval X and $f(x)$ is twice continuously differentiable, then the sequence

$$x_0, x_1, x_2, \dots, x_\nu, \dots$$

converges quadratically towards x^* if x_0 is sufficiently close to x^*. If the latter condition does not hold the method may well fail.

The interval version of Newton's method computes an enclosure of the zero x^* of a continuously differentiable function $f(x)$ in the interval X by the following iteration scheme:

$$X_{\nu+1} := (m(X_\nu) - \frac{f(m(X_\nu))}{F'(X_\nu)}) \cap X_\nu, \quad \nu = 0, 1, 2, \dots, \tag{9.3.5}$$

with $X_0 = X$. Here $F'(X_\nu)$ is the interval evaluation of the first derivative $f'(x)$ of f over the interval X_ν and $m(X_\nu)$ is the midpoint of the interval X_ν. Instead of $m(X_\nu)$ another point within X_ν could be chosen. In conventional interval arithmetic, where division A/B is only defined if $0 \notin B$, the interval Newton method can only be applied if $0 \notin F'(X_0)$. This guarantees that $f(x)$ has only a single zero in X_0.

In contrast to (9.3.4), the method (9.3.5) must always converge. Because of the intersection with X_ν the sequence

$$X_0 \supseteq X_1 \supseteq X_2 \supseteq \dots \tag{9.3.6}$$

is bounded. It can be shown that under natural conditions on the function f the sequence converges quadratically to x^* [29, 444].

The operator

$$N(X) := x - \frac{f(x)}{F'(X)}, \quad x \in X \in I\mathbb{R}, \tag{9.3.7}$$

is called the *interval Newton operator*. It has the following properties:

I. If $N(X) \subseteq X$, then $f(x)$ has exactly one zero x^* in X.

II. If $N(X) \cap X = \varnothing$, then $f(x)$ has no zero in X.

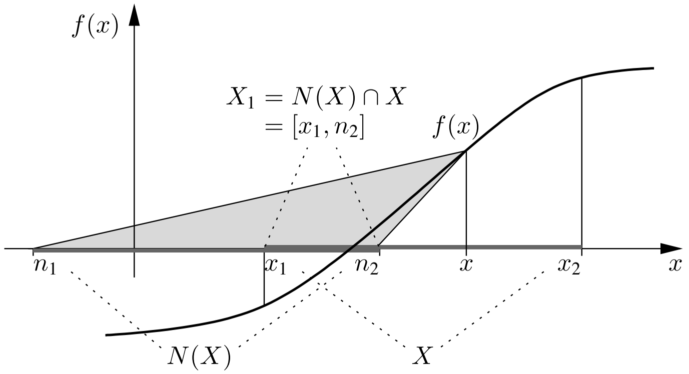


Figure 9.7. Geometric interpretation of the interval Newton method.

Thus, $N(X)$ can be used to prove the existence or absence of a zero x^* of $f(x)$ in X. Since if there is a zero x^* in X the sequence (9.3.5), (9.3.6) converges, if there is no such zero $N(X) \cap X = \varnothing$ must occur in (9.3.6).

The interval version of Newton's method (9.3.5) can also be derived via the mean value theorem. If $f(x)$ is continuously differentiable and has a single zero x^* in the interval X, and $f'(x) \neq 0$ for all $x \in X$, then

$$f(x) = f(x^*) + f'(\xi)(x - x^*) \text{ for all } x \in X \text{ and some } \xi \text{ between } x \text{ and } x^*.$$

Since $f(x^*) = 0$ and $f'(\xi) \neq 0$ this leads to

$$x^* = x - \frac{f(x)}{f'(\xi)}.$$

If $F'(X)$ denotes the interval evaluation of $f'(x)$ over the interval X, we have $f'(\xi) \in F'(X)$ and therefore

$$x^* = x - \frac{f(x)}{f'(\xi)} \in x - \frac{f(x)}{F'(X)} = N(X) \text{ for all } x \in X,$$

i.e., $x^* \in N(X)$ and thus

$$x^* \in (x - \frac{f(x)}{F'(X)}) \cap X = N(X) \cap X.$$

Now we obtain by setting $X_0 := X$ and $x = m(X_0)$

$$X_1 := (m(X_0) - \frac{f(m(X_0))}{F'(X_0)}) \cap X_0,$$

and by continuation (9.3.5).

In close similarity to the conventional Newton method the interval Newton method also allows some geometric interpretation. For that purpose let be $X = [x_1, x_2]$ and $N(X) = [n_1, n_2]$. $F'(X)$ is the interval evaluation of $f'(x)$ over the interval X. As such it is a superset of all the slopes of tangents that can occur in X. The conventional Newton method (9.3.3) computes the zero of the tangent of $f(x)$ in $(x_0, f(x_0))$. Similarly $N(X)$ is the interval of all zeros of straight lines through $(x, f(x))$ with slopes within $F'(X)$, see Figure 9.7. Of course, $f'(x) \in F'(X)$.

The straight line through $f(x)$ with the least slope within $F'(X)$ cuts the real axis at n_1, and the one with the greatest slope at n_2. Thus the Interval Newton Operator $N(X)$ computes the interval $[n_1, n_2]$ which in the sketch of Figure 9.7 is situated on the left hand side of x. The intersection of $N(X)$ with X then delivers the new interval X_1. In the example in Figure 9.7, $X_1 = [x_1, n_2]$.

Newton's method allows some visual interpretation. From the point $(x, f(x))$ the conventional Newton method sends a ray of light along the tangent. The search is continued at the intersection of this ray with the x-axis. The interval Newton method sends a set of rays like a floodlight from the point $(x, f(x))$ to the x-axis. This set includes the directions of all tangents that occur in the entire interval X. The interval $N(X)$ comprises all cuts of these rays with the x-axis.

It is a fascinating discovery that the interval Newton method can be extended so that it can be used to compute all zeros of a real function in a given interval. The basic idea of this extension is quite old [12]. Many scientists have worked on details of how to use this method, of how to define the necessary arithmetic operations, and of how to bring them to the computer. But inconsistencies have occurred again and again. However, understanding has now reached a point which allows a consistent realization of the method and of the necessary arithmetic. Newton's method reaches its ultimate elegance and power in the *extended interval Newton method* which we are now going to discuss.

9.4 The Extended Interval Newton Method

The extended interval Newton method can be used to compute enclosures of all the zeros of a continuously differentiable function $f(x)$ in a given interval X. The iteration scheme is identical to the one defined by (9.3.5) in Section 9.3:

$$X_{\nu+1} := (m(X_\nu) - \frac{f(m(X_\nu))}{F'(X_\nu)}) \cap X_\nu = N(X_\nu) \cap X_\nu, \quad \nu = 0, 1, 2, \ldots,$$

with $X_0 := X$. Here again $F'(X_\nu)$ is the interval evaluation of the first derivative $f'(x)$ of the function f over the interval X_ν, and $m(X_\nu)$ is any point within X_ν, the midpoint for example. If $f(x)$ has more than one zero in X, then the derivative $f'(x)$ has at least one zero (horizontal tangent of $f(x)$) in X also, and the interval evaluation $F'(X)$ of $f'(x)$ contains zero. Thus extended interval arithmetic has to be used to execute the Newton operator

$$N(X) = x - \frac{f(x)}{F'(X)}, \text{ with } x \in X.$$

As shown by Tables 4.8 and 4.9 the result is no longer an interval of $I\mathbb{R}$. It is an element of the power set $\mathbb{PR}$ which, in general, stretches continuously to $-\infty$ or $+\infty$ or both. The intersection $N(X) \cap X$ with the finite interval X then produces a finite set again. It may consist of a finite interval of $I\mathbb{R}$, or of two separate such intervals, or of the empty set. These sets are now the starting values for the next iteration. This means that where two separate intervals have occurred, the iteration has to be continued with two different starting values. This situation can occur repeatedly. On a sequential computer where only one iteration can be performed at a time all intervals which are not yet dealt with are collected in a list. This list then is treated sequentially. If more than one processor is available different subintervals can be dealt with in parallel.

Again, we illustrate this process by a simple example. The starting interval is denoted by $X = [x_1, x_2]$ and the result of the Newton operator by $N = [n_1, n_2]$. See Figure 9.8.

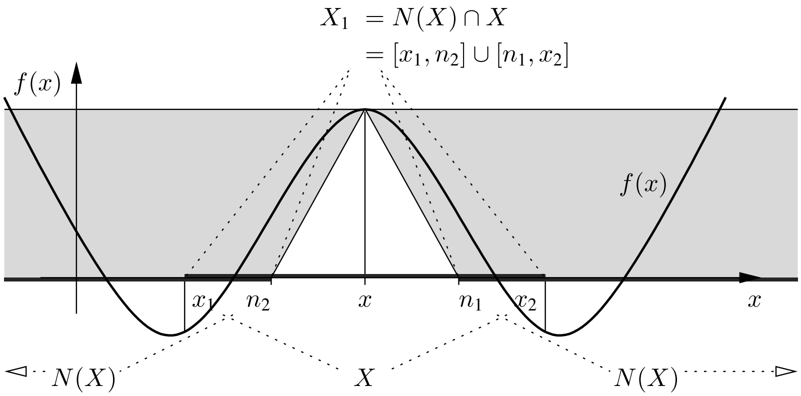


Figure 9.8. Geometric interpretation of the extended interval Newton method.

Now $F'(X)$ is again a superset of all slopes of tangents of $f(x)$ in the interval $X = [x_1, x_2]$. But now $0 \in F'(X)$. $N(X)$ again is the set of zeros of straight lines through $(x, f(x))$ with slopes within $F'(X)$. Let be $F'(X) = [s_1, s_2]$. Since $0 \in F'(X)$ we have $s_1 \leq 0$ and $s_2 \geq 0$. The straight lines through $(x, f(x))$ with the slopes s_1 and s_2 cut the real axis at n_1 and n_2. Thus the Newton operator produces the set

$$N(X) = (-\infty, n_2] \cup [n_1, +\infty).$$

The floodlight is now shining in two directions. Intersection with the original set X (the former iterate) delivers the set

$$X_1 = N(X) \cap X = [x_1, n_2] \cup [n_1, x_2]$$

consisting of two finite intervals of $I\mathbb{R}$. From this point the iteration has to be continued with the two starting intervals $[x_1, n_2]$ and $[n_1, x_2]$.

9.5 Verified Solution of Systems of Linear Equations

Systems of linear equations play a central role in numerical analysis. Such systems are often very large. An avalanche of numbers is produced if a linear system of equations with one million unknowns is "solved" on a computer by a direct method. Every floating-point operation is potentially in error. This brings to the fore the question of whether the computed result really solves the problem or of how many of the computed digits are correct. Based on rigorous mathematics a validated inclusion of the solution can be computed which answers the question.

Various methods of so-called automatic result verification or validation by the computer for systems of linear equations have been developed over recent years. Here we only discuss a few basic ideas and techniques. Interval arithmetic and mathematical fixed-point theorems are applied. Basic is the use of the Brouwer fixed-point theorem which is now stated.

Theorem 9.4. *A continuous mapping $g : \mathbb{R}^n \to \mathbb{R}^n$ which maps a non-empty, convex, closed and bounded set $X \subseteq \mathbb{R}^n$ into itself, i.e.,*

$$\bigwedge_{x \in X} g(x) \in X,$$

has at least one fixed-point $x^* \in X$. ■

A few concepts of interval analysis are needed. For an interval $X = [x_1, x_2] \in I\mathbb{R}$ the diameter $d(X)$ and the absolute value $|X|$ are defined by

$$d(X) := x_2 - x_1 \qquad \text{(diameter)},$$

$$|X| := \max\{|x| \mid x \in X\} \qquad \text{(absolute value)}.$$

The distance between two intervals $X = [x_1, x_2]$, $Y = [y_1, y_2] \in I\mathbb{R}$ is defined by the Hausdorff metric

$$q(X, Y) := \max\{|x_1 - y_1|, |x_2 - y_2|\} \qquad \text{(Hausdorff metric)}.$$

Convergence of a sequence of intervals $X_i = [x_{i1}, x_{i2}] \in I\mathbb{R}$, $i = 1, 2, 3, \ldots$, is defined as usual in a metric space:

$$\lim_{i\to\infty} X_i = X = [x_1, x_2] :\Leftrightarrow \lim_{i\to\infty} q(X_i, X) = 0$$
$$:\Leftrightarrow \left(\lim_{i\to\infty} x_{i1} = x_1 \wedge \lim_{i\to\infty} x_{i2} = x_2\right).$$

For vectors and matrices the diameter, absolute value and distance are defined componentwise. An interval vector $X \in V_n I\mathbb{R}$ is an n-dimensional box, an interval in each coordinate direction.

Now we consider a system of linear equations

$$Ax = b \text{ with } A \in \mathbb{R}^{n\times n},\ b \in \mathbb{R}^n. \tag{9.5.1}$$

A solution $x^* \in \mathbb{R}^n$ is a zero of the function

$$g(x) := Ax - b. \tag{9.5.2}$$

Linear systems of equations are often solved iteratively. The following theorem gives a necessary and sufficient criterion for convergence of an iterative scheme using intervals. The proof is given in [28, 29].

Theorem 9.5. *An iterative method*

$$X_{i+1} := BX_i + c,\ \ B \in \mathbb{R}^{n\times n},\ \ c \in \mathbb{R}^n,\ \ X_i \in V_n I\mathbb{R},\ \ i = 0, 1, 2, \ldots,$$

converges to the unique solution of the equation

$$x = Bx + c$$

for every $X_0 \in V_n I\mathbb{R}$ if and only if $\rho(|B|) < 1$. Here $\rho(|B|)$ denotes the spectral radius of the matrix $|B|$. ■

Now we derive an iterative method for the solution of (9.5.1). Newton's method applied to (9.5.2) would lead to

$$x_{i+1} := x_i - A^{-1}(Ax_i - b),$$

where A^{-1} denotes the inverse of the matrix A. Since A^{-1} is unknown we replace it by a suitably chosen matrix R, an approximate inverse $R \approx A^{-1}$ of A, for instance. This leads to

$$x_{i+1} := (I - RA)x_i + Rb.$$

In the relevant literature the operator

$$g(X) := (I - RA)X + Rb, \quad X \in V_n I\mathbb{R}, \tag{9.5.3}$$

is called the *Krawczyk-operator*, see [336].

In the expression of the right hand side of the assignment in (9.5.3) all operations are evaluated in interval arithmetic. The variable X occurs only once in the expression. Thus no overestimation appears if it is evaluated for an interval vector X.

By Theorem 9.5 an iterative method with the Krawczyk-operator

$$X_{i+1} := (I - RA)X_i + Rb, \quad X_0 \in V_n I\mathbb{R}, \tag{9.5.4}$$

converges for every initial interval vector X_0 to the unique solution of (9.5.1) if and only if $\rho(|I - RA|) < 1$.

The necessary and sufficient criterion for convergence of (9.5.4) is hard to check. Computation of the eigenvalues of a matrix is of complexity similar to that of the solution of the linear system. The following theorem gives criteria for a convergence of (9.5.4) which can be checked very easily. In this theorem a comparison relation $x < y$ for vectors $x, y \in \mathbb{R}^n$ is used. It is defined componentwise.

Theorem 9.6. *(a) The iterative method (9.5.4) converges to a unique solution x^* of (9.5.1) if for an initial interval vector $X = X_0 \in V_n I\mathbb{R}$ the Krawczyk-operator strictly reduces the diameter*

$$d(X_1) := d(g(X)) < d(X). \tag{9.5.5}$$

Then the matrices A and R in (9.5.3) are not singular, i.e., the linear system (9.5.1) has a unique solution.

(b) If in addition to (9.5.5) the Krawczyk-operator maps the interval X_0 into itself, i.e., if

$$X_1 := g(X_0) \subseteq X_0 \tag{9.5.6}$$

then $x^* \in X_0$ and the iteration (9.5.4) converges to x^* by a nested sequence

$$X_0 \supseteq X_1 \supseteq X_2 \supseteq \dots$$

with $x^* \in X_i$ for all $i = 0, 1, 2, \dots$.

Proof. (a) $d(g(X)) = d((I - RA)X + Rb) = d((I - RA)X) = |I - RA|d(X)$

$$\underset{(9.5.5)}{\Rightarrow} |I - RA|d(X) < d(X). \tag{9.5.7}$$

X is an interval vector of dimension n, i.e., an n-dimensional box. By (9.5.5) the components d_i of the diameter vector $d(X)$ are all positive, $d_i > 0, i = 1(1)n$. Multiplication of $d(X)$ by the diagonal matrix

$$D := \operatorname{diag}(1/d_i) := \begin{pmatrix} 1/d_1 & 0 & \cdots & 0 \\ 0 & 1/d_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1/d_n \end{pmatrix}$$

leads to the vector $u := D \cdot d(X)$, all components of which are 1. By expansion of the left hand side of (9.5.7) we now obtain the inequality

$$D \cdot |I - RA| \cdot D^{-1} u < u. \tag{9.5.8}$$

The components of the vector on the left hand side of the inequality (9.5.8) are the row sums of the matrix $D|I - RA|D^{-1}$. Their maximum is a norm, i.e., we have

$$||D \cdot |I - RA| \cdot D^{-1}|| < 1,$$

and therefore

$$\rho(D \cdot |I - RA| \cdot D^{-1}) < 1.$$

Since similar matrices have the same eigenvalues we obtain

$$\rho(|I - RA|) < 1$$

and by the theorem of *Perron and Frobenius* [597, 598]

$$\rho(I - RA) \leq \rho(|I - RA|) < 1.$$

Therefore the matrix $I - (I - RA) = RA$ is not singular, nor are A and R. The iteration (9.5.4) converges to the unique solution of (9.5.1).

(b) By the Brouwer fixed-point theorem the mapping $g(x)$ has a fixed-point x^* in X_0. The fixed-point is unique by (a) and the iteration (9.5.4) converges to it. By (9.5.4) $x^* \in X_i$, for all $i = 0, 1, 2, \ldots$: With (9.5.6) we obtain by induction and the inclusion isotonicity of interval arithmetic

$$X_i \subseteq X_{i-1} \Rightarrow X_{i+1} := g(X_i) \subseteq g(X_{i-1}) = X_i. \qquad \blacksquare$$

The condition (9.5.5) of Theorem 9.6(a) can easily be checked by the computer. However, under this condition the solution x^* of the linear system can lie outside the iterates X_i produced by the iteration (9.5.4). In this case both bounds of the iterates X_i converge from outside to the solution x^*. An enclosure $x^* \in X_i$ is not obtained.

From the practical point of view it is desirable that the iteration delivers bounds, i.e., an enclosure of the solution x^*. This is achieved by the additional condition (9.5.6) of Theorem 9.6(b).

In the relevant literature instead of (9.5.5), $d(g(X)) < d(X)$, and (9.5.6), $g(X) \subseteq X$, the criterion

$$g(X) \subseteq \overset{\circ}{X} \tag{9.5.9}$$

is often given and used. In (9.5.9) $\overset{\circ}{X}$ denotes the interior of X (the bounds are excluded), [503]. (9.5.5) and (9.5.6) hold under the condition (9.5.9). The opposite, however, is not true.

(9.5.9) guarantees convergence of the iteration and it delivers an enclosure of the solution x^*. (9.5.9) can easily be checked on the computer. For two interval vectors $X = [x_1, x_2]$ and $Y = [y_1, y_2]$ we have

$$X \subseteq \overset{\circ}{Y} \quad \Leftrightarrow \quad y_1 < x_1 \wedge x_2 < y_2.$$

The criteria (9.5.6) and (9.5.9) can only apply if $x^* \in X_0$. This property can be achieved by a process which is known as *ϵ-inflation*.

First an approximation $\tilde{x}$ for the solution x^* is computed by some ordinary method, for example by Gaussian elimination. This approximation is expanded by offsetting it by a small value ϵ in each component direction. This results in an n-dimensional cube which has the computed approximation as its midpoint. This cube is now chosen as the starting value X_0 for the iteration (9.5.4). If R is a good approximation of A^{-1} then the criterion for convergence $\rho(|I - RA|) < 1$ usually holds. Then in most cases in practice the condition (9.5.6) or (9.5.9) holds already after one iteration. If it does not occur another ϵ-inflation is applied. Only rarely are more iterations needed to verify (9.5.6) or (9.5.9), and then only two or three. Rump has shown in [515] that the iteration with ϵ-inflation reaches the condition (9.5.9) if and only if $\rho(|I - RA|) < 1$.

The methods that just have been described can be further improved. Let us assume that an approximate solution $\tilde{x}$ of the linear system has already been computed by a favorite algorithm. Then an interval inclusion E of the error $e = x^* - \tilde{x}$ usually leads to a very good inclusion of the solution x^*:

$$e = x^* - \tilde{x} \in E \Rightarrow x^* \in \tilde{x} + E. \tag{9.5.10}$$

It is well known that the error e is a solution of a linear system with the same matrix and the defect or residual r of the approximation $\tilde{x}$ as the right hand side:

$$Ae = r. \tag{9.5.11}$$

Here the residual r is defined by

$$r := b - A\tilde{x}. \tag{9.5.12}$$

Thus an inclusion of the error can be obtained by the interval iteration scheme:

$$E_{i+1} := (I - RA)E_i + Rr. \tag{9.5.13}$$

See [503].

When used on the computer (9.5.12) and (9.5.13) are very sensitive to rounding. If $\tilde{x}$ in (9.5.12) is already a good approximation of the solution x^*, then $A\tilde{x}$ is close to b and the subtraction $b - A\tilde{x}$ causes cancellation in conventional floating-point arithmetic. For an imprecisely known r, determination of the error e from (9.5.11) could be worthless.

Similarly, if R in (9.5.13) and in (9.5.4) is a good approximation of A^{-1}, then RA is close to I and the subtraction $I-RA$ in (9.5.13) causes cancellation in conventional floating-point arithmetic. In such a case the iteration matrix in (9.5.13) is known only approximately so that the computed result of the iteration (9.5.13) and (9.5.4) could be worthless.

The situation is essentially improved if the expressions $b - A\tilde{x}$ in the residual (9.5.12) and $I - RA$ in the iteration (9.5.13) and (9.5.4) are computed to full accuracy by the exact scalar product. The result is then rounded to the least including interval. This makes the residual correction process work very well.

On the computer the iteration (9.5.13) is then carried out by the following algorithm:

```
(i)   Compute an approximate inverse $R$ of $A$ using your favorite algorithm.

(ii)  $\tilde{X} := \Diamond(R \cdot b)$
      $B := \Diamond(I - R \cdot A)$
      $r := \Diamond(b - A \cdot \tilde{X})$
      $E := \Diamond(R \cdot r)$
      $Z := E$
      $i := 0$

(iii) repeat $Y := E \Diamond [1-\epsilon, 1+\epsilon]$
             $i := i + 1$
             $E := \Diamond(B \cdot Y + Z)$
      until $(E \subseteq \mathring{Y})$ or $i = 10$

(iv)  if $(E \subseteq \mathring{Y})$ then {It has been verified that a unique solution
                              $x^*$ of $Ax = b$ exists and $x^* \in \tilde{X} + E$}
                    else {Verification failed, $A$ is probably ill-conditioned}.
```

In the *else* case a more powerful algorithm could be called (see *Rump-operator*). In the steps 2 and 3 of the algorithm above all operations $+, -, \cdot$ are to be executed exactly as real number operations.

What has been described so far is known as result verification. For an approximation $\tilde{x}$ of the solution x^* of a linear system an enclosure of the solution is computed and existence and uniqueness of the solution within the computed bounds are verified. In the algorithm all operations and expressions are to be executed exactly in complete arithmetic.

It is possible that the verification step fails to produce an enclosure after let's say 10 iterations with ϵ-inflation. This can happen in the case of an extremely ill conditioned linear system. Then the matrix R in (9.5.4) and in (9.5.13) may not be good enough to allow a successful iteration. This situation is recognized by the computer in step 4 of the algorithm above. The algorithm then automatically calls a more powerful operator which in almost all cases leads to a successful inclusion. This procedure has been proposed, implemented, and successfully applied by S. M. Rump, [503]. We briefly sketch it here.

For the Krawczyk-operator the matrix RA has already been computed. Now its inverse $(RA)^{-1}$ is computed in floating-point arithmetic. Then the product $(RA)^{-1} \cdot R$ is computed with the exact scalar product. Theoretically, if real arithmetic could be used, the multiplication $(RA)^{-1} \cdot R$ would eliminate the matrix R and lead to the ideal situation A^{-1}. But it can be expected that the product $(RA)^{-1} \cdot R$ is closer to A^{-1} than the original matrix R in the Krawczyk-operator. The exact scalar product computes the product $(RA)^{-1} \cdot R$ to full accuracy. From that a portion can be read to two or three or more fold precision as a long real matrix, for instance, $(RA)^{-1} \cdot R \approx R_1 + R_2 + R_3 + R_4$. In many cases the first two approximations $(RA)^{-1} \cdot R \approx R_1 + R_2$ suffice. This leads to the iteration scheme

$$X_{i+1} := (I - R_1 A - R_2 A) X_i + R_1 b + R_2 b.$$

This scheme is called the *Rump method* and the operator

$$h(X) := (I - R_1 A - R_2 A) X + R_1 b + R_2 b$$

or its alternative for the error E is called the *Rump-operator*.

If executed on the computer each component in the expression $I - R_1 A - R_2 A$ is evaluated as a single exact scalar product. The result then is rounded to the least including interval. The expression on the right hand side of the Rump-operator is then evaluated in interval arithmetic.

It is essential to understand that even in the Rump-operator all operations, apart from the exact scalar product and in particular the computation of the inverse $(RA)^{-1}$, are performed in conventional floating-point arithmetic and thus are very fast. If the exact scalar product is implemented in hardware it is faster than a conventional scalar product in floating-point arithmetic.

Rump has applied the method to the Hilbert matrix of dimension 20. Its condition number is of the magnitude 10^{28}. To present the matrix on the computer with no rounding errors the components are multiplied by a common factor to obtain integer entries. Rump has shown in [508] that with an approximation of $(RA)^{-1} \cdot R$ by a two fold sum the inverse of the Hilbert matrix can be computed to least bit accuracy.

In a recent paper by Oishi, Tanabe, Ogita, and Rump [459] the authors prove the convergence of variants of this method even for extremely ill-conditioned problems. They show that matrices with a condition number of about 10^{100} can successfully be inverted if the product $(RA)^{-1}$ is computed to k fold precision (with $k \approx 10$).

The linear system solver discussed in this section including the extension by the Rump-operator is an essential ingredient of many problem solving routines with automatic result verification in numerical analysis. It can be extended to problems with interval coefficients and complex coefficients, [508]. Also, generalizations are available for systems of nonlinear equations, [39, 522].

The whole process described in this section delivers highly accurate bounds for the solution and simultaneously proves the existence and uniqueness of the solution within the computed bounds. Thus the computer is no longer merely a fast calculating tool, but a scientific mathematical instrument. This makes the computer much more user friendly. Of course, often the computer has to do more work to obtain verified results. But the mathematical certainty should be worth it. Computing that is continually and greatly speeded up makes this step necessary. It is the very speed that calls conventional floating-point computing into question.

9.6 Accurate Evaluation of Arithmetic Expressions

Arithmetic expressions are basic ingredients of all numerical software. The need to have mathematical models which approximate the real world leads to more and more complicated formulas. Many of these formulas used in modern engineering are not created by hand but by symbolic manipulations on a computer. Numerical evaluation of these formulas is usually done in floating-point arithmetic. This is a speedy process since floating-point arithmetic is supported by fast hardware. However, the computed result may differ quite a bit from the correct result of the expression without the user being informed of it. It is important, therefore, to have methods which deliver highly accurate bounds for the value of the expression.

In this section we develop a method which evaluates an arithmetic expression with guaranteed high (maximum) accuracy. The method uses fast floating-point arithmetic with directed roundings (interval arithmetic) and a fast and exact scalar product. If all these operations are reasonably supported by the computer's hardware, the computing time of the method is of the same order as that for a conventional evaluation of the expression in floating-point arithmetic. Should the conventional evaluation fail completely additional computing time and storage is needed.

Occasionally a simple method of evaluating an arithmetic expression to high accuracy is recommended. It uses multiple precision interval arithmetic (see the next section). In the first step the expression is evaluated in any fast hardware supported interval arithmetic. If the width of the result is larger than the desired accuracy, the evaluation is repeated in doubled precision, in tripled precision, and so on, until the desired accuracy is obtained.

The method has two disadvantages: Evaluation in a higher precision does not make use of the work that was already done in lower precision, and the speed of the arithmetic decreases drastically with increasing precision multiples.

9.6.1 Complete Expressions

There is a class of arithmetic expressions which can always be evaluated exactly. These expressions have been called *complete expressions* in Section 8.7. Examples of such expressions are given in the previous section where in the case of a system of linear equations the residual $b - A \cdot \tilde{x}$ or the iteration matrix $I - R \cdot A$ had to be computed to full accuracy. Disregarding the usual priority for the operations, every component in these expressions is computed as a single exact scalar product. Thus cancellation of digits is avoided. In Section 8.7 complete expressions are defined on the scalar, vector or matrix level.

We give two other examples for complete expressions. If A and B are matrices, and a, b and c are vectors, then

$$a + A \cdot b + B \cdot c$$

is a complete expression. Or, if A_i and B_i, $i = 1(1)4$, are vectors or matrices, then

$$A_1 \cdot B_1 + A_2 \cdot B_2 + A_3 \cdot B_3 + A_4 \cdot B_4$$

is a complete expression. Sharp or accurate evaluation of a complete expression means that the whole expression, i.e., each component of it, is evaluated as a single exact scalar product.

Sharp evaluation of complete expressions is a basic feature in the programming languages PASCAL-XSC, ACRITH-XSC, and Fortran-XSC. In these languages complete expressions are enclosed in parentheses and prefixed by a #, the sharp symbol. A rounding symbol can follow the sharp symbol. Complete expressions are evaluated exactly by complete arithmetic. The full result is stored if no rounding is specified. It is rounded to real, if the rounding symbol is $<$ (down), $>$ (up), or $*$ (to nearest), and it is rounded to an interval, if the rounding symbol is #.

Accurate evaluation of general arithmetic expressions is performed in three steps. The first step transforms the expression into a simple system of equations. The second step evaluates this system in floating-point arithmetic. The third step then computes sharp bounds for the residual and with this an enclosure of the error. The key operation in the third step is an exact scalar product. If a desired accuracy of the result is not obtained the third step is repeated once or several times.

9.6.2 Accurate Evaluation of Polynomials

The basic ideas for an accurate evaluation of expressions can be extended to polynomials. Evaluating a polynomial can be very delicate, especially in the neighborhood of a zero. Accurate evaluation of a polynomial is thus a crucial task for all zero finders.

We consider the polynomial

$$p(z) = \sum_{i=0}^{n} a_i z^i,$$

where the a_i, $i = 1(1)n$, are given floating-point numbers. The evaluation is done for a floating-point number $z = t$. Horner's scheme leads to:

$$p(t) = (\ldots(a_n t + a_{n-1})t + \ldots + a_1)t + a_0.$$

Now for each step in the *Horner scheme* we introduce an auxiliary variable x_i, $i = n(-1)0$ (starting with $x_n = a_n$):

$$\begin{aligned} x_n &:= a_n, \\ x_{n-1} &:= x_n \cdot t + a_{n-1}, \\ &\vdots \\ x_i &:= x_{i+1} \cdot t + a_i, \\ &\vdots \\ x_0 &:= x_1 \cdot t + a_0. \end{aligned}$$

This is a system of linear equations

$$A \cdot x = b,$$

for the unknowns x_i, $i = 0(1)n$, with a simple matrix A and the right hand side $b = (a_i)$:

$$A = \begin{pmatrix} 1 & 0 & & 0 \\ -t & 1 & & \\ & \ddots & \ddots & \\ 0 & & -t & 1 \end{pmatrix}, \quad x = \begin{pmatrix} x_n \\ x_{n-1} \\ \vdots \\ x_0 \end{pmatrix}, \quad b = \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{pmatrix}.$$

x_0 is the value of the polynomial $p(t)$.

This system could be solved with high accuracy and verification of the result with one of the methods for linear systems discussed in Section 9.5. However, these methods are designed for general matrices and do not take advantage of the simple structure of the present problem. Furthermore, high accuracy is not needed for all components, but only for the last one. Thus it is worthwhile to develop a special method for the present problem.

For systems of linear equations the method of defect correction or iterative refinement works well if a fast and exact multiply and accumulate operation is used to compute critical terms like the defect $b - A \cdot \tilde{x}$ of an approximate solution $\tilde{x}$. So we use this method here also.

As a next step we compute an approximation $x_i^{(0)}$ for each of the auxiliary variables x_i in floating-point arithmetic by forward substitution

$$\begin{aligned} x_n^{(0)} &:= a_n, \\ x_i^{(0)} &:= x_{i+1}^{(0)} \cdot t + a_i, \quad i = n-1(-1)0. \end{aligned}$$

This is just a floating-point evaluation of the polynomial by Horner's scheme. Then $x_0^{(0)} \approx p(t)$.

If x^* denotes the exact solution and $x^{(0)} = (x_i^{(0)})$ the approximate solution of the linear system $Ax = b$, the error is

$$e := x^* - x^{(0)}.$$

The defect or the residual r of $x^{(0)}$ is defined by

$$r := b - A \cdot x^{(0)}.$$

It is well known that the error e is the solution of a system of linear equations with the same matrix A and the residual r as the right hand side:

$$A \cdot e = r.$$

If we compute an enclosure E of the error e, we have an enclosure of the solution x^*:

$$e = x^* - x^{(0)} \in E \Rightarrow x^* \in x^{(0)} + E.$$

With the exact scalar product a sharp enclosure $R^{(1)}$ for the residual is obtained by the complete expression

$$R^{(1)} := \lozenge(b - Ax^{(0)}).$$

Here an accurate evaluation of the residual by the exact scalar product is essential to avoid catastrophic cancellation and overestimation of the residual.

Now a first enclosure $E^{(1)}$ of the error e is obtained as solution of the equation

$$A \cdot E^{(1)} = R^{(1)}.$$

Because of the special form of the matrix A, $E^{(1)}$ can be computed by forward substitution (Horner's scheme) using interval arithmetic. If the components of $R^{(1)}$ and $E^{(1)}$ are denoted by $R_i^{(1)}$, $E_i^{(1)}$, $i = 0(1)n$, respectively: $R^{(1)} = (R_i^{(1)})$, $E^{(1)} = (E_i^{(1)})$, we obtain

$$E_n^{(1)} := R_n^{(1)}, \quad E_i^{(1)} := E_{i+1}^{(1)} \cdot t + R_i^{(1)}, \; i = n - 1(-1)0.$$

By construction (and using the properties of interval arithmetic) we have

$$p(t) \in x_0^{(0)} + E_0^{(1)}.$$

If the diameter of the interval on the right hand side is not sufficiently small, the residual iteration is continued:

Let $x^{(1)} := m(E^{(1)})$ be the midpoint of the error vector $E^{(1)}$. We use the vector $x^{(0)} + x^{(1)}$ as a new approximation of the solution of the linear system.

At the beginning of the $(k+1)$th iteration we have $k+1$ vectors $x^{(0)}, x^{(1)}, \dots, x^{(k)}$, the sum

$$\sum_{j=0}^{k} x^{(j)}$$

of which is an approximate solution of the linear system $Ax = b$. A sharp enclosure $R^{(k+1)}$ of the residual of this approximation is now obtained by the complete expression

$$R^{(k+1)} := \diamondsuit \left(b - A \cdot \sum_{j=1}^{k} x^{(j)} \right) \in I\mathbb{R}^{n+1}.$$

With this enclosure $R^{(k+1)}$ of the residual a corresponding new enclosure of the error is now defined by the linear system

$$A \cdot E^{(k+1)} = R^{(k+1)}.$$

The solution of this system is now computed by forward substitution in interval arithmetic (Horner's scheme). Thus we obtain by construction:

$$p(t) = x_0^* \in \sum_{j=0}^{k} x_0^{(j)} + E_0^{(k+1)}.$$

The iteration is stopped when the desired accuracy is reached. It is shown in [86, 87, 368] that the sequence of interval vectors $E^{(k)}$ converges toward the zero vector as $k \to \infty$, as $\sum_{j=0}^{k} x^{(j)}$ approximates x^*. With this we have

$$p(t) = x_0^* = \lim_{k \to \infty} \left(\sum_{j=0}^{k} x_0^{(j)} \right)$$

as an exact evaluation of the polynomial $p(z)$ at the point $z = t$.

The rate of convergence of the method is linear. It is inversely proportional to the condition number of the matrix A. In fact, the number of residual iterations necessary to achieve maximum accuracy is a rough indicator for the condition number of A.

The method is also discussed and executable programs are given in [205, 206, 207]. See also [319]. We cite a few examples which are given in the literature:

Examples. (i) Evaluation of the polynomial

$$p(t) = (t-2)^4 = t^4 - 8t^3 + 24t^2 - 32t + 16$$

for $t = 2.0001$ delivers the verified inclusion of $p(t)$:

$$[1.000000000008441E - 016, 1.000000000008442E - 016]$$

after two iterations. Horner scheme evaluation in floating-point arithmetic yields

$$-3.552713678800501E - 015.$$

(ii) Evaluation of the polynomial

$$p(t) = (1-t)^3 = -t^3 + 3t^2 - 3t + 1$$

for $t = 1.000005$ delivers the verified inclusion of $p(t)$:

$$[-1.250000000024568E - 016, -1.250000000024566E - 016]$$

after two iterations. Horner scheme evaluation in floating-point arithmetic yields

$$1.110223024625157E - 016.$$

(iii) It is well known that evaluation of the power series is not a suitable method for computing values of the exponential function for negative arguments. With the new method evaluation is simple. With

$$p(t) = \sum_{k=0}^{90} \frac{t^k}{k!}$$

we obtain for $t = -20$ the inclusion of $p(t)$:

$$[2.06115362243E - 9, 2.06115362244E - 9]$$

whereas Horner scheme evaluation in floating-point arithmetic yields

$$1.188534E - 4.$$

Here an arithmetic with 13 decimal digits was used.

Note: The coefficients $t^k/k!$ are not exactly representable. Therefore, the computation was done for $90!p(t)$. To obtain an enclosure of $\exp(-20)$ an enclosure of the 91st summand would still have to be computed and added to the received interval.

Throughout this subsection it has been assumed that the coefficients of the polynomial are floating-point numbers. In [389] R. Lohner has extended the method to polynomials with long interval coefficients.

9.6.3 Arithmetic Expressions

For brevity in this section we only sketch the methods. They follow the scheme used to obtain highly accurate enclosures for values of polynomials. Again iterative refinement using an exact scalar product is the basic mathematical tool.

We begin with the example

$$(a + b) \cdot c - d/e,$$

where a, b, c, d, e are floating-point numbers. Evaluation of the expression can be performed by the following steps:

$$\begin{aligned} x_1 &:= a, \\ x_2 &:= x_1 + b, \\ x_3 &:= x_2 \cdot c, \\ x_4 &:= d, \\ x_5 &:= x_4/e, \\ x_6 &:= x_3 - x_5. \end{aligned}$$

This is again a system of linear equations with a lower triangular matrix. An enclosure of the value of the expression can be computed very similarly to the case of polynomials.

But there are arithmetic expressions which lead to a system of nonlinear equations. For example, the expression

$$(a+b)(c+d)$$

with floating-point numbers a, b, c, d leads to the system of nonlinear equations:

$$\begin{aligned} x_1 &:= a, \\ x_2 &:= x_1 + b, \\ x_3 &:= c, \\ x_4 &:= x_3 + d, \\ x_5 &:= x_2 \cdot x_4. \end{aligned}$$

All such systems are of a special lower triangular form. Whenever a new auxiliary variable is introduced, only those with lower indices appear on the right hand side.

If the resulting system is nonlinear it can be transformed into a linear system by an automatic algebraic transformation process. Then iterative refinement techniques with an exact scalar product can be applied in a very similar way to the polynomial case. See [86, 87].

Other methods directly use the system of nonlinear equations. These methods again proceed in two steps:

(a) an approximation $x^{(0)}$ for the correct solution x^* is computed in floating-point arithmetic by forward substitution.

(b) an enclosure E of the error $e = x^* - x^{(0)}$ is computed.

This leads to an enclosure for the solution:

$$x^* \in x^{(0)} + E.$$

Formulas for E are obtained by use of the mean value theorem. They are evaluated by the exact scalar product.

If the quality of the enclosure $x^* \in x^{(0)} + E$ is not sufficient, the midpoint $x^{(1)} := m(E)$ is used to improve the approximation $x^{(0)}$ to $x^{(0)} + x^{(1)}$. Then step (b) is repeated and so on.

A detailed description and executable algorithms and programs can be found in Chapter 8 of the books [205, 206, 207]. See also [175].

The methods have also been extended to matrix expressions and complex expressions.

Example 9.7. The method which uses the nonlinear system of equations described in this subsection works well even for extremely ill conditioned problems, for instance in the example:

$$333.75 \cdot b^6 + a^2 \cdot (11 \cdot a^2 \cdot b^2 - b^6 - 121 \cdot b^4 - 2) + 5.5 \cdot b^8 + a/(2 \cdot b). \quad (9.6.1)$$

Evaluation of the expression in IEEE double precision floating-point arithmetic for $a = 77617.0$ and $b = 33096.0$ delivers the value

$$1.172604.$$

The same value is obtained in double and extended precision on a /370 mainframe in IBM floating-point arithmetic. The method sketched here delivers the enclosure

$$[-0.8273960599469, -0.8273960599467]$$

after three iterations.

Due to its developer the polynomial (9.6.1) is known as the *Rump polynomial* in the literature.

9.7 Multiple Precision Arithmetics

With a fast and exact *multiply and accumulate* operation or scalar product, fast quadruple and multiple precision arithmetics can easily be provided on the computer for real as well as for interval data. A multiple precision number, for instance, is represented as an array of floating-point numbers. The value of this number is the sum of its components. It can be represented in the complete register as a long fixed-point variable. In this section we briefly sketch the definition of the operations $+$, $-$, $\cdot$, $/$, and the *square root* for multiple precision numbers and for multiple precision intervals. Further information and PASCAL-XSC programs can be found in the literature [332, 392].

On the basis of these operations for these multiple precision numbers and intervals, algorithms for the elementary functions also can easily be provided. They have been provided and are available in the programming languages PASCAL-XSC, ACRITH-XSC, and Fortran-XSC. The concept of function and operator overloading in these languages makes application of multiple precision real and interval arithmetic very simple.

We assume that multiple precision numbers and intervals are elements of special data types which we call long real and long interval, respectively.

9.7.1 Multiple Precision Floating-Point Arithmetic

A multiple precision number x of type `long real` is represented as an array of floating-point numbers. The value of the number is the sum of its components:

$$x = \sum_{i=1}^{n} x_i, \tag{9.7.1}$$

and n is called the length of the representation (9.7.1).

Sometimes we will refer to x_i as the ith component of x. Of course, the representation of a multiple precision number in the form (9.7.1) is not unique. The components x_i are floating-point numbers and as such they may have different signs. Also there may be representations of x of different lengths. However, this will not present any difficulties. All calculations will be carried out in a complete register which is a long fixed-point register and therefore provides a means for the unique representation of numbers and intermediate results.

It is desirable that the absolute values of the components in (9.7.1) are ordered as $|x_1| > |x_2| > \ldots > |x_n|$ and that the exponents of two successive summands x_i, x_{i+1} differ at least by l where l is the mantissa length. We then say that the mantissas do not overlap. In this case x is represented with an optimal precision of about $n \cdot l$ mantissa digits. However, overlapping mantissas are not excluded in (9.7.1) but their use means loss of precision.

We assume that the number of components of a multiple precision number is controlled by a global variable called `stagprec`, (staggered precision). If `stagprec` is 1, the `long real` data type is identical to the type `real`. If, for instance, `stagprec` is 4, each variable of this type consists of an array of four variables of type `real`. It is desirable that the variable `stagprec` can be increased or decreased at any point in a program. This enables the use of higher precision data and operations in numerically critical parts of a computation. It helps to increase software reliability.

Let now x, y and z be multiple precision numbers:

$$x = \sum_{i=1}^{n_x} x_i, \qquad y = \sum_{i=1}^{n_y} y_i, \qquad z = \sum_{i=1}^{n_z} z_i.$$

We suppose that z is the result of an arithmetic operation, where n_z is the current run time value of the variable `stagprec`, i.e., `stagprec`$= n_z$. We consider arithmetic operations for multiple precision numbers:

Negation. Negation of a multiple precision number x is obtained by changing the sign of all components:

$$-x = -\sum_{i=1}^{n_x} x_i = \sum_{i=1}^{n_x}(-x_i).$$

Arithmetic operations for multiple precision numbers are evaluated in a complete variable. In the following algorithms for addition, subtraction, and multiplication this complete register is called cv. After execution of any sequence of operations the content of the complete register still has to be converted into the multiple precision number $z = \sum_{i=1}^{n_z} z_i$. For this conversion it is reasonable to use the rounding towards zero. It is denoted by *chop*. The operator symbols $+$, $-$, and $\cdot$ in the algorithms denote the operations for real numbers.

Addition and Subtraction.

$$cv := \sum_{i=1}^{n_x} x_i \pm \sum_{i=1}^{n_y} y_i,$$

```
for $i := 1$ to $n_z$ do
    $z_i :=$ chop$(cv)$
    $cv := cv - z_i$
```

Multiplication.

$$cv := \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} x_i \cdot y_j, \tag{9.7.2}$$

```
for $i := 1$ to $n_z$ do
    $z_i :=$ chop$(cv)$
    $cv := cv - z_i$
```

Here the products $x_i \cdot y_i$ of the floating-point numbers x_i and y_i are to be computed to the full double length and accumulated in the complete register.

By the conversion into the multiple precision number z used after addition, subtraction, and multiplication we obtain non-overlapping components z_i, $i = 1(1)n_z$ which all have the same sign.

Division. For division we use an iterative algorithm which computes the n_z components z_i of the quotient $z = x/y$ successively. We start with computing

$$z_1 := (\square x) \boxslash (\square y).$$

Here $\square$ denotes the rounding to a floating-point number by the rounding towards zero and $\boxslash$ denotes the standard floating-point division. With this approximation we compute the next component z_2 of z by

$$z_2 := \square \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} y_i \cdot z_1 \right) \boxslash (\square y).$$

Here the expression in parentheses is computed correctly in the complete register as a single exact scalar product. The result then is rounded into a standard floating-point number by rounding toward zero. A floating-point division finally delivers the second component z_2 of z.

In general, if we have an approximation $\sum_{i=1}^{k} z_i$, the next component z_{k+1} is computed inductively by

$$z_{k+1} := \square \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{k} y_i \cdot z_j \right) \boxslash (\square y). \tag{9.7.3}$$

Here again the numerator is computed correctly in the complete register as a single exact scalar product and then rounded into a standard floating-point number. Finally a floating-point division is performed.

Since in (9.7.3) the residual of the approximation $\sum_{j=1}^{k} z_j$ is computed with only a single rounding, the components z_i, $i = 1(1)n_z$ of the quotient $z = x/y$ do not overlap. However, for division the quotient

$$z = \sum_{i=1}^{n_z} z_i$$

may have positive as well as negative components.

Scalar Product. Let $X = (x^k)$ and $Y = (y^k)$ be vectors with multiple precision components x^k and y^k respectively:

$$x^k = \sum_{i=1}^{n_x} x_i^k, \qquad y^k = \sum_{i=1}^{n_y} y_i^k.$$

By (9.7.2) the product $x^k \cdot y^k$ of the two vector components is obtained by

$$x^k \cdot y^k = \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} x_i^k \cdot y_j^k. \tag{9.7.4}$$

Computation of the scalar product of the two vectors $X = (x^k)$ and $Y = (y^k)$ is just the accumulation of all the products (9.7.4) for k from 1 to n:

$$X \cdot Y = \sum_{k=1}^{n} x^k \cdot y^k = \sum_{k=1}^{n} \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} x_i^k \cdot y_j^k.$$

This shows that the scalar product of the two multiple precision vectors X and Y can be computed by accumulating products of floating-point numbers in a single complete variable *cv*. The result is a multiple precision number $z = \sum_{i=1}^{n_z} z_i$. It is obtained by conversion of the complete register contents into the multiple precision number z by:

```
for $i := 1$ to $n_z$ do
    $z_i := \text{chop}(cv)$
    $cv := cv - z_i$
```

Similar considerations hold for the computation of two multiple precision matrices or for the computation of the defect of a system of linear equations with multiple precision data. Of course, the formulas for these computations are getting more and more complicated. But the user does not have to be concerned with these. By operator overloading the work is done by the computer automatically.

The key operation for all these processes is a fast and exact scalar product. Quadruple precision arithmetic is not a substitute for it.

At several occasions in this section rounding towards zero is applied where rounding to nearest could have been used instead. The reason for this is that rounding towards zero is simpler and faster in general than rounding to nearest.

9.7.2 Multiple Precision Interval Arithmetic

Definition 9.8. Let x_i, $i = 1(1)n$, $n \geq 0$, be floating-point numbers and $X = [x_{\text{low}}, x_{\text{high}}]$ an interval with floating-point bounds x_{low} and x_{high}. Then an element of the form

$$x = \sum_{i=1}^{n} x_i + X \tag{9.7.5}$$

is called a *long interval* of *length* n. The x_i are called the components of x and X is called the *interval component*. ■

In (9.7.5) n is permitted to be zero. Then the sum in (9.7.5) is empty and $x = X$ is just an interval with standard floating-point bounds.

In the representation (9.7.5) of a long interval it is desirable that the components do not overlap. The following operations for long intervals are written so that they produce results with this property.

Arithmetic operations for long intervals are defined as usual in interval arithmetic:

Definition 9.9. Let x and y be long intervals, then

$$x \circ y := \{\xi \circ \eta | \xi \in x \wedge \eta \in y\}, \text{ for } \circ \in \{+, -, \cdot, /\},$$

with $0 \notin y$ for $\circ = /$. ■

Of course, in general, this theoretical result is not representable on the computer. Here the result must be a long interval again. We do not, however, require that it is the least enclosing long interval of some prescribed length. But we must require that the computed long interval z is a superset of the result defined in Definition 9.9: $x \circ y \subseteq z$. Not to require optimality of the result gives room for a compromise between tightness of the enclosure and the efficiency of the implementation.

Negation.

$$-x = \sum_{i=1}^{n_x} (-x_i) + [-x_{\text{high}}, -x_{\text{low}}].$$

Let now x, y, and z be three long intervals:

$$x = \sum_{i=1}^{n_x} x_i + [x_{\text{low}}, x_{\text{high}}], \quad y = \sum_{i=1}^{n_y} y_i + [y_{\text{low}}, y_{\text{high}}], \quad z = \sum_{i=1}^{n_z} z_i + [z_{\text{low}}, z_{\text{high}}].$$

Addition and subtraction of two long intervals x and y simply add and subtract the lower and higher bounds of x and y into two complete registers which we call *lo* and *hi*. Their contents finally have to be converted into a long interval z. This conversion routine will be discussed after the description of the operations of addition and subtraction. In the following algorithms the symbols $+$, $-$, and $\cdot$ again denote the operations for real numbers.

Addition.

$$\begin{aligned} hi &:= \sum_{i=1}^{n_x} x_i + \sum_{i=1}^{n_y} y_i \\ lo &:= hi + x_{\text{low}} + y_{\text{low}} \\ hi &:= hi + x_{\text{high}} + y_{\text{high}} \\ z &:= \text{convert}(lo, hi, n_z). \end{aligned}$$

Subtraction.

$$\begin{aligned} hi &:= \sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} y_i \\ lo &:= hi + x_{\text{low}} - y_{\text{high}} \\ hi &:= hi + x_{\text{high}} - y_{\text{low}} \\ z &:= \text{convert}(lo, hi, n_z). \end{aligned}$$

Conversion.

```
convert($lo, hi, n_z$)
  $stop := false$
  $i := 0$
  repeat $i := i + 1$
    $z_{\text{low}} := \text{chop}(lo)$
    $z_{\text{high}} := \text{chop}(hi)$
    if $z_{\text{low}} = z_{\text{high}}$ then $z_i := z_{\text{low}}$
      $lo := lo - z_i$
      $hi := hi - z_i$
    else $z_i := 0$
      $stop := true$
  until *stop* or $i = n_z$
  for $i := i + 1$ to $n_z$ do $z_i := 0$
    $z_{\text{low}} := \triangledown lo$
    $z_{\text{high}} := \triangle hi$
```

This routine reads successive numbers z_{low} and z_{high} from the complete registers and, as long as they are equal and the length n_z has not yet been reached, they are assigned to z_i and subtracted from the complete registers. If z_{low} and z_{high} are different or the length n_z for the result z is reached, then the remaining values in the complete registers are converted to the interval component Z of z by appropriate rounding. If some of the z_i are not yet defined, they are set to zero.

The conversion routine has the property that the real components of z do not overlap.

Multiplication. Multiplication can be implemented in various ways yielding different results because of the subdistributivity law of interval arithmetic. Thus we have:

$$\begin{aligned} x \cdot y &= \left(\sum_{i=1}^{n_x} x_i + X\right)\left(\sum_{j=1}^{n_y} y_j + Y\right) \\ &\subseteq \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} x_i \cdot y_j + X\sum_{j=1}^{n_y} y_j + Y\sum_{i=1}^{n_x} x_i + XY \qquad (9.7.6) \\ &\subseteq \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} x_i \cdot y_j + \sum_{j=1}^{n_y} X y_j + \sum_{i=1}^{n_x} Y x_i + XY. \qquad (9.7.7) \end{aligned}$$

This seems to suggest that better results can be obtained from using the second line (9.7.6) than from the third line. However, to compute the products $X\sum_{i=1}^{n_y} y_i$ and $Y\sum_{i=1}^{n_x} x_i$ we first have to round the sums to machine intervals. As a consequence of these additional roundings, the second line (9.7.6) yields coarser enclosures than the third line. Therefore, we use line three for the multiplication algorithm.

Again, *lo* and *hi* are two complete registers, x_i, y_i are reals, X, Y are intervals, and z is the resulting long interval. The multiplication routine is as follows:

$$\begin{aligned}
lo &:= \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} x_i \cdot y_j \\
hi &:= lo \\
[lo, hi] &:= [lo, hi] + \sum_{j=1}^{n_y} X y_j + \sum_{i=1}^{n_x} Y x_i + X \cdot Y \\
z &:= \text{convert}(lo, hi, n_z).
\end{aligned}$$

Here again all accumulations in *lo* and *hi* are to be done without any intermediate roundings.

Division. For division, again an iterative algorithm is applied. It computes the n_z real components z_i of the quotient x/y successively.

To compute the approximation $\sum_{i=1}^{n_z} z_i$, we start with

$$z_1 := \square m(x) \boxslash \square m(y).$$

Here $m(x)$ and $m(y)$ represent points selected within x and y respectively, the midpoints for instance. $\square$ denotes the rounding to a floating-point number by the rounding towards zero and $\boxslash$ means floating-point division.

Now the components z_i, $i = 1(1)n_z$ of z are computed by the same formula as for a long real arithmetic:

$$z_{k+1} := \square \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{k} y_i z_j \right) \boxslash (\square m(y)). \tag{9.7.8}$$

Here the numerator is computed exactly in a complete register and then rounded towards zero into a floating-point number. Finally a floating-point division is performed.

This iteration again guarantees that the z_i do not overlap.

Now the interval component Z of the result z is computed as

$$Z := \Diamond \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{n_z} y_i z_j + X - \sum_{j=1}^{n_z} Y z_j \right) \Diamond\!\!\!\!/\, (\Diamond y), \tag{9.7.9}$$

where $\Diamond$ denotes the rounding to a floating-point interval and $\Diamond\!\!\!\!/$ denotes the division of two floating-point intervals.

It is not difficult to see that $z = \sum_{i=1}^{n_z} z_i + Z$ is a superset of the exact range $\{\xi \circ \eta \mid \xi \in x \wedge \eta \in y\}$. For $\alpha \in X$ and $\beta \in Y$ we have the identity

$$\frac{\sum_i^{n_x} x_i + \alpha}{\sum_i^{n_y} y_i + \beta} = \sum_j^{n_z} z_j + \frac{\sum_i^{n_x} x_i + \alpha - \sum_i^{n_y} \sum_j^{n_z} y_i z_j - \sum_j^{n_z} z_j \beta}{\sum_i^{n_y} y_i + \beta}.$$

An interval evaluation of this expression for $\alpha \in X$ and $\beta \in Y$ shows immediately that the exact range x/y is contained in $\sum_{i=1}^{n_z} z_i + Z$ as computed by (9.7.8) and (9.7.9).

This leads to the following algorithm. Therein x_i, y_i, z_i, and y_m are floating-point reals, X, Y, and Z are floating-point intervals. $\square$ denotes rounding towards zero into a floating-point number and $\Diamond$ denotes the rounding to a floating-point interval.

```
$lo := \sum_{i=1}^{n_x} x_i$
$my := \square m(y)$
$z_1 := (\square lo) \boxslash my$
for $k := 2$ to $n_z$ do
    $lo := lo - \sum_{i=1}^{n_y} y_i z_{k-1}$
    $z_k := (\square lo) \boxslash my$
$lo := lo - \sum_{i=1}^{n_y} y_i z_{n_z}$
$hi := lo$
$Z := \Diamond([lo, hi] + X - \sum_{i=1}^{n_z} Y z_i) \mathbin{\Diamond\!\!\!/} (\Diamond y)$.
```

In this algorithm the double sum in (9.7.8) and (9.7.9) is accumulated in the complete register *lo* as long as the z_k are computed. The final value in *lo* is then used in the computation of the interval part Z. Thus the amount of work is reduced to a minimum.

Scalar Product. We leave it to the reader to derive formulas for the computation of the scalar product of two vectors with long interval components. We mention, however, that this is not necessary at all. By operator overloading the computer solves this problem automatically. What is needed are two complete registers and a fast and exact scalar product for floating-point numbers.

Square Root. An algorithm for the square root can be obtained analogously as in the case of division. It computes the z_i, $i = 1(1)n_z$ of the approximation part iteratively:

$$\begin{aligned} z_1 &:= \sqrt{\square x}, \\ z_{k+1} &:= \square \left(\sum_{i=1}^{n_x} x_i - \sum_{i,j=1}^{k} z_i z_j \right) / (2 z_1). \end{aligned} \tag{9.7.10}$$

This guarantees that the z_i do not overlap since in the numerator of (9.7.10) the defect of the approximation $\sum_{i=1}^{n_z} z_i$ is computed with one rounding only. Now the interval part Z is computed as

$$Z := \frac{\diamondsuit \left(\sum_{i=1}^{n_x} x_i - \sum_{i,j=1}^{n_z} z_i z_j + X \right)}{\sqrt{\diamondsuit x} + \diamondsuit \sum_{i=1}^{n_z} z_i} . \tag{9.7.11}$$

As in the case of division, it is easy to see that $\sum_{i=1}^{n_z} z_i + Z$ as computed by (9.7.10) and (9.7.11) is a superset of the exact range $\{\sqrt{\xi} \mid \xi \in x\}$; in fact, for all $\gamma \in X$ we have the identity:

$$\sqrt{\sum_{i=1}^{n_x} x_i + \gamma} = \sum_{j=1}^{n_z} z_j + \frac{\sum_{i=1}^{n_x} x_i + \gamma - \sum_{i,j=1}^{n_z} z_i z_j}{\sqrt{\sum_{i=1}^{n_x} x_i + \gamma} + \sum_{j=1}^{n_z} z_j} . \tag{9.7.12}$$

This leads to the following algorithm for the computation of the square root:

```
$lo := \sum_{i=1}^{n_x} x_i$
$z_1 := \sqrt{\square\, lo}$
for $k := 2$ to $n_z$ do
    $lo := lo - 2\sum_{j=1}^{k-2} z_j z_{k-1} - z_{k-1} z_{k-1}$
    $z_k := (\square\, lo)/(2 z_1)$
$lo := lo - 2\sum_{j=1}^{n_z-1} z_j z_{n_z} - z_{n_z} z_{n_z}$
$hi := lo$
$Z := \diamondsuit([lo, hi] + X)/(\sqrt{\diamondsuit x} + \diamondsuit \sum_{j=1}^{n_z} z_j)$.
```

To allow easy application of the long interval arithmetic just described a few additional operations should be supplied such as computation of the infimum, the supremum, the diameter, and the midpoint. Also elementary functions can be and have been implemented for long intervals. They may already make use of the arithmetic for long intervals.

9.7.3 Applications

We now briefly sketch a few applications of multiple precision interval arithmetic. We restrict the discussion to problem classes which have already been dealt with in this chapter. We assume that the floating-point inputs to an algorithm are exact. Imprecise data should be brought into the computer as intervals as accurately as possible, possibly as long intervals. It should be clear that in such a case the result is a set, and if the algorithm is unstable, this set may well be large. Even the best arithmetic can only compute bounds for this set. These bounds may not look very accurate even if they may be so.

Among the first problems that have been solved to very high and guaranteed accuracy was systems of linear equations by S. M. Rump [503]. The method was then continually extended to other problem classes. We consider a system of linear equations $A \cdot x = b$. Let x_1 be an approximate solution and $e_1 := x^* - x_1$ be the error to the exact solution x^*. Then e_1 is the solution of the system

$$A \cdot e_1 = b - Ax_1. \tag{9.7.13}$$

If we compute an interval enclosure X_1 of e_1, we have an enclosure of x^* by a long interval: $x^* \in x_1 + X_1$. This method can now be iterated. With a new approximate solution $x_2 := m(x_1 + X_1)$, where m denotes the midpoint of the interval $x_1 + X_1$, a second error e_2 can be computed by

$$A \cdot e_2 = b - Ax_2.$$

An enclosure X_2 of e_2 leads to a new enclosure of x^*:

$$x^* \in x_2 + X_2.$$

Essential for success of this method is the fact that the defect $b - A \cdot x_i$ of the approximate solution x_i can be computed to full accuracy by the exact scalar product.

This method of iterated defect correction can also be applied to compute with very high accuracy enclosures of arithmetic expressions or of polynomials. An enclosure of the solution is obtained as a long interval.

The methods just discussed can also be applied to problems where the initial data themselves are long intervals. See [389].

The method for the evaluation of polynomials with long interval coefficients allows additional applications. It can be applied, for instance, to evaluate higher dimensional polynomials and to represent the result as a long interval. To avoid too many indices we sketch the method for the two-dimensional case. The independent variables are denoted by x and y. Let the polynomial of degree n in x and m in y be

$$p(x, y) = \sum_{j=0}^{m} \sum_{i=0}^{n} a_{ij} x^i y^j = \sum_{j=0}^{m} \left(\sum_{i=0}^{n} a_{ij} x^i \right) y^j.$$

Its value can be obtained by successively computing the values of the $m + 2$ one-dimensional polynomials

$$b_j := b_j(x) := \sum_{i=0}^{n} a_{ij} x^i, \;\; j = 0(1)m, \tag{9.7.14}$$

and

$$p(x, y) := \sum_{j=0}^{m} b_j y^j. \tag{9.7.15}$$

The results of the computation (9.7.14) are long intervals. The final result (9.7.15) is also a long interval. It may be rounded into a floating-point interval if desired.

Long interval arithmetic has also been very successfully applied to the computation of orbits of discrete dynamic systems. It is well known that such computations are highly unstable if the system exhibits chaotic behavior. In this case even for very simple systems ordinary floating-point arithmetic delivers results which are completely wrong. Also ordinary interval arithmetic (i.e., intervals of floating-point numbers) yields very poor enclosures after a few iterations. By use of a long interval arithmetic highly accurate enclosures of orbits can be computed for a considerably longer time.

Of course, similar results also can be obtained by means of multiple precision arithmetic simulated in software by integer arithmetic. However, if the exact multiply and accumulate operation is available in hardware, the long interval arithmetic fully benefits from the speed of this floating-point hardware. Algorithms and programs are available in [389, 392].

9.7.4 Adding an Exponent Part as a Scaling Factor to Complete Arithmetic

The components of a multiple precision variable and the bounds of a multiple precision interval variable are floating-point numbers. If the basic floating-point format is double precision this seems to be a severe limitation for the multiple precision arithmetics considered in this section. If the largest component of a multiple precision datum has a small exponent it may well be that one or several of the other components underflow the exponent range of the basic floating-point format. This is particularly troublesome in case of multiple precision interval arithmetic. In applications of interval arithmetic it is essential that the width of the intervals be kept as tiny as possible. If components of a multiple precision interval variable lie in the underflow region of the basic floating-point format a rounding into a floating-point interval means a great loss of precision and accuracy.

As an example let us assume that a and b are multiple precision real numbers, both of 144 decimal digits and a small exponent

$$a = 1.098 \cdots 981 \cdot 10^{-154}, \qquad b = 1.234 \cdots 789 \cdot 10^{-154}.$$

Then the exact product $a \cdot b = 1.343 \cdots \cdot 10^{-308}$ has 288 decimal digits and its exponent nearly underflows the IEEE 754 double precision exponent range. Thus an enclosure of the exact product $a \cdot b$ can only be computed to about 16 decimal digits. This is a great loss of accuracy compared with the correct product.

This loss of accuracy can be avoided by scaling a and b appropriately, for instance,

$$a_s := 2^{+1022} \cdot a, \qquad b_s := 2^{+1022} \cdot b.$$

Then an enclosure of $a_s \cdot b_s$ can be computed to least bit accuracy, i.e., to 288 correct digits and we have

$$a \cdot b = 2^{-2044} \cdot (a_s \cdot b_s).$$

In conventional floating-point arithmetic the exponent part accomplishes an automatic scaling of the computation. In a very similar manner now an exponent part can be added to a multiple precision real or interval variable x with an appropriately chosen exponent range $E_{\text{min}} \leq ex \leq E_{\text{max}}$. Here ex is an integer. Thus a scaled multiple precision real or interval variable is a pair (ex, x). Based on the developments in Sections 9.7.1 and 9.7.2, arithmetic for these pairs can now be defined. The result is a long real or long interval arithmetic with an exponent part as a scaling factor.

Hardware implementation of an exact scalar product (complete arithmetic) is not at all more complicated than implementation of a full quadruple precision arithmetic as defined in IEEE P 754. By pipelining an exact scalar product can be computed with extreme speed. This allows a very fast multiple precision real and interval arithmetic. By adding an exponent part as a scaling factor a very flexible and fast computing tool is obtained. It allows highly accurate solution of even very ill-conditioned problems.

Multiple precision data types with an exponent part as scaling factor for the basic data types real, interval, complex, and complex interval have been made available in C-XSC by Blomquist, Krämer and Hofschuster [288, 331]. The number of components used for the fraction part is controlled by a global variable `stagprec` (staggered precision) which can be increased or decreased during program execution depending on the desired accuracy. With reasonably chosen E_{min} and E_{max} underflow and overflow need not occur.

To make these scaled multiple precision arithmetics generally applicable, elementary and possibly also special functions have to be provided. Here care has to be taken that the functions are evaluated with high accuracy in short computing times over large definition ranges.

The programming environment C-XSC provides a large class of elementary and special functions for multiple precision data with an exponent part as scaling factor for the basic data types real, interval, complex, and complex interval, see [331].

What has been discussed in this section can be seen as advanced floating-point arithmetic, floating-point arithmetic on a higher level, adapted to the needs of high speed scientific computing. Via complete arithmetic this advanced floating-point arithmetic is based ultimately on the standardized double precision floating-point arithmetic. If complete arithmetic is hardware supported by pipelining the advanced floating-point arithmetics for real and interval data are also very fast. Very ill-conditioned problems have been solved with these arithmetics to several hundred correct digits.

An exact scalar product could also be adapted to implement greatly extended precision integer arithmetic.

Appendix A

Frequently Used Symbols

A logical statement has a value that is either true or false, t or f for short. Such statements can be combined in logical expressions by logical operators. In this treatise the logical operators *and*, *or* and *not* are occasionally used. They are denoted by the symbols $\wedge$, $\vee$, and $\neg$ respectively and are defined by the following table:

a	b	$\neg a$	$a \wedge b$	$a \vee b$
t	t	f	t	t
t	f	f	f	t
f	t	t	f	t
f	f	t	f	f

In words, $a \wedge b$ is true if both operands are true, and $a \vee b$ is true if at least one operand is true.

Besides the logical operators $\wedge$ and $\vee$, the *for all quantor* $\bigwedge$, and the *existence quantor* $\bigvee$ are frequently used in the text. If M is a set, $a \in M$ and $P(a)$ is a logical statement, then

(i) $\bigwedge\limits_{a \in M} P(a)$ means: $P(a)$ is true for all $a \in M$.

(ii) $\bigvee\limits_{a \in M} P(a)$ means: there exists an $a \in M$ such that $P(a)$ is true.

Thus the *for all quantor* $\bigwedge$, and the *existence quantor* $\bigvee$ in a certain sense can be understood as generalizations of the logical operators $\wedge$ and $\vee$ respectively. That is, if M is a finite set with elements $a_1, a_2, \ldots, a_n$, then

(i) $\bigwedge\limits_{a \in M} P(a)$ is equivalent to $P(a_1) \wedge P(a_2) \wedge \ldots \wedge P(a_n)$, and

(ii) $\bigvee\limits_{a \in M} P(a)$ is equivalent to $P(a_1) \vee P(a_2) \vee \ldots \vee P(a_n)$.

Other frequently used symbols:

$\bigtriangledown$	symbol for rounding downward.
$\bigtriangleup$	symbol for rounding upward.
$\bigtriangledown\!\!\!\!\!\!\!{\scriptstyle +}\;, \bigtriangledown\!\!\!\!\!\!\!{\scriptstyle -}\;, \bigtriangledown\!\!\!\!\!\!\!{\scriptstyle \cdot}\;\;, \bigtriangledown\!\!\!\!\!\!\!{\scriptstyle /}\;$	floating-point operations with rounding downward.
$\bigtriangleup\!\!\!\!\!\!\!{\scriptstyle +}\;, \bigtriangleup\!\!\!\!\!\!\!{\scriptstyle -}\;, \bigtriangleup\!\!\!\!\!\!\!{\scriptstyle \cdot}\;\;, \bigtriangleup\!\!\!\!\!\!\!{\scriptstyle /}\;$	floating-point operations with rounding upward.
$a\|\|b$	the elements a and b of an ordered set are incomparable.
$\mathring{A}$	denotes the interior of the interval $A = [a_1, a_2]$, i.e., $c \in \mathring{A}$ means $a_1 < c < a_2$.
$I\mathbb{R}$	set of closed and bounded real intervals.
$(I\mathbb{R})$	set of extended real intervals together with closed and bounded real intervals.

Appendix B
On Homomorphism

Theorem B.1. *Let $\{\mathbb{R}, +\}$ be the additive group of real numbers, S a finite subset of $\mathbb{R}$ with n elements, and $0 \in S$. If $\square : \mathbb{R} \to S$ is a mapping with $\square(0) = 0$ and $\boxplus$ an operation in S with the property (homomorphism)*

$$\bigwedge_{a,b\in\mathbb{R}} \square(a+b) = (\square a) \boxplus (\square b), \tag{B.0.1}$$

then

$$\bigwedge_{a,b\in\mathbb{R},a<b} \ \bigvee_{x\in[a,b]} \square x = 0.$$

Proof. Let $c_1, c_2, \ldots, c_{n+1} \in [a,b]$ be distinct elements of $[a,b]$, $c_i \neq c_j$, for $i \neq j$, then by a possible renumbering

$$\square c_1 = \square c_2. \tag{B.0.2}$$

We obtain

$$\square(c_2 + (-c_1)) = \square(c_2 - c_1) \underset{\text{(B.0.1)}}{=} \square(c_2) \boxplus (\square(-c_1))$$
$$\underset{\text{(B.0.2),(B.0.1)}}{=} \square(c_1 - c_1) = \square(0) = 0.$$

Thus with $c = c_2 - c_1$ we have $\square c = \square(0) = 0$. Moreover

$$\square(2c) = \square(c+c) \underset{\text{(B.0.1)}}{=} \square(c) \boxplus \square(c)$$
$$= 0 \boxplus 0 \underset{\text{(B.0.1)}}{=} \square(0+0) = \square(0) = 0,$$

and by induction

$$\square(kc) = 0 \ \text{ for all } \ k \in \mathbb{N}.$$

Since $c = c_2 - c_1 \leq b - a$, there exists an $r \in \mathbb{N}$ with $rc \in [a,b]$, and with $x := rc$ we have $\square x = 0$ with $x \in [a,b]$. ■

A minimum requirement of a rounding is the property

(R1) $\square a = a$ for all $a \in S$,

i.e., all elements of S are fixed points of the mapping. By Theorem B.1 in case of a homomorphism every interval $[a,b] \subset \mathbb{R}$, whatever small it might be, contains an element x which is mapped onto zero. A mapping with this property is not a reasonable rounding. For further details see [503].

Bibliography

[1] O. Aberth, *Precise Numerical Analysis*, Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[2] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, Nat. Bur. Standards, Appl. Math. Series 55, U.S. Government Printing Office, Washington, D.C., 1964.

[3] E. Adams, Enclosure methods and scientific computation, in: [48], pp. 3–31, 1989.

[4] E. Adams and U. Kulisch, On scientific computing with automatic result verification, in: [5], pp. 1–12, 1993.

[5] E. Adams and U. Kulisch (eds.), *Scientific Computing with Automatic Result Verification*. I. Language and Programming Support for Verified Scientific Computation, II. Enclosure Methods and Algorithms with Automatic Result Verification, III. Applications in the Engineering Sciences, Academic Press, San Diego, 1993.

[6] E. Adams and R. Lohner, Error bounds and sensitivity analysis, in: *Scientific Computing*, edited by R. S. Stepleman, pp. 213–222, North Holland, Amsterdam, 1983.

[7] R. C. Agarwal, J. W. Cooley, F. G. Gustavson, J. B. Shearer, G. Slishman and B. Tuckerman, *New Scalar and Vector Elementary Functions for the IBM System/370*, IBM Journal of Research and Development 30:2 (1986), 123–144.

[8] A. Akkas, *Instruction Set Enhancements for Reliable Computations*, Ph.D. Thesis, Lehigh University, January 2002.

[9] A. Akkas, A combined interval and floating-point comparator/selector, in: *IEEE 13th International Conference on Application-specific Systems, Architectures and Processors*, San Jose, USA, July, 2002, pp. 208–217, 2003.

[10] R. Albrecht, G. Alefeld and H. J. Stetter (eds.), *Validation Numerics – Theory and Applications*, Computing Supplementum 9, Springer, Wien New York, 1993.

[11] R. Albrecht and U. Kulisch (eds.), *Grundlagen der Computerarithmetik*, Computing Supplementum 1, Springer, Wien New York, 1977.

[12] G. Alefeld, *Intervallrechnung über den komplexen Zahlen und einige Anwendungen*, Dissertation, Universität Karlsruhe, 1968.

[13] G. Alefeld, Über die aus monoton zerlegbaren Operatoren gebildeten Iterationsverfahren, *Computing* 6 (1970), 161–172.

[14] G. Alefeld, Über die Durchführbarkeit des Gaußschen Algorithmus bei Gleichungen mit Intervallen als Koeffizienten, in: [11], pp. 15–19, 1977.

[15] G. Alefeld, Intervallanalytische Methoden bei nichtlinearen Gleichungen, in: *Jahrbuch Überblicke Mathematik 1979*, pp. 63–78, 1979.

[16] G. Alefeld, Bounding the slope of polynomials and some applications, *Computing* 26 (1981), 227–237.

[17] G. Alefeld, On the convergence of some interval-arithmetic modifications of Newton's method, in: *Scientific Computing*, edited by R. Stepleman, pp. 223–230, IMACS/North Holland Publishing Company, Amsterdam, 1983.

[18] G. Alefeld, Berechenbare Fehlerschranken für ein Eigenpaar unter Einschluss von Rundungsfehlern bei Verwendung des genauen Skalarproduktes, *Z. Angew. Math. Mech.* 67 (1987), 145–152.

[19] G. Alefeld, Rigorous error bounds for singular values of a matrix using the precise scalar product, in: [270], pp. 9–30, 1987.

[20] G. Alefeld, Über die Konvergenz des Intervall-Newton-Verfahrens, *Computing* 39 (1987), 363–369.

[21] G. Alefeld, Errorbounds for quadratic systems of nonlinear equations using the precise scalar product, in: [372], pp. 59–67, 1988.

[22] G. Alefeld, Existence of solutions and iterations for nonlinear equations, in: [426], pp. 207–227, 1988.

[23] G. Alefeld, Enclosure methods, in: [591], pp. 55–72, 1990.

[24] G. Alefeld, Über das Divergenzverhalten des Interval-Newton-Verfahrens, *Computing* 46 (1991), 289–294.

[25] G. Alefeld, Inclusion methods for systems of nonlinear equations – the interval Newton method and modifications, in: [224], pp. 7–26, 1994.

[26] G. Alefeld, A. Frommer and B. Lang (eds.), *Scientific Computing and Validated Numerics*, Proceedings of SCAN-95, Akademie Verlag, Berlin, 1996.

[27] G. Alefeld and R. D. Grigorieff (eds.), *Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis)*, Computing Supplementum 2, Springer, Wien New York, 1980.

[28] G. Alefeld and J. Herzberger, *Einführung in die Intervallrechnung*, Informatik 12, Bibliographisches Institut, Mannheim Wien Zürich, 1974.

[29] G. Alefeld and J. Herzberger, *Introduction to Interval Computations*, Academic Press, New York, 1983.

[30] G. Alefeld and J. Herzberger (eds.), *Numerical Methods and Error Bounds*, Mathematical Research 89, Akademie Verlag, Berlin, 1996.

[31] G. Alefeld, B. Illg and F. Potra, On a class of enclosure methods for systems of equations with higher order of convergence, in: [591], pp. 151–159, 1990.

[32] G. Alefeld, G. Kreinovich and G. Mayer, On the shape of the symmetric, and skew-symmetric solution set, *SIAM J. Matrix Anal. Appl.* 18 (1997), 693–705.

[33] G. Alefeld, G. Kreinovich and G. Mayer, The shape of the solution set of linear interval equations with dependent coefficients, *Math. Nachr.* 192 (1998), 23–36.

[34] G. Alefeld and R. Lohner, On higher order centered forms, *Computing* 35 (1985), 177–184.

[35] G. Alefeld and G. Mayer, *The Cholesky Method for Interval Data*, Presentation at the International Conference on "Numerical Analysis with Automatic Result Verification", Lafayette, Louisiana, USA, February 25 – March 1, 1993.

[36] G. Alefeld and G. Mayer, A computer-aided existence and uniqueness proof for an inverse matrix eigenvalue problem, *Int. J. Interval Computations* 1 (1994), 4–27.

[37] G. Alefeld and G. Mayer, Einschließungsverfahren, in: [226], pp. 155–186, 1995.

[38] G. Alefeld and G. Mayer, On the symmetric and unsymmetric solution set of interval systems, *SIAM J. Matrix Anal. Appl.* 16 (1995), 1223–1240.

[39] G. Alefeld and G. Mayer, Interval analysis: theory and applicastions, *J. Comput. Appl. Math.* 121 (2000), 421–464.

[40] G. Alefeld and G. Mayer, *On singular interval systems*, in: [47], pp. 191–197, 2004.

[41] G. Alefeld, J. Rohn, S. M. Rump and T. Yamamoto (eds.), *Symbolic Algebraic Methods and Verification Methods*, Springer, Wien New York, 2001.

[42] G. Alefeld and U. Schäfer, Iterative methods for linear complementary problems with interval data, *Computing* 70 (2003), 235–259.

[43] J. Alex, *Wege und Irrwege des Konrad Zuse*, Spektrum der Wissenschaft, Januar 1997, 78–90.

[44] U. Allendörfer and D. Shiriaev, PASCAL-XSC to C – a portable PASCAL-XSC compiler, in: [271], pp. 91–104, 1991.

[45] U. Allendörfer and D. Shiriaev, *PASCAL-XSC. A Portable Development System*, Proceedings of 13th World Congress on Computation and Applied Mathematics, IMACS '91, Dublin, 1991.

[46] U. Allendörfer and D. Shiriaev, PASCAL-XSC. A portable development system, in: [125], pp. 11–20, 1992.

[47] R. Alt, A. Frommer, R. B. Kearfott and W. Luther (eds.), *Numerical Software with Result Verification*, Lecture Notes in Computer Science, Springer, Berlin, 2004.

[48] W. F. Ames (ed.), *Numerical and Applied Mathematics*, J. C. Baltzer Scientific Publishing, Basel, 1989.

[49] A. S. Andreev, I. T. Dimov, S. M. Markov and Ch. Ullrich (eds.), *Mathematical Modelling and Scientific Computations*, Bulgarian Academy of Sciences, Sofia, 1991.

[50] N. Apostolatos, U. Kulisch, R. Krawczyk, B. Lortz, K. Nickel and H.-W. Wippermann, The algorithmic language Triplex-ALGOL 60, *Numerische Mathematik* 11 (1968), 175–180.

[51] H.-R. Arndt and G. Mayer, On the semi-convergence of interval matrices, *Linear Algebra and its Applications* 393 (2004), 15–37.

[52] L. Atanassova and J. Herzberger (eds.), *Computer Arithmetic and Enclosure Methods*, Proceedings of SCAN 91, North-Holland, Elsevier Science Publishers B. V., Amsterdam, 1992.

[53] W. Auzinger and H. J. Stetter, Accurate arithmetic results for decimal data on non-decimal computers, *Computing* 35 (1985), 141–151.

[54] D. H. Bailey, *A Portable High Performance Multiprecision Package*, RNA Technical Report RNR-90-022, 1993.

[55] D. H. Bailey, A Fortran-90 based multiprecision system, *ACM Trans. Math. Software* 21 (1995), 379–387.

[56] B. Barth, *Eine verifizierte Einschließung von Werten der Weierstraßschen $\wp$-Funktion*, Diplomarbeit am Institut für Angewandte Mathematik, Universität Karlsruhe, 1991.

[57] W. Barth and E. Nuding, Optimale Lösungen von Intervallgleichungssystemen, *Computing* 12 (1974), 117–125.

[58] H. Bauch, K.-U. Jahn, D. Oelschlägel, H. Süsse and V. Wiebigke, *Intervallmathematik. Theorie und Anwendungen*, BSB B. G. Teubner Verlagsgesellschaft, Leipzig, 1987.

[59] Ch. Baumhof, *Behavioural Description of a Scalar Product Unit*, Universität Karlsruhe, ESPRIT Project OMI/HORN, Deliverable Report D1.2/2, December 1992.

[60] Ch. Baumhof, A new VLSI vector arithmetic coprocessor for the PC, in: [654], Vol. 12, pp. 210–215, 1995.

[61] Ch. Baumhof, *Ein Vektorarithmetik-Koprozessor in VLSI-Technik zur Unterstützung des Wissenschaftlichen Rechnens*, Dissertation, Universität Karlsruhe, 1996.

[62] Ch. Baumhof and G. Bohlender, *A VLSI Vector Arithmetic Coprocessor for the PC*, Proceedings of WAI'96 in Recife/Brasil, RITA (Revista de Informática Teórica e Aplicada), Extra Edition, October 1996.

[63] W. De Beauclair, *Rechnen mit Maschinen*, Vieweg, Braunschweig, 1968.

[64] H. Beek, Über die Struktur und Abschätzungen der Lösungsmenge von linearen Gleichungssystemen mit Intervallkoeffizienten, *Computing* 10 (1972), 231–244.

[65] H. Behnke, Inclusion of eigenvalues of general eigenvalue problems of matrices, in: [372], pp. 69–78, 1988.

[66] H. Behnke, The determination of guaranteed bounds to eigenvalues with the use of variational methods II, in: [591], pp. 155–170, 1990 (Part I see [186]).

[67] H. Behnke, The calculation of guaranteed bounds for eigenvalues using complementary variational principles, *Computing* 47 (1992), 11–27.

[68] H. Behnke and U. Mertins, Bounds for eigenvalues with the use of finite elements, in: [366], pp. 119–131, 2001.

[69] H. Behnke, U. Mertins, M. Plum and Ch. Wieners, Eigenvalue inclusions via domain decomposition, *Proc. R. Soc. London* 456 (2000), 2717–2730.

[70] M. Berz, Automatic differentiation as nonarchimedean analysis, in: [52], pp. 439–450, 1992.

[71] M. Berz, C. Bischof, G. Corliss and A. Griewank (eds.), *Computational Differentiation, Techniques, Applications, and Tools*, SIAM, 1996.

[72] D. Bethke and J. Herzberger, Über Eigenschaften von zwei Methoden zur Einschließung der Inversen einer Intervallmatrix, in: [245], pp. 409–413, 1991.

[73] N. Bierlox, *Ein VHDL Koprozessor für das exakte Skalarprodukt*, Dissertation, Universität Karlsruhe, 2002.

[74] C. Bischof, A. Carle, G. Corliss, A. Griewank and P. Hovland, *ADIFOR: Fortran Source Translation for Efficient Derivatives*, ADIFOR Working Note No. 4, Argonne National Laboratory, address: see [147], Report MCS-P278-1291, February 1992.

[75] Ch. M. Black, R. B. Burton and T. H. Miller, The need for an industry standard of accuracy for elementary-function programs, *ACM Trans. on Math. Software* 10:4 (1984), 361–366.

[76] J. H. Bleher, S. M. Rump, U. Kulisch, M. Metzger, Ch. Ullrich and W. Walter, FORTRAN-SC: A study of a FORTRAN extension for engineering/scientific computation with access to ACRITH, *Computing* 39 (1987), 93–110 (also in [372], pp. 227–244, 1988).

[77] F. Blomquist, *Implementierung und Fehlerabschätzungen von PASCAL-XSC Standardfunktionen für ein dezimales Datenformat*, Universität Karlsruhe, 1992.

[78] F. Blomquist, *Implementierung einiger spezieller mathematischer Funktionen in PASCAL-XSC*, private communication, 1992.

[79] F. Blomquist, *PASCAL-XSC, BCD-Version 1.0, Benutzerhandbuch für das dezimale Laufzeitsystem*, Universität Karlsruhe, Institut für Angewandte Mathematik, 1997.

[80] F. Blomquist, *Verifzierende Numerik mit PASCAL-XSC, BCD-Version.* 1-713, Institut für Angewandte Mathematik, Universiät Karlsruhe, 2000, available at `www.math.uni-wuppertal.de/org/WRST/blom.html`.

[81] F. Blomquist, W. Hofschuster and W. Krämer, *Complex Interval Functions in C-XSC*, preprint BUW-WRSWT 2005/2, Universität Wuppertal, 2005.

[82] F. Blomquist, W. Hofschuster and W. Krämer, *Real and Complex Taylor Arithmetic in C-XSC*, preprint BUW-WRSWT 2005/4, Universität Wuppertal, 2005.

[83] F. Blomquist, W. Hofschuster and W. Krämer, *Vermeidung von Über- und Unterlauf und Verbesserung der Genauigkeit bei reeller und komplexer staggered Intervall-Arithmetik*, preprint BUW-WRSWT 2007, Universität Wuppertal, 2007.

[84] P. Bochev and S. Markov, A self-validating numerical method for the matrix exponential, *Computing* 43 (1989), 59–72.

[85] P. Bochev and S. Markov, Simultaneous self-verified computation of $\exp(A)$ and $\int_0^1 \exp(As)ds$, *Computing* 45 (1990), 183–191.

[86] H. Boehm, *Berechnung von Polynomnullstellen und Auswertung arithmetischer Ausdrücke mit garantierter maximaler Genauigkeit*, Dissertation, Universität Karlsruhe 1983.

[87] H. Boehm, Evaluation of arithmetic expressions with maximum accuracy, in: [368], pp. 121–137, 1983.

[88] G. Bohlender, Genaue Summation von Gleitkommazahlen, in: [11], pp. 21–32, 1977.

[89] G. Bohlender, *Floating-Point Computation of Functions with Maximum Accuracy*, IEEE Transactions on Computers, Vol. C-26, no. 7, July 1977.

[90] G. Bohlender, *Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen*, Dissertation, Universität Karlsruhe, 1978.

[91] G. Bohlender, What do we need beyond IEEE arithmetic?, in: [593], pp. 1–32, 1990.

[92] G. Bohlender, A vector extension of the IEEE standard for floating-point arithmetic, in: [271], pp. 3–12, 1991.

[93] G. Bohlender, Bibliography on enclosure methods and related topics, in: [5], pp. 571–608, 1993.

[94] G. Bohlender, *Literature List on Enclosure Methods and Related Topics*, Institut für Angewandte Mathematik, Universität Karlsruhe, Report, 1998.

[95] G. Bohlender, D. Cordes, A. Knöfel, U. Kulisch, R. Lohner and W. V. Walter, Proposal for accurate floating-point vector arithmetic, in: [5], pp. 87–102, 1993.

[96] G. Bohlender, K. Grüner, E. Kaucher, R. Klatte, W. Krämer, U. Kulisch, W. L. Miranker, S. Rump, Ch. Ullrich and J. Wolff von Gudenberg, *PASCAL-SC: A PASCAL*

for Contemporary Scientific Computation, Research Report RC 9009, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1981.

[97] G. Bohlender, E. Kaucher, R. Klatte, U. Kulisch, W. L. Miranker, Ch. Ullrich and J. Wolff von Gudenberg, FORTRAN for contemporary numerical computation, IBM Research Report RC 8348, *Computing* 26 (1981), 277–314.

[98] G. Bohlender and A. Knöfel, A survey of pipelined hardware support for accurate scalar products, in: [271], pp. 29–43, 1991.

[99] G. Bohlender, P. Kornerup, D. W. Matula and W. V. Walter, Semantics for exact floating-point operations, in: *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pp. 22–26, Grenoble France, IEEE Comp. Soc., 1991.

[100] G. Bohlender, W. Krämer and W. L. Miranker, Grading of Basic Arithmetical Operations and Functions, IBM Rsearch Report, RC 19593(86059)6-1-94, 1994.

[101] G. Bohlender, W. L. Miranker and J. Wolff von Gudenberg, Floating-point systems for theorem proving, in: *Computer Aided Proofs in Analysis*, edited by K. R. Meyer and D. S. Schmidt, Springer, 1990.

[102] G. Bohlender, L. Rall, Ch. Ullrich and J. Wolff von Gudenberg, *PASCAL-SC – Wirkungsvoll programmieren, kontrolliert rechnen*, Bibliographisches Institut, Mannheim, 1986.

[103] G. Bohlender, L. Rall, Ch. Ullrich and J. Wolff von Gudenberg, *PASCAL-SC: A Computer Language for Scientific Computation*, Academic Press, New York, 1987.

[104] G. Bohlender and T. Teufel, Demonstration of the bit-slice processor unit BAP–SC in a 68000 environment, in: [603], Vol. 1, pp. 155–158, 1985, or in: [532], pp. 331–336, 1986.

[105] G. Bohlender and T. Teufel, BAP–SC: A decimal floating-point processor for optimal arithmetic, in: [270], pp. 31–58, 1987.

[106] G. Bohlender, W. Walter, P. Kornerup and D. W. Matula, Semantics for exact floating-point operations, in: *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pp. 22–26, Grenoble France, IEEE Press, 1991.

[107] H. Böhm, Auswertung arithmetischer Ausdrücke mit maximaler Genauigkeit, in: [374], pp. 175–184, 1982.

[108] H. Böhm, *Berechnung von Polynomnullstellen und Auswertung arithmetischer Ausdrücke mit garantierter maximaler Genauigkeit*, Dissertation, Universität Karlsruhe, 1983.

[109] H. Böhm, Evaluation of arithmetic expressions with maximum accuracy, in: [368], pp. 121–137, 1983.

[110] J. M. Borwein and P. B. Borwein, *The Arithmetic-Geometric Mean and Fast Computation of Elementary Functions*, SIAM Review, Vol. 26, No. 3, July 1984.

[111] J. M. Borwein and P. B. Borwein, *Pi and the AGM*, John Wiley & Sons, 1987.

[112] J. M. Borwein, P. B. Borwein and D. H. Bailey, Ramanujan, modular equations, and approximations to Pi, *Amer. Math. Monthly* 96 (1989), 201–219, 1989.

[113] G. Brassard, S. Monet and D. Zuffellato, *Algorithmes pour l'arithmétique des très grands entiers*, TSI Technique et Science Informatiques, Vol. 5, No. 2, 1986.

[114] K. Braune, *Hochgenaue Standardfunktionen für reelle und komplexe Punkte und Intervalle in beliebigen Gleitpunktrastern*, Dissertation, Universität Karlsruhe, 1987.

[115] K. Braune, Standard functions for real and complex point and interval arguments with dynamic accuracy, *Computing Supplementum* 6 (1988), 159–184.

[116] K. Braune, A-posteriori Fehlerschranken bei der Berechnung inverser Standardfunktionen mit Hilfe des Newton-Verfahrens, *ZAMM* 70 (1990), T579–T581.

[117] K. Braune and W. Krämer, Standard functions for intervals with maximum accuracy, in: *Proceedings of the 11th IMACS World Congress*, Vol. 1, pp. 167–170, Oslo, 1985.

[118] K. Braune and W. Krämer, High-accuracy standard functions for intervals, in: *Computer Systems: Performance and Simulation*, edited by M. Ruschitzka, Elsevier Science Publishers, 1985.

[119] K. Braune and W. Krämer, High-accuracy standard functions for real and complex intervals, in: E. Kaucher, U. Kulisch and Ch. Ullrich, *Computerarithmetic: Scientific Computation and Programming Languages*, pp. 81–114, B. G. Teubner, Stuttgart, 1987.

[120] P. B. Brent, Fast multiple-precision evaluation of elementary functions, *J. of the Association for Computing Machinery* 23:2 (1976), 242–251.

[121] R. P. Brent, A FORTRAN multiple precision arithmetic package, *ACM Trans. Math. Software* 4 (1978), 57–70.

[122] R. P. Brent, *MP User's Guide*, fourth edition, Technical Report TR–CS–81–08, Department of Computer Science, Australian National University, Canberra, 1981.

[123] R. P. Brent, J. A. Hooper and J. M. Yohe, An Augment interface for Brent's multiple-precision arithmetic package, *ACM Trans. Math. Software* 6 (1980), 146–149.

[124] C. Brezinski (ed.), *SCAN'2002 International Conference* (Guest Editors: Rene Alt and Jean-Luc Lamotte), *Numerical Algorithms*, Vol. 37, 1–4, Kluwer Academic Publishers, 2004.

[125] C. Brezinski and U. Kulisch (eds.), *Computational and Applied Mathematics I – Algorithms and Theory*, Proceedings of the 13th IMACS World Congress, Dublin, Ireland, Elsevier Science Publishers B.V., 1992.

[126] K. Bühler and W. Barth, A new intersection algorithm for parametric surfaces based on LIEs, in: [329], pp. 179–190, 2001.

[127] R. Bulirsch and J. Stoer, Asymptotic upper and lower bounds for results of extrapolation methods, *Numerische Mathematik* 8 (1966), 93–104.

[128] P. R. Cappello and W. L. Miranker, Systolic super summation, *IEEE Transactions on Computers* 37:6 (1988), 657–677.

[129] P. R. Cappello and W. L. Miranker, *Systolic Super Summation with Reduced Hardware*, IBM Research Report RC 14259 (#63831), IBM Research Division, Yorktown Heights, New York, November 30, 1988.

[130] C. Y. Chen, *Adaptive numerische Quadratur und Kubatur mit Genauigkeitsgarantie*, Dissertation, Universität Karlsruhe, 1998.

[131] X. Chen, A verification method for solutions of nonsmooth equations, *Computing* 58 (1997), 281–294.

[132] D. M. Claudio, *Beiträge zur Struktur der Rechnerarithmetik*, Dissertation, Universität Karlsruhe, 1979.

[133] D. M. Claudio, Contribution to the structure of computer arithmetic, *Computing* 24 (1980), 115–118.

[134] D. M. Claudio, An algorithm for solving nonlinear equations based on the regula falsi and Newton methods, *ZAMM* 64 (1984), T407–T408.

[135] L. Collatz, *Funktionalanalysis und numerische Mathematik*, Springer, Berlin Heidelberg New York, 1968.

[136] D. Cordes, *Verifizierter Stabilitätsnachweis für Lösungen periodischer Differentialgleichungen auf dem Rechner mit Anwendungen*, Dissertation, Universität Karlsruhe, 1987.

[137] D. Cordes, Spärlich besetzte Matrizen, in: [357], pp. 129–136, 1989.

[138] D. Cordes, Runtime system for a PASCAL-XSC compiler, in: [271], pp. 151–160, 1991.

[139] D. Cordes and E. Kaucher, Self-validating computation for sparse matrix problems, in: [270], pp. 133–149, 1987.

[140] D. Cordes and W. Krämer, Vom Problem zum Einschließungsalgorithmus, in: *Wissenschaftliches Rechnen mit Ergebnisverifikation*, edited by U. Kulisch, pp. 167–181, Vieweg, Braunschweig, 1989.

[141] D. Cordes and W. Krämer, *PASCAL-XSC Modules for Multiple-Precision Operations and Functions*, Universität Karlsruhe, 1991.

[142] G. F. Corliss, Computing narrow inclusions for definite integrals, in: [270], pp. 150–169, 1987.

[143] G. F. Corliss, Applications of differentiation arithmetic, in: [426], pp. 127–148, 1988.

[144] G. F. Corliss, Industrial applications of interval techniques, in: [591], pp. 91–113, 1990.

[145] G. F. Corliss, Automatic differentiation bibliography, in: [194], pp. 331–353, 1991 (see also [147]).

[146] G. F. Corliss, Validated anti-derivatives, in: *Computer Aided Proofs in Analysis*, edited by R. K. Meyer and D. S. Schmidt, IMA Volumes in Mathematics and Its Applications 28, Springer, New York, 1991.

[147] G. F. Corliss (ed.), *Automatic Differentiation Bibliography*, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, Illinois, Report ANL/MCS-TM-167 (30 pages), July 1992.

[148] G. F. Corliss, *Intorduction to Validated ODE Solving*, Technical Report No. 416, Marquette University, Milwaukee, Wisconsin, March 1995.

[149] G. F. Corliss and L. B. Rall, *Adaptive, Self-Validating Numerical Quadrature*, MRC Technical Summary Report # 2815, University of Wisconsin, Madison, 1985.

[150] G. F. Corliss and L. B. Rall, Computing the range of derivatives, in: [271], pp. 195–212, 1991.

[151] H. Cornelius and R. Lohner, Computing the range of values of real functions with accuracy higher than second order, *Computing* 33 (1984), 331–347.

[152] H. Cornelius and R. Lohner, Enclosing the range of values of real functions, *ZAMM* 64 (1984), T408–T410.

[153] T. Csendes (ed.), *Developments in Reliable Computing*, Kluwer Academic Publishers, 1999.

[154] L. Dadda, Some schemes for parallel multipliers, *Alta Frequenza* 34 (1965), 346–356.

[155] Ph. J. Davis and Ph. Rabinowitz, *Methods of Numerical Integration*, Academic Press, San Diego, 1984.

[156] T. J. Dekker, A floating-point technique for extending the available precision, *Numerical Mathematics* 18 (1971), 224–242.

[157] J. Demmel and Y. Hida, A floating-point technique for extending the available precision, *SIAM J. Sci. Comput.* 25 (2003), 1214–1248.

[158] St. Dietrich, *Adaptive verifizierte Lösung gewöhnlicher Differentialgleichungen*, Dissertation, Universität Karlsruhe, 2002.

[159] H.-J. Dobner, *Einschließungsalgorithmen für hyperbolische Differentialgleichungen*, Dissertation, Universität Karlsruhe, 1986.

[160] W. E. Egbert, *Personal Calculator Algorithms I, II, III and IV*, Hewlett-Packard Journal, Mai 1977, Juni 1977, November 1977 und April 1978.

[161] P. Eijgenraam, *The Solution of Initial Value Problems Using Interval Arithmetic. Formulation and Analysis of an Algorithm*, Dissertation, Mathematisch Centrum, Amsterdam, 1981.

[162] B. Einarsson (ed.), *Accuracy and Reliability in Scientific Computing*, SIAM, 2005.

[163] H. Engels, *Numerical Quadrature and Cubature*, Academic Press, New York, 1980.

[164] H. Erb, *Ein Gleitpunkt-Arithmetikprozessor mit mehrfacher Präzision zur verifizierten Lösung linearer Gleichungssysteme*, Dissertation, Universität Karlsruhe, 1992.

[165] A. Facius, Influence of rounding errors in solving large sparse linear systems, in: [153], pp. 17–30, Kluwer Academic Publishers, 1999.

[166] A. Facius, *Iterative Solution of Linear Systems with Improved Arithmetic and Result Verification*, Dissertation, Universität Karlsruhe, 2000.

[167] A. Facius, *Highly accurate verified error bounds for Krylov type linear system solvers*, in: [366], pp. 76–98, 2001.

[168] K. V. Fernando and M. W. Pont, Computing accurate eigenvalues of a hermitian matrix, in: [595], pp. 104–115, 1989.

[169] H. Fischer, Fast method to compute the scalar product of gradient and given vector, *Computing* 41 (1986), 261–265.

[170] H.-C. Fischer, *Schnelle automatische Differentiation, Einschliessungsmethoden und Anwendungen*, Dissertation, Universität Karlsruhe, 1990.

[171] H.-C. Fischer, Range computation and applications, in: [591], pp. 197–211, 1990.

[172] H.-C. Fischer, Effiziente Berechnung von Ableitungswerten, Gradienten und Taylorkoeffizienten, in: *Jahrbuch Überblicke Mathematik 1992*, edited by S. D. Chatterji, B. Fuchssteiner, U. Kulisch, R. Liedl and W. Purkert, pp. 59–73, Vieweg, Braunschweig, 1992.

[173] H.-C. Fischer, Automatic differentiation and applications, in: [5], pp. 105–142, 1993.

[174] H.-C. Fischer, Automatisches Differenzieren, in: [226], pp. 53–104, 1995.

[175] H.-C. Fischer, G. Schumacher and R. Haggenmüller, Evaluation of arithmetic expressions with guaranteed high accuracy, *Computing Supplementum* 6 (1988), 149–158.

[176] G. E. Forsythe, Pitfalls in computation, or why a math book isn't enough, *Amer. Math. Monthly* 9 (1970), 931.

[177] A. Frommer, *Lösung linearer Gleichungssysteme auf Parallelrechnern*, Vieweg, Braunschweig, 1990.

[178] A. Frommer, Asynchronous iterations for enclosing solutions of fixed point problems, in: [52], pp. 243–252, 1992.

[179] A. Frommer, Proving conjectures by use of interval arithmetic, in: [366], pp. 1–13, 2001.

[180] P. Gaffney and E. Houstis (eds.), *Programming Environments for High Level Scientific Problem Solving*, Proceedings of IFIP WG 2.5 conference, Karlsruhe, September 23–27, 1991, Elsevier Science Publishers, 1993.

[181] S. Gal, *Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance*, IBM Technical Report 88.153, 1985.

[182] S. Gal and B. Bachelis, *An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard*, IBM Technical Report 88.223, IBM Israel, Technion City, Haifa, Israel, 1988.

[183] J. Garloff, Interval mathematics. A bibliography, *Freib. Int.-Ber.* 6 (1985), 1–222.

[184] J. Garloff, Bibliography on interval mathematics. Continuation, *Freiburger Intervall-Berichte* 87:2 (1987), 1–50.

[185] A. Gienger, *Zur Lösungsverifikation bei Fredholmschen Integralgleichungen zweiter Art*, Dissertation, Universität Karlsruhe, 1997.

[186] F. Goerisch and Z. He, The determination of guaranteed bounds to eigenvalues with the use of variational methods I, in: [591], pp. 137–153, 1990 (Part II see [66]).

[187] D. Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys 23, No. 1, March 1991.

[188] G. H. Golub and C. F. van Loan, *Matrix Computations*, third edition, John Hopkins, Baltimore, 1995.

[189] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, SIAM, Philadelphia, 1997.

[190] R. T. Gregory and D. L. Karney, *A Collection of Matrices for Testing Computational Algorithms*, John Wiley & Sons, New York, 1969.

[191] A. Griewank, On automatic differentiation, in: *Mathematical Programming: Recent Developments and Applications*, edited by M. Iri and K. Tanabe, pp. 83–108, Kluwer Academic Publishers, 1989.

[192] A. Griewank, *Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation*, Preprint MCS-P228-0491, Argonne National Laboratory, Argonne, 1991.

[193] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers Appl. Math., 19. SIAM, Philadelphia, 2000.

[194] A. Griewank and G. Corliss (eds.), *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications*, Proceedings of Workshop on Automatic Differentiation at Breckenridge, SIAM, Philadelphia, 1991.

[195] K. Grüner, Fehlerschranken für lineare Gleichungssysteme, in: [11], pp. 47–55, 1977.

[196] K. Grüner, *Allgemeine Rechnerarithmetik und deren Implementierung*, Dissertation, Universität Karlsruhe, 1979.

[197] K. Grüner, Solving complex problems for polynomials and linear systems with verified high accuracy, in: [270], pp. 199–220, 1987.

[198] K. Grüner, *Solving the Complex Eigenvalue Problem with Verified High Accuracy*, ESPRIT project DIAMOND, Doc. No. 03/3-2/1/Kl.p, 1987.

[199] K. Grüner, Solving the complex algebraic eigenvalue problem with verified high accuracy, in: [595], pp. 59–78, 1989.

[200] W. Hahn and K. Mohr, *APL/PCXA. Erweiterung der IEEE Arithmetik für technisch wissenschaftliches Rechnen*, Hanser Verlag, München, 1989.

[201] H. Hamada, A new real number representation and its operations, in: [654], Vol. 8, pp. 153–157, 1987.

[202] R. Hammer, How reliable is the arithmetic of vector computers?, in: [592], pp. 467–482, 1990.

[203] R. Hammer, *Maximal genaue Berechnung von Skalarproduktausdrücken und hochgenaue Auswertung von Programmteilen*, Dissertation, Universität Karlsruhe, 1992.

[204] R. Hammer, PASCAL-XSC: From accurate expressions to the accurate evaluation of program parts, in: [52], pp. 119–128, 1992.

[205] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *Numerical Toolbox for Verified Computing I: Basic Numerical Problems*, Springer, Berlin Heidelberg New York, 1993.

[206] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *C++ Toolbox for Verified Computing: Basic Numerical Problems*. Springer, Berlin Heidelberg New York, 1995.

[207] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *Numerical Toolbox for Verified Computing I: Basic Numerical Problems*, MIR, Moskau, 2005 (in Russian).

[208] R. Hammer, M. Neaga and D. Ratz, PASCAL-XSC. New concepts for scientific computation and numerical data processing, in: [5], pp. 15–44, 1993.

[209] E. R. Hansen, *Topics in Interval Analysis*, Clarendon Press, Oxford, 1969.

[210] E. R. Hansen, The centered form, in: [209], pp. 102–106, 1969.

[211] E. R. Hansen, An overview of global optimization using interval analysis, in: [426], pp. 289–307, 1988.

[212] E. R. Hansen, *Global Optimization Using Interval Analysis*, Marcel Dekker Inc., New York Basel Hong Kong, 1992.

[213] E. R. Hansen and G. W. Walster, *Global Optimization Using Interval Analysis*, second edition, Chapman and Hall, London, 2004.

[214] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. C. Thacher Jr. and C. Witzgall, *Computer Approximations*, Wiley, New York London Sydney, 1968.

[215] G. Heindl, Inclusions of the range of functions and their derivatives, in: [271], pp. 229–238, 1991.

[216] G. Heindl, *An improved algorithm for computing the product of two machine intervals*, Report IAGMPI - 9314, Universität Wuppertal, 1983.

[217] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, third edition, Elsevier Science, San Francisco, CA, 2003.

[218] A. Hergenhan, *Spezifikation und Entwurf einer hochleistungsfähigen Gleitkomma-Architektur*, Diplomarbeit, Technische Universität Dresden, 1994.

[219] P. Hertling, *A Lower Bound for Range Enclosure in Interval Arithmetic*, Centre for Discrete Mathematics and Theoretical Computer Science Research Report Series, Department of Computger Science, University of Auckland, January 1998.

[220] J. Herzberger, *Metrische Eigenschaften von Mengensystemen und einige Anwendungen*, Dissertation, Universität Karlsruhe, 1969.

[221] J. Herzberger, Intervallmäßige Auswertung von Standardfunktionen in ALGOL-60, *Computing* 5 (1970), 377–384.

[222] J. Herzberger, On Schulz's method in circular complex arithmetic, in: [592], pp. 93–102, 1990.

[223] J. Herzberger, Iterative inclusion of the inverse matrix, in: [49], pp. 14–26, 1991.

[224] J. Herzberger (ed.): *Topics in Validated Computations*, Proceedings of IMACS–GAMM International Workshop on Validated Numerics, Oldenburg 1993, Elsevier Science Publishers (North Holland), Amsterdam, 1994.

[225] J. Herzberger, Basic definitions and properties of interval arithmetic, in: [224], pp. 1–6, 1994.

[226] J. Herzberger, *Wissenschaftliches Rechnen. Eine Einführung in das Scientific Computing*, Akademie Verlag, Berlin, 1995.

[227] J. Herzberger and D. Bethke, Interval Schulz's method: On the case of an interval matrix, in: [52], pp. 199–203, 1992.

[228] J. Herzberger and Lj. Petković, Efficient iterative algorithms for bounding the inverse of a matrix, *Computing* 44 (1990), 237–244.

[229] N. J. Higham, The accuracy of floating-point summation, *SIAM J. Sci. Comput.* 14 (1993), 783–799.

[230] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, second edition, SIAM, Philadelphia, 2002.

[231] M. Hocks, *Innere-Punkt-Methoden und automatische Ergebnisverifikation in der Linearen Optimierung*, Dissertation, Universität Karlsruhe, 1995.

[232] B. Hoefflinger, Next-generation floating-point arithmetic for top-performance PCs, in: *Conference Proceedings of the 1995 Silicon Valley Personal Computer Design Conference and Exposition*, pp. 319–325, 1995.

[233] T. Hoff, How children accumulate numbers or why we need a fifth floating-point operation, in: *Jahrbuch Überblicke Mathematik*, pp. 219–222, Vieweg, Braunschweig Wiesbaden, 1993.

[234] W. Hofschuster, *Zur Berechnung von Funktionseinschließungen bei speziellen Funktionen der mathematischen Physik*, Dissertation, Universität Karlsruhe, 2000.

[235] W. Hofschuster, F. Blomquist and W. Krämer, *Vermeidung von Über- und Unterlauf und Verbesserung der Genauigkeit bei reeller und komplexer staggered Intervall-Arithmetik*, preprint BUW-WRSWT 2007, Universität Wuppertal, 2007.

[236] W. Hofschuster and W. Krämer, *Rechnergestütztes Fehlerkalkül mit Anwendung auf ein genaues Tabellenverfahren*, preprint des IWRMM, Universität Karlsruhe, 1996.

[237] W. Hofschuster and W. Krämer, A computer oriented approach to get sharp reliable error bounds, *Reliable Computing* 3 (1997), 239–248.

[238] W. Hofschuster and W. Krämer, C-XSC 2.0 – A C++ class library for extended scientific computing, in: *Numerical Software with Result Verification*, edited by R. Alt, A. Frommer, B. Kearfott and W. Luther, pp. 15–35, Springer Lecture Notes in Computer Science 2991, 2004. For more information see: `http://www.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[239] W. Hofschuster, W. Krämer, M. Lerch, G. Tischler and J. Wolff von Gudenberg, $filib++$ *a Fast Interval Library Supporting Containment Computations*, TOMS, to appear.

[240] W. N. Holmes, *Computers and People*, Wiley & Sons, Hoboken, New Jersey, 2006.

[241] W. N. Holmes, Binary arithmetic, in: *Computer*, Vol. 40, pp. 1–4, IEEE Computer Society, 2007.

[242] T. E. Hull and A. Abrham, Properly rounded variable precision square root, *ACM Trans. on Math. Software* 11:3 (1985), 229–237.

[243] T. E. Hull, A. Abrham, M. S. Cohen, A. F. X. Curley, D. A. Penny and J. T. M. Sawchuk, Numerical turing, *ACM SIGNUM Newsletter* 20:3 (1985), 26–34.

[244] M. Iri, Simultaneous computation of functions, partial derivatives and estimates for rounding errors – complexity and practicality, *Japan Journal of Applied Mathematics* 1 (1984), 223–252.

[245] K.-U. Jahn (ed.), *Computernumerik mit Ergebnisverifikation*, Problemseminar, Technische Hochschule Leipzig, 13.–15. März 1991. Proceedings in Wissenschaftliche Zeitschrift der Technischen Hochschule Leipzig, Jahrgang 15, Heft 6, 1991.

[246] C. Jansson, *Zur linearen Optimierung mit unscharfen Daten*, Dissertation, Universität Kaiserslautern, 1985.

[247] C. Jansson, Guaranteed error bounds for the solution of linear systems, in: [592], pp. 103–110, 1990.

[248] C. Jansson, *A Fast Direct Method for Computing Verified Inclusions*, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Technische Universität Hamburg-Harburg, Bericht 90.4, 1990.

[249] C. Jansson, Interval linear systems with symmetric matrices, skew-symmetric matrices, and dependencies in the right-hand side, *Computing* 46 (1991), 265–274.

[250] C. Jansson, A geometric approach for computing a posteriori error bounds for the solution of a linear system, *Computing* 47 (1991), 1–9.

[251] C. Jansson, On self-validating methods for optimization problems, in: [224], pp. 381–438, 1994.

[252] C. Jansson, A branch-and-bound algorithm for bound constrained optimization problems without derivatives, *J. Glob. Optim.* 7:3 (1995), 297–331.

[253] C. Jansson, Convex-concave extensions, *BIT* 402 (2000), 291–313.

[254] C. Jansson, Quasiconvex relaxations based on interval arithmetic, *Lin. Alg. Appl.* 324 (2001), 27–53.

[255] C. Jansson and O. Knüppel, A branch-and-bound algorithm for bound constrained optimization problems without derivatives, *J. Glob. Optim.* 7:3 (1995), 297–331.

[256] C. Jansson and S. M. Rump, Rigorous sensitivity analysis for real symmetric matrices with uncertain data, in: [271], pp. 293–316, 1991.

[257] P. Januschke, *Oberon-XSC, Eine Programmiersprache für das wissenschaftliche Rechnen*, Dissertation, Universität Karlsruhe, 1998.

[258] K. Jensen and N. Wirth, *Pascal User Manual and Report*, ISO Pascal Standard, third edition, Springer, Berlin Heidelberg New York, 1985.

[259] D. W. Juedes, A taxonomy of automatic differentiation tools, in: *Automatic Differentiation of Algorithms: Theory, Implementation and Applications*, edited by A. Griewank and G. F. Corliss, pp. 315–329, SIAM, Philadelphia, Penn., 1991.

[260] W. Kahan, Further remarks on reducing truncation errors, *Comm. ACM* 8:1 (1965), 40.

[261] W. Kahan, *A More Complete Interval Arithmetic*, Lecture Notes prepared for a summer course at the University of Michigan, June 17–21, 1968.

[262] W. Kahan, A survey on error analysis, in: *Proceedings of the IFIP Congress*, pp. 1214–1239, Information Processing 71, North-Holland, Amsterdam, 1972.

[263] W. Kahan, Branch cuts for complex elementary functions, in: *The State of the Art in Numerical Analysis*, edited by A. Iserles and M. J. D. Powell, pp. 165–211, Clarendon Press, Oxford, 1987.

[264] W. Kahan, *Doubled Precision IEEE Standard 754 Floating-Point Arithmetic*, Mini-Course on “The Regrettable Failure of Automated Error Analysis”, Conference on Computers and Mathematics, MIT, June 13, 1989.

[265] S. A. Kalmykov, J. I. Shokin and S. C. Juldashev, *Methods of Interval Analysis*, Novosibirsk, 1986 (in Russian).

[266] A. Karatsuba and Y. Ofman, Multiplication of multidigit numbers on automata, *Soviet Physics Dokl.* 7 (1963), 595–596.

[267] E. Kaucher, *Über metrische und algebraische Eigenschaften einiger beim numerischen Rechnen auftretender Räume*, Dissertation, Universität Karlsruhe, 1973.

[268] E. Kaucher, Algebraische Erweiterungen der Intervallrechnung unter Erhaltung der Ordnungs- und Verbandsstrukturen, in: *Grundlagen der Computerarithmetik*, edited by R. Albrecht and U. Kulisch, pp. 65–79, Computing Supplementum 1, Springer, Wien New York, 1977.

[269] E. Kaucher, Über Eigenschaften und Anwendungsmöglichkeiten der erweiterten Intervallrechnung und des hyperbolischen Fastkörpers über $\mathbb{R}$, in: *Grundlagen der Computerarithmetik*, edited by R. Albrecht and U. Kulisch, pp. 81–94, Computing Supplementum 1, Springer, Wien New York, 1977.

[270] E. Kaucher, U. Kulisch and Ch. Ullrich (eds.), *Computerarithmetic: Scientific Computation and Programming Languages*, B. G. Teubner, Stuttgart, 1987.

[271] E. Kaucher, G. Mayer and S. M. Markov (eds.), *Computer Arithmetic, Scientific Computation and Modelling*, Proceedings of SCAN-90, IMACS Annals on Computing and Applied Mathematics 12, 1992, J. C. Baltzer AG, Basel, 1991.

[272] E. Kaucher and W. L. Miranker, *Self-Validating Numerics for Function Space Problems, Computations with Guarantees for Differential and Integral Equations*, Academic Press, Orlando, 1984.

[273] R. B. Kearfott, A review of techniques in the verified solution of constrained global optimization problems, in: [276], pp. 23–59, 1996.

[274] R. B. Kearfott, *A specific proposal for interval arithmetic in Fortran*, March 1996, available from `http://interval.usl.edu/F90/f96-pro.asc`.

[275] R. B. Kearfott, K. D. Dawande and C. Hu, Algorithm 737: INTLIB: A portable Fortran 77 interval standard function library, *ACM Transactions on Mathematical Software* 20 (1994), 447–459.

[276] R. B. Kearfott and V. Kreinovich (eds.), *Applications of Interval Computations*, Kluwer, Dordrecht, 1996.

[277] R. Kelch, *Ein adaptives Verfahren zur numerischen Quadratur mit automatischer Ergebnisverifikation*, Dissertation, Universität Karlsruhe, 1989.

[278] R. Kelch, Self-validating numerical quadrature, in: [595], pp. 162–202, 1989.

[279] R. Kelch, An adaptive procedure for numerical quadrature with automatic result verification, in: [592], pp. 301–317, 1990.

[280] R. Kelch, Numerical quadrature by extrapolation with automatic result verification, in: [5], pp. 143–185, 1993.

[281] J. Kernhof, Ch. Baumhof, B. Höfflinger, U. Kulisch, S. Kwee, P. Schramm, M. Selzer and Th. Teufel, *A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic*, ESSIRC 94, Ulm, Sept. 1994.

[282] I. Kießling, M. Lowes and A. Paulik, *Genaue Rechnerarithmetik – Initervallrechnung und Programmieren mit PASCAL-SC*, B. G. Teubner, Stuttgart, 1988.

[283] R. Kirchner and U. Kulisch, Schaltungsanordnung und Verfahren zur schnellen Berechnung von Summen und Skalarprodukten von Gleitkommazahlen mit maximaler Genauigkeit mittels Pipelinetechnik, in: *Beiträge zur Angewandten Mathematik und Statistik*, edited by M. J. Beckmann, K. W. Gaede and K. Ritter, pp. 139–177, Hansa, München Wien, 1987.

[284] R. Kirchner and U. Kulisch, Arithmetic for vector processors, in: [654], Vol. 8, pp. 256–269, 1987.

[285] R. Kirchner and U. Kulisch, Accurate arithmetic for vector processors, *Journal of Parallel and Distributed Computing* 5 (1988), 250–270.

[286] R. Kirchner and U. Kulisch, Fast and accurate computation of sums and inner products, in: *Computational Techniques and Applications: CTAC–87*, edited by J. Noye and C. Fletcher, pp. 3–28, Proceedings of CTAC–87 in Sydney (Australia), North-Holland, 1988.

[287] R. Kirchner and U. Kulisch, Hardware support for interval arithmetic, *Reliable Computing* 12:3 (2006), 225–237.

[288] R. Klatte, U. Kulisch, C. Lawo, M. Rauch and A. Wiethoff, *C-XSC – A C++ Class Library for Extended Scientific Computing*, Springer, Berlin Heidelberg New York, 1993. See also `http://www.math.uni-wuppertal.de/~xsc/` ord `http://www.xsc.de/`.

[289] R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Sprachbeschreibung mit Beispielen*, Springer, Berlin Heidelberg New York, 1991. See also `http://www.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[290] R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Language Reference with Examples*, Springer, Berlin Heidelberg New York, 1992. See also `http://www.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[291] R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Language Reference with Examples*, MIR, Moskau, 1995, third edition 2006 (in Russian). See also `http://www.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[292] W. Klein, *Verified Solution of Linear Systems with Band-Shaped Matrices*, DIAMOND Workpaper, Doc. No. 03/3-3/3, January 1987.

[293] W. Klein, *Data Structure and Symbolic LU-Factorization for General Sparse Matrices*, DIAMOND Workpaper, Doc. No. 03/3-7/3, March 1987.

[294] W. Klein, Verified results for linear systems with sparse matrices, in: [595], pp. 137–161, 1989.

[295] W. Klein, *Zur Einschließung der Lösung von linearen und nichtlinearen Fredholmschen Integralgleichungssystemen zweiter Art*, Dissertation, Universität Karlsruhe, 1990.

[296] G. Klotz, *Faktorisierung von Matrizen mit maximaler Genauigkeit*, Dissertation, Universität Karlsruhe, 1987.

[297] U. Klug, Verified inclusions for eigenvalues and eigenvectors of real symmetric matrices, in: [592], pp. 111–125, 1990.

[298] A. Knöfel, *Hardwareentwurf eines Rechenwerks für semimorphe Skalar- und Vektoroperationen unter Berücksichtigung der Anforderungen verifizierender Algorithmen*, Dissertation, Universität Karlsruhe, 1991.

[299] A. Knöfel, Advanced circuits for the computation of accurate scalar products, in: [271], pp. 63–67, 1991.

[300] A. Knöfel, Fast hardware units for the computation of accurate dot products, in: [654], Vol. 10, pp. 70–74, 1991.

[301] A. Knöfel, A hardware kernel for scientific/engineering computations, in: [5], pp. 549–570, 1993.

[302] O. Knüppel, PROFIL/BIAS – a fast interval library, *Computing* 53 (1994), 277–287.

[303] D. E. Knuth, Euler's constant to 1271 places, *Math. Comp.* 16 (1962), pp. 275–281.

[304] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, second edition, Addison-Wesley, Reading, Massachusetts, 1981.

[305] M. Koeber, *Lösungseinschließung bei Anfangswertproblemen für quasilieare hyperbolische Differentialgleichungen*, Dissertation, Universität Karlsruhe, 1997.

[306] R. Kolla, A. Vodopivec and J. Wolff von Gudenberg, *The IAX Architecture – Interval Arithmetic Extension*, Report No. 225, Institut für Informatik, Universität Würzburg, 1999.

[307] R. Kolla, A. Vodopivec and J. Wolff von Gudenberg, Splitting double precision FPUs for single precision interval arithmetic, in: *ARCS'99 Workshops zur Architektur von Rechensystemen*, edited by W. Erhard, K.-E. Großpietsch, W. Koch, E. Maehle and E. Zehendner, pp. 5–16, Universität Jena, 1999.

[308] S. König, On the inflation parameter used in self-validating methods, in: [592], pp. 127–132, 1990.

[309] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[310] M Koshelev and V. Kreinovich, *Interval Computations*, 1997, available from `http://cs.utep.edu/interval-comp.html`.

[311] W. Krämer, *Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate*, Dissertation, Universität Karlsruhe, 1987.

[312] W. Krämer, Inverse standard functions for real and complex point and interval arguments with dynamic accuracy, *Computing Supplementum* 6 (1988), 185–211.

[313] W. Krämer, Mehrfachgenaue reelle und intervallmäßige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen, in: *Bericht des Instituts für Angewandte Mathematik*, pp. 1–80, Universität Karlsruhe, 1988.

[314] W. Krämer, Fehlerschranken für häufig auftretende Approximationsausdrücke, *ZAMM* 69 (1989), T44–T47.

[315] W. Krämer, *Genaue Auswertung von Polynomen in mehreren Variablen*, DFG-Bericht zum Forschungsvorhaben Ku 155/12-1, 1989.

[316] W. Krämer, Berechnung der Gammafunktion $\Gamma(X)$ für reelle Punkt- und Intervallargumente, *ZAMM* 70 (1990), T581–T584.

[317] W. Krämer, Highly accurate evaluations of program parts with applications, in: *Contributions to Computer Arithmetic and Self-Validating Numerical Methods*, edited by C. Ullrich, pp. 397–409, J. C. Baltzer AG, Scientific Publishing Co., IMACS, 1990.

[318] W. Krämer, Computation of verified bounds for elliptic integrals, in: *Proceedings of the International Symposium on Computer Arithmetic and Scientific Computation, Oldenburg 1991 (SCAN91)*, edited by J. Herzberger and L. Atanassova, Elsevier Science Publishers (North-Holland), Amsterdam, 1992.

[319] W. Krämer, Evaluation of polynomials in several variables with high accuracy, in: *Computer Arithmetic, Scientific Computation and Mathematical Modelling*, edited by E. Kaucher, S. Markov and G. Mayer, pp. 239–249, I. C. Baltzer AG, Scientific Publishing Co., IMACS, 1991.

[320] W. Krämer, Einschluß eines Paares konjugiert komplexer Nullstellen eines reellen Polynoms, *ZAMM* 71 (1991), T820–T824.

[321] W. Krämer, Genaue Berechnung von komplexen Polynomen in mehreren Variablen, in: [245], pp. 401–407, 1991.

[322] W. Krämer, Verified solution of eigenvalue problems with sparse matrices, in: [600], pp. 32–33, 1991.

[323] W. Krämer, *PASCAL-XSC Module for Multiple-Precision Interval Operations and Functions*, Universität Karlsruhe, 1991.

[324] W. Krämer, Die Berechnung von Standardfunktionen in Rechenanlagen, in: *Jahrbuch Überblicke Mathematik*, edited by S. D. Chatterji, B. Fuchssteiner, U. Kulisch, R. Liedl and W. Purkert, pp. 97–115, Vieweg, Braunschweig, 1992.

[325] W. Krämer, Eine portable Langzahl- und Langzahlintervallarithmetik mit Anwendungen, *ZAMM* 73 (1992), T849–T853.

[326] W. Krämer, Verified solution of eigenvalue problems with sparse matrices, in: *Computational and Applied Mathematics I. Algorithms and Theory*, edited by C. Brezinsky, pp. 1–11, Proceedings of the 13th IMACS World Congress in Dublin (Ireland), Elsevier Science Publishers B. V., Amsterdam, 1992.

[327] W. Krämer, Multiple-precision computations with result verification, in: [5], pp. 325–356, 1993.

[328] W. Krämer, Constructive error analysis, *Journal of Universal Computer Science* 4:2 (1998), 147–163.

[329] W. Krämer and A. Bantle, Automatic forward error analysis for floating-point algorithms, *Reliable Computing* 7:4 (2001), 321–340.

[330] W. Krämer and B. Barth, Computation of interval bounds for Weierstrass' elliptic function, in: [10], pp. 147–159, 1993.

[331] W. Krämer, F. Blomquist and W. Hofschuster, *Vermeidung von Über- und Unterlauf und Verbesserung der Genauigkeit bei reeller und komplexer staggered Intervall-Arithmetik*, preprint BUW-WRSWT 2007, Universität Wuppertal, 2007.

[332] W. Krämer, U. Kulisch and R. Lohner, *Numerical Toolbox for Verified Computing II: Theory, Algorithms and Pascal-XSC Programs*, electronically available via `Rudolf.Lohner@math.uka.de` (Vol. I see [205, 206]).

[333] W. Krämer and W. Walter, *FORTRAN-SC: A FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH, General Information Notes and Sample Programs*, 1–51, IBM Deutschland GmbH, Stuttgart, 1989.

[334] W. Krämer and J. Wolff von Gudenberg (eds.), *Scientific Computing, Validated Numerics, Interval Methods*, Kluwer Acadamic Publishers, New York Boston Dordrecht London Moscow, 2001.

[335] W. Krämer and J. Wolff von Gudenberg, *Extended Interval Power Function*, Proceedings of Validated Computing, Reliable Computing, 2003.

[336] R. Krawczyk, Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken, *Computing* 4 (1969), 187–201.

[337] R. Krawczyk and A. Neumaier, Interval slopes for rational functions and associated centered forms, *SIAM Journal on Numerical Analysis* 22 (1985), 604–616.

[338] V. Kreinovich, A. Lakeyev, J. Rohn and P. Kahl, *Computaional Complexity and Feasibility of Data Processing and Interval Computations*, Kluwer, Dordrecht, 1998.

[339] V. Kreinovich and J. Wolff von Gudenberg, An optimality criterion for arithmetic on complex sets, *Geombinatorics* X:1 (2000), 31–37.

[340] V. Kreinovich and J. Wolff von Gudenberg, A full function-based calculus of directed and undirected intervals: Markov's interval arithmetic revisited, *Numerical Algorithms* 37:1–4 (2004), 417–428.

[341] F. Krückeberg, Ordinary differential equations, in: [209], 1969.

[342] F. Krückeberg, Arbitrary accuracy with variable precision arithmetic, in: [453], pp. 95–101, 1985.

[343] K. Kubota and M. Iri, PADRE2 – a FORTRAN precompiler yielding error estimates and second derivatives, in: [194], pp. 251–262, 1991.

[344] O. Kühn, Rigorously computed orbits of dynamical systems without the wrapping effect, *Computing* 61 (1998), 47–67.

[345] O. Kühn, Towards an optimal control of the wrapping effect, in: [153], pp. 43–51, 1999.

[346] U. Kulisch, Grundzüge der Intervallrechnung, in: *Jahrbuch Überblicke Mathematik 2*, pp. 51–98, Bibliographisches Institut, Mannheim, 1969.

[347] U. Kulisch, An axiomatic approach to rounded computations, TS Report No. 1020, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, 1969, and *Numerische Mathematik* 19 (1971), 1–17.

[348] U. Kulisch, On the concept of a screen, Mathematics Research Center, The University of Wisconsin, Madison, Wisconsin, Technical summary Report Nr. 1084, 1–12, 1970, and *ZAMM* 53 (1973), 115–119.

[349] U. Kulisch, Rounding invariant structures, Mathematics Research Center, The University of Wisconsin, Madison, Wisconsin, Technical Summary Report Nr. 1103, 1970, 1–47.

[350] U. Kulisch, Interval arithmetic over completely ordered ringoids, Mathematics Research Center, The University of Wisconsin, Madison, Wisconsin, Technical Summary Report Nr. 1105, 1–56, 1970.

[351] U. Kulisch, Implementation and formalization of floating-point arithmetics, IBM T. J. Watson-Research Center, Report Nr. RC 4608, 1–50, 1973. Invited talk at the Caratheodory Symposium, September 1973 in Athens, published in: The Greek Mathematical Society, C. Caratheodory Symposium, 328–369, 1973, and in *Computing* 14 (1975), 323–348.

[352] U. Kulisch, Über die Arithmetik von Rechenanlagen, in: *Jahrbuch Überblicke Mathematik 1975*, pp. 69–198, Wissenschaftsverlag, Bibliographisches Institut, Mannheim Wien Zürich, 1975.

[353] U. Kulisch, *Grundlagen des Numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik*, Informatik 19, Bibliographisches Institut, Mannheim Wien Zürich, 1976.

[354] U. Kulisch, *Mathematical Foundation of Computer Arithmetic*, 3rd Symposium on Computer Arithmetic of the IEEE Computer Society in Dallas/Texas, 1–13, published in Transactions on Computers, Vol. C-26, 610–621, 1977.

[355] U. Kulisch (ed.), *PASCAL-SC: A PASCAL Extension for Scientific Computation, Information Manual and Floppy Disks, Version IBM PC/AT, Operating System DOS*, Wiley-Teubner series in computer science, B. G. Teubner Verlag, Stuttgart, 1987.

[356] U. Kulisch (ed.), *PASCAL-SC: A Pascal extension for Scientific Computation*, Information Manual and Floppy Disks. Version ATARI ST., B. G. Teubner, Stuttgart, 1987.

[357] U. Kulisch (ed.), *Wissenschaftliches Rechnen mit Ergebnisverifikation – Eine Einführung*, Vorträge einer Tagung in Karlsruhe 1988, ausgearbeitet von S. Geörg, R. Ham-

mer und D. Ratz, Vol. 58, Akademie Verlag, Berlin, und Vieweg Verlagsgesellschaft, Wiesbaden, 1989.

[358] U. Kulisch, *Numerik mit automatischer Ergebnisverifikation*, Jahrbuch Überblicke Mathematik 1993, pp. 199–218, Vieweg Verlag, 199, and *GAMM-Mitteilungen* 1 (1994), 39–58.

[359] U. Kulisch, How children accumulate numbers – or: Why we need a fifth floating-point operation, in: *Jahrbuch Überblicke Mathematik 1993*, pp. 219–222, Vieweg, 1993.

[360] U. Kulisch, Memorandum über Computer, Arithmetik und Numerik. Anläßlich des 85. Geburtstages von K. Zuse, in: *Jahrbuch Überblicke Mathematik*, pp. 1–59, Vieweg, Braunschweig Wiesbaden, 1995.

[361] U. Kulisch, Numerical algorithms with automatic result verification, in: *The Mathematics of Numerical Algorithms*, edited by J. Renegar, M. Shub and S. Smale, pp. 471–502, Lectures in Applied Mathematics 32, AMS, 1996.

[362] U. Kulisch, Advanced arithmetic for the digital computer – design of arithmetic units, in: *Electronic Notes in Theoretical Computer Science*, pp. 1–72, Elsevier, Amsterdam, 1999. `http://www.elsevier.nl/locate/entcs/volume24.html`.

[363] U. Kulisch, Advanced arithmetic for the digital computer – interval arithmetic revisited, in: [366], pp. 15–75, 2001.

[364] U. Kulisch, *Advanced Arithmetic for the Digital Computer – Design of Arithmetic Units*, Springer, Wien New York, 2002.

[365] U. Kulisch and R. Kirchner, Arithmetic for vector processors, in: *Proceedings of the 8th Symposium on Computer Arithmetic of the IEEE Computer Society, Como, Italy*, pp. 256–269, IEEE Computer Society Press, Washington, D.C., 1987.

[366] U. Kulisch, R. Lohner and A. Facius (eds.), *Perspectives on Enclosure Methods*, Springer, Wien New York, 2001.

[367] U. Kulisch and W. L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, New York, 1981.

[368] U. Kulisch and W. L. Miranker (eds.), *A New Approach to Scientific Computation*, Proceedings of Symposium held at IBM Research Center, Yorktown Heights, N.Y., 1982, Academic Press, New York, 1983.

[369] U. Kulisch and W. L. Miranker, The arithmetic of the digital computer: A new approach, IBM Research Center RC 10580, 1–62, 1984, *SIAM Review* 28:1 (1986), 1–40.

[370] U. Kulisch and L. B. Rall, Numerics with automatic result verification, *Mathematics and Computers in Simulation* 35 (1993), 435–450.

[371] U. Kulisch, S. M. Rump and J. Wolff von Gudenberg, *Accuracy of the IBM /370 floating-point arithmetic and possible improvements*, Technical Report, IBM, 1982.

[372] U. Kulisch and H. J. Stetter (eds.), *Scientific Computation with Automatic Result Verification*, Computing Supplementum 6, Springer, Wien New York, 1988.

[373] U. Kulisch, T. Teufel and B. Hoefflinger, Genauer und trotzdem schneller: Ein neuer Coprozessor für hochgenaue Matrix- und Vektoroperationen. Titelgeschichte, *Elektronik* 26 (1994), 52–56.

[374] U. Kulisch and Ch. Ullrich (eds.), *Wissenschaftliches Rechnen und Programmiersprachen*, Berichte des German Chapter of the ACM 10, B. G. Teubner, Stuttgart, 1982.

[375] J.-R. Lahmann, *Eine Methode zur Einschließung von Eigenpaaren nichtselbstadjungierter Eigenwertprobleme und ihre Anwendung auf die Orr–Sommerfeld-Gleichung*, Dissertation, Universität Karlsruhe, 1999.

[376] J.-R. Lahmann and M. Plum, A computer-assisted instability proof for the Orr–Sommerfeld equation with Blasius profile, *ZAMM* 84:3 (2004), 188–204.

[377] Ch. Lawo, C-XSC – A programming environment for extended scientific computation, in: [600], p. 34, 1991.

[378] Ch. Lawo, *C-XSC. Eine Programmierumgebung für verifiziertes Rechnen in C++*, Talk on SCAN'91 conference, Oldenburg, 1991.

[379] Ch. Lawo, *C-XSC. Eine objektorientierte Programmierumgebung für verifiziertes wissenschaftliches Rechnen*, Dissertation, Universität Karlsruhe, 1992.

[380] Ch. Lawo, C-XSC. A programming environment for verified scientific computing and numerical data processing, in: [5], pp. 71–86, 1993.

[381] M. Lerch and J. Wolff von Gudenberg, Expression templates for dotproduct expressions, Interval 98, *Reliable Computing* 5:1 (1999), 69–80.

[382] M. Lerch and J. Wolff von Gudenberg, Implementation and test of a library for extended interval arithmetic, in: *RNC4 Fourth Real Numbers and Computers*, edited by P. Kornerup, pp. 111–124, Online Proceedings RNC4, 2000.

[383] H. Leuprecht and W. Oberaigner, Parallel algorithms for the rounding exact summation of floating-point numbers, *Computing* 28 (1982), 89–104.

[384] P. Lichter, *Realisierung eines VLSI-Chips für das Gleitkomma-Skalarprodukt der Kulisch-Arithmetik*, Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, 1988.

[385] S. Linnainmaa, Analysis of some known methods of improving the accuracy of floating-point sums, *BIT* 14 (1974), 167–202.

[386] S. Linnainmaa, Software for double-precision floating-point computations, *ACM Trans. on Math. Software* 7:3 (1981), 272–283.

[387] R. Lohner, Enclosing the solutions of ordinary initial and boundary value problems, in: [270], pp. 255–286, 1987.

[388] R. Lohner, *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen*, Dissertation, Universität Karlsruhe, 1988.

[389] R. Lohner, Precise evaluation of polynomials in several variables, in: [372], pp. 139–148, 1988.

[390] R. Lohner, Enclosing all eigenvalues of symmetric matrices, in: [595], pp. 87–103, 1989.

[391] R. Lohner, Computation of guaranteed enclosures for the solutions of ordinary initial and boundary value problems, in: *Computational Ordinary Differential Equations*, edited by J. R. Cash and I. Gladwell, pp. 425–435, Clarendon Press, Oxford, 1992.

[392] R. Lohner, Interval arithmetic in staggered correction format, in: [5], pp. 301–321, 1993.

[393] R. Lohner and J. Wolff von Gudenberg, Complex interval division with maximum accuracy, in: *Proceedings of the 7th IEEE Symposium on Computer Arithmetic in Urbana (Illinois)*, pp. 332–336, IEEE Comp. Soc., 1985.

[394] B. Lortz, *Eine Langzahlarithmetik mit optimaler einseitiger Rundung*, Dissertation, Universität Karlsruhe, 1971.

[395] G. Ludyk, *CAE von dynamischen Systemen: Analyse, Simulation, Entwurf von Regelungssystemen*, Springer, Berlin, 1990.

[396] M. Malcolm, On accurate floating-point summation, *Comm. ACM* 14 (1971), 731–736.

[397] S. M. Markov, *Scientific Computation and Mathematical Modeeling*, DATECS Publishing, Sofia, 1993.

[398] P. W. Markstein, Computation of elementary functions on the IBM RISC system/6000 processor, *IBM Journal of Research and Development* 34:1 (1990), 111–119.

[399] G. Mayer, On the convergence of powers of interval matrices, *Linear Algebra Appl.* 58 (1984), 201–216.

[400] G. Mayer, On the vonvergence of powers of interval matrices (2), *Numer. Math.* 46 (1985), 69–83.

[401] G. Mayer, *Reguläre Zerlegungen und der Satz von Stein und Rosenberg für Intervallmatrizen*, Habilitationsschrift, Universität Karlsruhe, 1986.

[402] G. Mayer, Enclosing the solutions of linear equations by interval iterative processes, in: [372], pp. 47–58, 1988.

[403] G. Mayer, Grundbegriffe der Intervallrechnung, in: [357], pp. 101–117, 1989.

[404] G. Mayer, Old and new Aspects for the interval Gaussian algorithm, in: [271], pp. 329–349, 1991.

[405] G. Mayer, Some remarks on two interval-arithmetic modifications of the Newton method, *Computing* 48, pp. 125–128, 1992.

[406] G. Mayer, Enclosures for eigenvalues and eigenvectors, in: [52], pp. 49–67, 1992.

[407] G. Mayer, Result verification for eigenvectors and eigenvalues, in: [224], pp. 209–276, 1994.

[408] G. Mayer, Epsilon-inflation in verification algorithms, *J. Com. Appl. Math.* 43 (1995), 147–169.

[409] G. Mayer, Epsilon-inflation with contractive interval functions, *Appl. Math.* 43 (1998), 241–254.

[410] G. Mayer, Beiträge zur Intervallrechnung, in: *Lexikon der Mathematik*, Spektrum, Mannheim, 2000.

[411] G. Mayer and A. Frommer, A multisplitting method for verification and enclosure on a parallel computer, in: [592], pp. 483–497, 1990.

[412] G. Mayer and J. Rohn, On the applicability of the interval Gaussian algorithm, *Reliable Computing* 4 (1998), 205–222.

[413] G. Mayer and J. Wolff von Gudenberg, Stichwörter zur Intervallrechnung, in: *Lexikon der Mathematik*, Spektrum, Mannheim, 2000.

[414] J. Mayer, *A Crout ILU preconditioner with pivoting and row permutation*, preprint Nr. 05/04, Universität Karlsruhe, Institut für Wissenschaftliches Rechnen, 2005.

[415] O. Mayer, *Über die in der Intervallrechnung auftretenden Räume und einige Anwendungen*, Dissertation, Universität Karlsruhe, 1968.

[416] E. J. McShane and T. A. Botts, *Real Analysis*, Van Nostrand-Reinhold, Princeton, New Jersey, 1959.

[417] T. Meis, Brauchen wir eine Hochgenauigkeitsarithmetik?, in: *Elektronische Rechenanlagen*, pp. 19–23, Carl Hanser, München, 1987.

[418] R. K. Meyer and D. S. Schmidt (eds.), *Computer Aided Proofs in Analysis*, IMA Volumes in Mathematics and Its Applications 28, Springer, New York, 1991.

[419] C. Miranda, Un'osservazione su un teorema di Brouwer, *Bol. Un. Mat. Ital. Ser. II* 3 (1941), 5–7.

[420] W. L. Miranker and R. A. Toupin, *Accurate Scientific Computations*, Lecture Notes in Computer Science 235, Springer, Berlin, 1986 (Symposium Bad Neuenahr, Germany, 1985).

[421] O. Möller, Quasi double precision in floating-point addition, *BIT* 5 (1965), 37–50.

[422] R. E. Moore, *Interval Arithmetic and Automatic Error analysis in Digital Computing*, Thesis, Stanford University, October 1962.

[423] R. E. Moore, *Interval Analysis*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1966.

[424] R. E. Moore, *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, Pennsylvania, 1979.

[425] R. E. Moore, *Computational Functional Analysis*, Ellis Horwood, Chichester, 1985.

[426] R. E. Moore (ed.), *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, Perspectives in Computing 19, Proceedings of the Conference at Columbus in Ohio, September 8–11, 1987, Academic Press, San Diego, 1988.

[427] R. E. Moore, Variable precision computing. Strategies for improving efficiency, in: [600], pp. 73–74, 1991.

[428] V. Moynihan, Techniques for generating accurate eigensolutions in ADA, in: [595], pp. 79–86, 1989.

[429] F. Mráz, Nonnegative solutions of interval linear systems, in: [52], pp. 299–308, 1992.

[430] J.-M. Muller, *Discrete Basis and Computation of Elementary Functions*, IEEE Trans. on Computers, Vol. C-34, No. 9, 1985.

[431] M. Müller, *Entwicklung eines Chips für auslöschungsfreie Summation von Gleitkommazahlen*, Dissertation, Universität des Saarlandes, Saarbrücken, 1993.

[432] M. Müller, Ch. Rüb and W. Rülling, *Exact Addition of Floating Point Numbers*, Sonderforschungsbericht 124, FB 14, Informatik, Universität des Saarlandes, Saarbrücken, 1990.

[433] M. R. Nakao, A numerical approach to the proof of existence of solutions for elliptic problems, *Japan J. Appl. Math.* 5:2 (1988), 313–332.

[434] M. R. Nakao, State of the art for numerical computations with guaranteed accuracy, *Math. Japanese* 48 (1998), 323–338.

[435] M. R. Nakao, Y. Watanabe, N. Yamamoto and T. Nishida, Some computer assisted proofs for solutions of the heat convection problems, *Reliable Computing* 9 (2003), 359–372.

[436] M. Neaga, *Erweiterungen von Programmiersprachen für wissenschaftliches Rechnen und Erörterung einer Implementierung*, Dissertation, Universität Kaiserslautern, 1984.

[437] N. S. Nedialkov, *Computing Rigorous Bounds on the Solution of Initial Value Problem for an Ordinary Differential Equation*, Thesis, University of Toronto, Toronto, 1999.

[438] M. Neher, An enclosure method of the solution of linear ODEs with polynomial coefficients, *Numer. Funct. Anal. Optim.* 20 (1999), 779–803.

[439] M. Neher, Geometric series bounds for the local errors of Taylor methods for linear n-th order ODEs, in: [41], pp. 183–193, 2001.

[440] A. Neumaier, Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen, *Z. Angew. Math. Mech.* 54 (1974), 39–51.

[441] A. Neumaier, Overestimation in linear interval equations, *SIAM J. Numer. Anal.* 24:1 (1987), 207–214.

[442] A. Neumaier, The enclosure of solutions of parameter-dependent systems of equations, in: [426], pp. 269–286, 1988.

[443] A. Neumaier, Rigorous sensitivity analysis for parameter-dependent systems of equations, *J. Math. Anal. Appl.* 144 (1989), 16–25.

[444] A. Neumaier, *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, 1990.

[445] A. Neumaier, The wrapping effect, ellipsoid arithmetic, stability and confidence regions, in: [10], pp. 175–190, 1993.

[446] A. Neumaier and T. Rage, Rigorous chaos verification in discrete dynamical systems, *Physica D* 67 (1993), 327–346.

[447] J. von Neumann and H. H. Goldstine, Numerical inverting of matrices of high order, *Bulletin of the American Mathematical Society* 53:11 (1947), 1021–1099.

[448] K. C. Ng, *Argument Reduction for Huge Arguments: Good to the Last Bit*, Sun Pro, Sun Microsystems Inc., 1992.

[449] H. T. Nguyen, V. Kreinovich, S. Nesterov and M. Nakumura, On hardware support for interval computations and for soft computing: Theorems, *IEEE Transactions on Fuzzy Systems* 5:1 (1997), 108–127.

[450] K. Nickel, Das Kahan–Babuskasche Summierungsverfahren in Triplex-ALGOL 60, *Z. Angew. Math. Mech.* 50 (1970), 369–373.

[451] K. Nickel (ed.), *Interval Mathematics*, Proceedings of the International Symposium in Karlsruhe 1975, Springer, Wien, 1975.

[452] K. Nickel, *Interval Mathematics 1980*, Proceedings of the International Symposium in Freiburg 1980, Academic Press, New York, 1980.

[453] K. Nickel, *Interval Mathematics 1985*, Proceedings of the International Symposium in Freiburg 1985, Springer, Wien, 1986.

[454] W. Oettli and W. Prager, Computability of approximate solution of linear equations with given error bounds for coefficients and right-hand sides, *Numer. Math.* 6 (1964), 405–409.

[455] T. Ogita, S. M. Rump and S. Oishi, Accurate sum and dot product with applications, in: *Proceedings of the IEEE International Symposium on Computer Aided Control Systems Design, Taipei, 2004*, 152–155, 2004.

[456] T. Ogita, S. M. Rump and S. Oishi, Accurate sum and dot product, *SIAM Journal on Scientific Computing* 26:6 (2005), 1955–1988.

[457] T. Ohta, T. Ogita, S. M. Rump and S. Oishi, Numerical verification method for arbitrarily ill-conditioned linear systems, *Trans. JSIAM* 15:3 (2005), 269–287.

[458] S. Oishi and S. M. Rump, Fast verification of solutions of matrix equations, *Num. Math.* 90:4 (2002), 755–773.

[459] S. Oishi, K. Tanabe, T. Ogita and S. M. Rump, Convergence of Rump's method for inverting arbitrarily ill-conditioned matrices, *Journal of Computational and Applied Mathematics* 205 (2007), 533–544.

[460] J. M. Ortega and W. C. Rheinbold, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York London, 1970.

[461] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, Elsevier Science, San Francisco, 2005.

[462] M. Payne and R. Hanek, Radian reduction for trigonometric functions, *Signum* 18 (1983), 19–24.

[463] O. Perron, *Irrationalzahlen*, de Gruyter, Berlin, 1960.

[464] M. Petkovic, *Iterative Methods for Simultaneous Inclusion of Polynomial Zeros*, Lecture Notes in Mathematics 1387, Springer, Berlin, 1989.

[465] M. Petkovic and L. D. Petkovic, *Complex Interval Arithmetic and its Applications*, Wiley, New York, 1998.

[466] M. Pichat, Correction d'une somme en arithmétique à virgule flottante, *Numerische Mathematik* 19 (1972), 400–406.

[467] M. Plum, Computer-assisted existence proofs for two point boundary value problems, *Computing* 46 (1991), 19–34.

[468] M. Plum, Inclusion methods for elliptic boundary value problems, in: [224], pp. 323–379, 1994.

[469] M. Plum, Computer-assisted enclosure methods for elliptic differential equations, *Linear Algebra and its Applications* 324 (2001), 147–187.

[470] M. Plum, Safe numerical error bounds for solutions of nonlinear elliptic boundary value problems, in: [41], pp. 195–207, 2001.

[471] M. Plum and Ch. Wieners, New solutions for the Gelfand problem, *J. Math. Anal. Appl.* 269 (2002), 588–606.

[472] M. Plum and Ch. Wieners, Optimal a priori estimates for interface problems, *Num. Math.* 95 (2003), 735–759.

[473] D. M. Priest, Algorithms for arbitrary precision floating-point arithmetic, in: [654], Vol. 10, pp. 132–143, 1991.

[474] L. B. Rall, *Error in Digital Computation*, J. Wiley, New York, 1965.

[475] L. B. Rall, *Computaional Solution of Nonlinear Operator Equations*, Wiley, New York, 1969.

[476] L. B. Rall, Applications of software for automatic differentiation in numerical computation, in: [27], pp. 141–156, 1980.

[477] L. B. Rall, *Automatic Differentiation: Techniques and Applications*, Lecture Notes in Computer Science 120, Springer, Berlin, 1981.

[478] L. B. Rall, Differentiation and generation of Taylor coefficients in PASCAL-SC, in: [368], pp. 291–309, 1983.

[479] L. B. Rall, *Differentiation in PASCAL-SC: Type GRADIENT*, *ACM TOMS* 10 (1984), 161–184.

[480] L. B. Rall, Optimal implementation of differentiation arithmetic, in: [270], pp. 287–295, 1987.

[481] L. B. Rall, Differentiation arithmetics, in: [591], pp. 73–90, 1990.

[482] L. B. Rall, Tools for mathematical computation, in: *Computer Aided Proofs in Analysis*, edited by R. K. Meyer and D. S. Schmidt, pp. 217–228, IMA Volumes in Mathematics and Its Applications 28, Springer, New York, 1991.

[483] L. B. Rall and T. W. Reps, Algorithmic differencing, in: [366], pp. 133–147, 2001.

[484] H. Ratschek, Teilbarkeitskriterien der Intervallarithmetik, *J. Reine Ange. Math.* 252 (1971), 128–137.

[485] H. Ratschek, Centered forms, *SIAM J. Numer. Anal.* 17 (1980), 656–662.

[486] H. Ratschek and J. Rokne, *Computer Methods for the Range of Functions*, Ellis Horwood Limited, Chichester, 1984.

[487] H. Ratschek and J. Rokne, *New Computer Methods for Global Optimizaation*, Ellis Horwood, Chichester, 1988.

[488] H. Ratschek and J. Rokne, Nonuniform variable precision computing, in: [600], pp. 71–72, 1991.

[489] D. Ratz, The effects of the arithmetic of vector computers on basic numerical methods, in: [591], pp. 499–514, 1990.

[490] D. Ratz, Programmierpraktikum mit PASCAL-SC, in: *Computer Theoretikum und Praktikum für Physiker 5*, edited by G. Höhler and H. M. Staudenmaier, pp. 43–67, Fachinformationszentrum Karlsruhe, 1990.

[491] D. Ratz, *Globale Optimierung mit automatischer Ergebnisverifikation*, Dissertation, Universität Karlsruhe, 1992.

[492] D. Ratz, *Automatic Slope Computation and its Application in Nonsmooth Global Optimization*, Shaker, Aachen, 1998.

[493] D. Ratz, *On Extended Interval Arithmetic and Inclusion Isotony*, preprint, Institut für Angewandte Mathematik, Universität Karlsruhe, 1999.

[494] D. Ratz, Nonsmooth global optimization, in: [366], pp. 277–337, 2001.

[495] G. Reitwiesner, An ENIAC determination of π and e to more than 2000 decimal places, *MTAC* 4 (1950), 11–15.

[496] H.-G. Rex, Zur Lösungseinschließung linearer Gleichungssysteme, in: [245], pp. 441–447, 1991.

[497] R. Rihm, *Über Einschließungsverfahren für gewöhnliche Anfangswertprobleme und ihre Anwendung auf Differentialgleichungen mit unstetiger rechter Seite*, Dissertation, Universität Karlsruhe, 1993.

[498] R. Rihm, Interval methods for initial value problems in ODEs, in: [224], pp. 173–207, 1994.

[499] J. Rohn and A. Deif, On the range of eigenvalues of an interval matrix, *Computing* 47 (1992), 373–377.

[500] R. Rojas, Die Architektur der Rechenmaschinen Z1 und Z3 von Konrad Zuse, *Informatik Spektrum* 19:6 (1996), 303–315.

[501] R. Rojas, Konrad Zuses Rechenmaschinen: sechzig Jahre Computergeschichte, in: *Spektrum der Wissenschaft*, pp. 54–62, Spektrum Verlag, Heidelberg, 1997.

[502] D. R. Ross, Reducing trancation errors using cascading accumulators, *Comm. ACM* 8 (1965), 32–33.

[503] S. M. Rump, *Kleine Fehlerschranken bei Matrixproblemen*, Dissertation, Universität Karlsruhe, 1980.

[504] S. M. Rump, Rechnervorführung, Pakete für Standardprobleme der Numerik, in: [374], pp. 29–50, 1982.

[505] S. M. Rump, Lösung linearer und nichtlinearer Gleichungssysteme mit maximaler Genauigkeit, in: [374], pp. 147–174, 1982.

[506] S. M. Rump, Solving non-linear systems with least significant bit accuracy, *Computing* 29 (1982), 183–200.

[507] S. M. Rump, How reliable are results of computers? / Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?, in: *Jahrbuch Überblicke Mathematik*, pp. 163–168, Bibliographisches Institut, Mannheim, 1983.

[508] S. M. Rump, Solving algebraic problems with high accuracy, in: [368], pp. 51–120, 1983.

[509] S. M. Rump, Solving of linear and nonlinear algebraic problems with sharp guaranteed bounds, *Computing Suppl.* 5 (1984), 147–168.

[510] S. M. Rump, *Algorithms for Verified Inclusions: Theory and Practice*, Reliability in Computing, Academic Press, San Diego, 1988.

[511] S. M. Rump, Lineare Probleme, in: [357], pp. 119–127, 1989.

[512] S. M. Rump, Rigorous sensitivity analysis for systems of linear and nonlinear equations, *Math. of Comp.* 54:190 (1990), 724–736.

[513] S. M. Rump, Estimation of the sensitivity of linear and nonlinear algebraic problems, *Linear Algebra and its Applications* 153 (1991), 1–34.

[514] S. M. Rump, Convergence properties of iterations using sets, in: [245], pp. 427–431, 1991.

[515] S. M. Rump, On the solution of interval linear systems, *Computing* 47 (1992), 337–353.

[516] S. M. Rump, Inclusion of the solution of large linear systems with positive definite symmetric M-matrix, in: [52], pp. 339–347, 1992.

[517] S. M. Rump, Validated solution of large linear systems, in: [10], pp. 191–212, 1993.

[518] S. M. Rump, Verification methods for dense and sparse systems of equations, in: [224], pp. 63–135, 1994.

[519] S. M. Rump, A note on epsilon-inflation, *Reliable Computing* 4 (1998), 371–375.

[520] S. M. Rump, *INTLAB – Interval Laboratory*, TU Hamburg-Harburg, 1998.

[521] S. M. Rump, INTLAB–INTerval LABoratory, in: [153], pp. 77–104, 1999.

[522] S. M. Rump, Self-validating methods, *Linear Algebra Appl.* 324 (2001), 3–13.

[523] S. M. Rump, Computational error bounds for multiple or nearly multiple eigenvalues, *Linear Algebra Appl.* 324 (2001), 209–226.

[524] S. M. Rump, Fast verification algorithms in MATLAB, in: [41], pp. 209–226, 2001.

[525] S. M. Rump, Ten methods to bound multiple roots of polynomials, *J. Comput. Appl. Math.* 156 (2003), 403–432.

[526] S. M. Rump, Perron–Frobenius theory for complex matrices, *Linear Algebra and its Applications* 363 (2003), 251–273.

[527] S. M. Rump, *INTLAB – Interval Laboratory. A Matlab toolbox for verified computations, Version 5.1, 2005*, available at `http://www.ti3.tu-harburg.de/rump/intlab/index.html`.

[528] S. M. Rump, Computer-assisted proofs and self-validating methods, in: [162], pp. 195–239, 2005.

[529] S. M. Rump and H. Böhm, Least significant bit evaluation of arithmetic expressions, *Computing* 30 (1983), 189–199.

[530] S. M. Rump and E. Kaucher, Small bounds for the solution of linear systems, in: [27], pp. 157–164, 1980.

[531] S. M. Rump, T. Ogita and S. Oishi, *Accurate Floating-point Summation*, Technical Report 05.12, Faculty for Information and Communication Sciences, Hamburg University of Technology, November 13, 2005.

[532] M. Ruschitzka (ed.), *Computer Systems: Performance and Simulation*, in collaboration with R. Vichnevetsky, Proceedings of the 11th IMACS World Congress on System Simulation and Scientific Computation, August 5–9, 1985, Oslo. Preprints see [603]; additional papers in [270]. Elsevier Science Publishers B.V. (North–Holland), 1986.

[533] H. Rutishauser, A. Speiser and E. Stiefel, Programmgesteuerte digitale Rechengeräte, *Zeitschrift für Angewandte Mathematik und Physik* 1 (1950), 277–297.

[534] E. Salamin, Computation of π using arithmetic-geometric mean, *Mathematics of Computation* 30:135 (1976), 565–570.

[535] U. Schauer and R. A. Toupin, Solving large sparse linear systems with guaranteed accuracy, in: [420], pp. 142–167, 1986.

[536] L. Schmidt, *Semimorphe Arithmetik zur automatischen Ergebnisverifikation auf Vektorrechnern*, Dissertation, Universität Karlsruhe, 1992.

[537] P. Schramm, *Sichere Verschneidung von Kurven und Flächen im CAGD*, Dissertation, Universität Karlsruhe, 1995.

[538] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, New York, 1986.

[539] M. J. Schulte, *A Variable Precision, Interval Arithmetic Processor*, Ph.D. Thesis, University of Texas at Austin, 1996, available at `http//www.eecs.lehigh.edu/~mschulte/papers`.

[540] M. J. Schulte, E. Bickerstaff and E. E. Swartzlander Jr., Hardware interval multipliers, *Journal of Theoretical and Applied Informatics* 3:2 (1996), 73–90.

[541] M. J. Schulte and E. E. Swartzlander Jr., A hardware design and arithmetic algorithms for a vatiable precision, interval arithmetic coprocessor, in: *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 163–171, IEEE Computer Society Press, 1995.

[542] M. J. Schulte and E. E. Swartzlander Jr., A processor for staggered interval arithmetic, in: *Proceedings of the International Conference on Application Specific Array Processors*, pp. 104–112, IEEE Computer Society Press, 1995.

[543] M. J. Schulte and E. E. Swartzlander Jr., A software interface and hardware design for variable-precision interval arithmetic, *Reliable Computing* 1 (1995), 325–342 (software student prize).

[544] M. J. Schulte and E. E. Swartzlander Jr., Variable precision, interval arithmetic coprocessor, *Reliable Computing* 2 (1996), 47–62.

[545] M. J. Schulte and E. E. Swartzlander Jr., A family of variable-precision, interval arithmetic processor, *IEEE Transactions on Computers* 5:49 (2000), 387–398.

[546] M. J. Schulte, V. Zelov, A. Akkas and J. C. Burley, *Adding interval support to the GNU Fortran Compiler*, January 19, 1998, available at `http//www.eecs.lehigh.edu/~mschulte/compiler/work-notes`.

[547] M. J. Schulte, V. Zelov, A. Akkas and J. C. Burley, The interval-enhanced GNU Fortran compiler, in: [153], pp. 311–322, 1999.

[548] G. Schumacher, Einschließung der Lösung von linearen Gleichungssystemen auf Vektorrechnern, in: [357], pp. 239–249, 1989.

[549] G. Schumacher, *Genauigkeitsfragen bei algebraisch-numerischen Algorithmen auf Skalar- und Vektorrechnern*, Dissertation, Universität Karlsruhe, 1989.

[550] H. Schwandt, *Schnelle fast global konvergente Verfahren für die Fünf-Punkte-Diskretisierung der Poissongleichung mit Dirichletschen Randbedingungen auf Rechteckgebieten*, Dissertation, TU Berlin, Berlin 1981.

[551] D. Shanks and W. Wrench, Calculation of π to 100,000 decimals, *Math. Computing* 16 (1962), 76–79.

[552] P. S. Shary, Solving the linear interval tolerance problem, *Mathematics and Computers in Simulation* 39 (1995), 53–85.

[553] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, *Disccrete Comput. Geom.* 18 (1997), 305–363.

[554] D. V. Shiriaev, On the memory efficient differentiation, *ZAMM* 72 (1992), T632–T634.

[555] D. V. Shiriaev, Reduction of spatial complexity in reverse automatic differentiation by means of inverted code, in: [52], pp. 475–484, 1992.

[556] D. V. Shiriaev, *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment*, Dissertation, Universität Kalrsuhe, 1993.

[557] D. V. Shiriaev and G. W. Walster, *Interval Arithmetic Specification*, 1998, available at `http://www.mscs.mu.edu/~globsol/readings.html`.

[558] R. D. Skeel, Iterative refinement implies numerical stability for Gaussian elimination, *Math. Comp.* 35 (1980), 817–832.

[559] D. M. Smith, Algorithm 693: A Fortran package for floating-point multiple-precision arithmetic, *ACM Trans. on Math. Software* 17:2 (1991), 273–283.

[560] J. Spanier and K. B. Oldham, *An Atlas of Functions*, Hemisphere publishing corporation, 1987.

[561] O. Spaniol, *Arithmetik in Rechenanlagen – Logik und Entwurf*, B. G. Teubner, Stuttgart, 1976.

[562] A. P. Speiser, 50 years of the seminar for applied mathematics, Swiss Federal Institute of Technology, November 18–20, 1998, The Early Years of the Institute: Acquisition and Operation of the Z4, Planning of the ERMETH, *Mitteilungen der Gesellschaft für Angewandte Mathematik und Mechanik* 22:2 (1999), 159–168.

[563] H. J. Stetter, Sequential defect correction for high-accuracy floating-point algorithms, in: *Numerical Analysis*, pp. 186–202, Lecture Notes in Mathematics 1066, Springer, Berlin Heidelberg New York, 1984.

[564] H. J. Stetter, Staggered correction representation, a feasible approach to dynamic precision, in: *Proceedings of the Symposium on Scientific Software*, edited by D. Y. Cai, L. D. Fosdick and H. C. Huang, China University of Science and Technology Press, Beijing, China, 1989.

[565] G. H. Stewart, *Introduction to Matrix Computations*, Academic Press, New York, 1973.

[566] J. E. Stine, *Design Issues for Accurate and Reliable Arithmetic*, Ph.D. Thesis, Lehigh University, January 2001.

[567] J. E. Stine and M. J. Schulte, A combined interval and floating point multiplier, in: *Proceedings of the 8th Great Lakes Symposium on VLSI, Lafayette, LA, February, 1998*, pp. 208–213, IEEE Computer Society Press, Wahington, D.C., 1998.

[568] J. E. Stine and M. J. Schulte, A case for interval hardware on superscalar processors, in: [334], pp. 53–68, 2001.

[569] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer, New York, 1980.

[570] U. Storck, Zweidimensionale Integration mit automatischer Ergebnisverifikation, *ZAMM* 73 (7/8) (1993), T897–T899.

[571] U. Storck, *Verified Calculation of the Nodes and Weights for Gaussian Quadrature Formulas*, Interval Computation, St. Petersburg, 1993.

[572] U. Storck, Numerical integration in two dimensions with automatic result verification, in: [5], pp. 187–224, 1993.

[573] U. Storck, *Verifizierte Berechnung mehrfach geschachtelter singulärer Integrale der Gaskinetik*, Dissertation, Universität Karlsruhe, 1995.

[574] U. Storck, R. Lohner and U. Schnabel, *An Algorithm for Inclusion of Multiple and Clusters of Eigenvalues*, Universität Karlsruhe, Fakultät für Mathematik, preprint Nr. 01/22, 2001, submitted to LAA for publication.

[575] A. Schönhage and V. Strassen, Schnelle Multiplikation großer Zahlen, *Computing* 7 (1971), 281–292.

[576] T. Sunaga, Theory of an interval algebra and its application to numerical analysis, *RAAG Memoires* 2 (1958), 547–564.

[577] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko and T. Sumi, Leading-zero anticipatory logic for high-speed floating-point addition, *IEEE Journal of Solid-State Circuits* 31:8 (1996), 1157–1169.

[578] D. W. Sweeny, On the computation of Euler's constant, *Math. Comp.* 17 (1963), 170–178.

[579] N. Takagi, *Studies on hardware algorithms for arithmetic operations with redundant binary representation*, Dissertation, Department of Information Science, Faculty of Engineering, Kyoto University, 1987.

[580] N. Takagi, H. Yasuura and S. Yajima, High speed VLSI multiplication algorithm with a redundant binary adder tree, *IEEE Trans. on Computers* 34 (1985), 789–796.

[581] P. T. P. Tang, Table-driven implementation of the exponential function in IEEE floating-point arithmetic, *ACM Trans. on Math. Software* 15:2 (1989), 144–157.

[582] P. T. P. Tang, Table-driven implementation of the logarithm function in IEEE floating-point arithmetic, *ACM Trans. on Math. Software* 16:4 (1990), 378–400.

[583] P. T. P. Tang, Table-driven implementation of the expm1 function in IEEE floating-point arithmetic, *ACM Trans. on Math. Software* 18:2 (1992), 211–222.

[584] R. J. W. T. Tangelder, *The Design of Chip Architectures for Accurate Inner Product Computation*, Dissertation, Technical University Eindhoven, 1992.

[585] T. Teufel, *Ein optimaler Gleitkommaprozessor*, Dissertation, Universität Karlsruhe, 1984.

[586] T. Teufel, Implementation of a floating-point arithmetic with an accurate scalar product for digital signal processing, in: [52], pp. 147–156, 1992.

[587] T. Teufel, Genauer und trotzdem schneller – Ein neuer Coprozessor für hochgeneue Matrix- und Vectoroperationen, *Elektronik* 26 (1994), 52–56.

[588] T. Teufel, *A Novel VLSI Vector Arithmetic Coprocessor for Advanced DSP Applications*, Proceedings of ICSPAT'96, Vol. 2, pp. 1894–1898, Boston, 1996.

[589] T. Teufel and G. Bohlender, A bit-slice processor unit for optimal arithmetic, in: [603], Vol. 1, pp. 151–154, and [532], pp. 325–329, 1986.

[590] Ch. Ullrich, *Rundungsinvariante Strukturen mit äußeren Verknüpfungen*, Dissertation, Universität Karlsruhe, 1972.

[591] Ch. Ullrich, *Computer Arithmetic and Self-Validating Numerical Methods*, Proceedings of SCAN-89, invited papers, Academic Press, San Diego, 1990.

[592] Ch. Ullrich, Programming languages for enclosure methods, in: [591], pp. 115–136, 1990.

[593] Ch. Ullrich (ed.), *Contributions to Computer Arithmetic and Self-Validating Numerical Methods*, J. C. Baltzer AG, Scientific Publishing Co., Basel, 1990.

[594] Ch. Ullrich, Interval arithmetic on computers, in: [224], pp. 473–497, 1994.

[595] Ch. Ullrich and J. Wolff von Gudenberg (eds.), *Accurate Numerical Algorithms. A Collection of DIAMOND Research Papers*, Springer, Berlin, 1989.

[596] D.K. Unkauf, A.T. Gerlicher, S.M. Rump and J.H. Bleher, *Verfahren und Schaltungsanordnung zur Addition von Gleitkommazahlen*, Patentanmeldung, Technical Report, IBM, 1986.

[597] R.S. Varga, *Matrix Iterative Analysis*, Prentice Hall, Englewood Cliffs, New Jersey, 1962.

[598] R. S. Varga, *Matrix Iterative Analysis*, second edition, Springer, Berlin, 2000.

[599] R.S. Varga, *Scientific Computation on Mathematical Problems and Conjectures*, SIAM, Philadelphia, 1990.

[600] R. Vichnevetsky and J. J. H. Miller (eds.), *IMACS'91, 13th World Congress on Computation and Applied Mathematics. Proceedings in 4 Volumes*, July 22–26, 1991, Trinity College, Dublin, Ireland, 1991 (see also [125]).

[601] J. Vignes, Discrete stochastic arithmetic for validating results of numerical software, in: [124], pp. 377–390, 2004.

[602] J. E. Volder, The CORDIC trigonometric computing technique, *IRE Transactions on Electronic Computing* EC-8:3 (1959), 330–334.

[603] B. Wahlström, R. Henriksen and N. P. Sundby (eds.), *Proceedings of 11th IMACS World Congress on System Simulation and Scientific Computation*, Vol. 5, August 5–9, 1985, Oslo, published in [532], additional papers in [270].

[604] C. S. Wallace, A suggestion for a fast multiplier, *IEEE Trans. on Computers* EC-13 (1964), 14–17.

[605] P. J. L. Wallis (ed.), *Improving Floating-Point Programming*, J. Wiley, Chichester, 1990 (ISBN 0 471 92437 7).

[606] G. W. Walster, The use and implementation of interval data types, in: [162], pp. 173–194, 2005.

[607] G. W. Walster, FORTRAN-SC. A FORTRAN extension for engineering/scientific computation with access to ACRITH: Language description with examples, in: [426], pp. 43–62 1988.

[608] G. W. Walster, Einführung in die wissenschaftlich-technische Programmiersprache FORTRAN-SC, *ZAMM* 69:4 (1989), T52–T54.

[609] G. W. Walster, *Flexible Precision Control and Dynamic Data Structures for Programming Mathematical and Numerical Algorithms*, Dissertation, Universität Karlsruhe, 1990.

[610] G. W. Walster, FORTRAN-XSC: A portable FORTRAN 90 module library for accurate and reliable scientific computing, in [10], pp. 265–285, 1993.

[611] W. Walter, *FORTRAN-SC: A FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH*, Language Reference and User's Guide, second edition, IBM Deutschland GmbH, Stuttgart, January 1989.

[612] W. V. Walter, *Mathematical Foundations of Fully Reliable and Portable Software for Scientific Computing*, Universität Karlsruhe, 1995.

[613] J. S. Walther, *A Unified Algorithm for Elementary Functions*, Spring Joint Computer Conference Proc., Vol. 38, 1971.

[614] J. Weissinger, *Numerische Mathematik auf Personal Computern*, Bibliographisches Institut, Mannheim, 1984.

[615] J. Weissinger, *Spärlich besetzte Gleichungssysteme. Eine Einführung mit Basic- und Pascal-Programmen*, Bibliographisches Institut, Mannheim, 1990.

[616] A. Wiethoff, *Globale Optimierung auf Parallelrechnern*, Dissertation, Universität Karlsruhe, 1998.

[617] J. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

[618] J. Wilkinson, *Rundungsfehler*, Springer, Berlin, 1969.

[619] J. Wilkinson and C. Reinsch, *Handbook for Automatic Computation*, Vol. 2: *Linear Algebra*, Springer, Berlin Heidelberg New York, 1971.

[620] D. T. Winter, Automatic identification of scalar products, in: [605], 1990.

[621] Th. Winter, *Ein VLSI-Chip für Gleitkomma-Skalarprodukt mit maximaler Genauigkeit*, Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, 1985.

[622] H.-W. Wippermann, Realisierung einer Intervallarithmetik in einem ALGOL-60 System, *Elektronische Rechenanlagen* 9 (1967), 224–233.

[623] H.-W. Wippermann, Implementierung eines ALGOL-60 Systems mit Schrankenzahlen, *Elektronische Datenverarbeitung* 10 (1968), 189–194.

[624] J. Wolff von Gudenberg, *Einbettung allgemeiner Rechnerarithmetik in PASCAL mittels eines Operatorkonzepts und Implementierung der Standardfunktionen mit optimaler Genauigkeit*, Dissertation, Universität Karlsruhe, 1980.

[625] J. Wolff von Gudenberg, Computing z^y with maximum accuracy, *Computing* 31 (1983), 185–189.

[626] J. Wolff von Gudenberg, Reliable expression evaluation in PASCAL-SC, in: *Reliability in Computing*, edited by R. E. Moore, pp. 81–98, Academic Press, Boston New York London Sydney, 1988.

[627] J. Wolff von Gudenberg, Modelling SIMD-type parallel arithmetic operations in Ada, in: *Ada. The Choice for '92*, edited by D. Christodoulakis, LNCS 499, Springer, Berlin 1991.

[628] J. Wolff von Gudenberg, Parallel accurate linear algebra subroutines, *Computing* 1:2 (1995), 189–199.

[629] J. Wolff von Gudenberg, Hardware support for interval arithmetic, in: [334], pp. 32–38, 2001.

[630] J. Wolff von Gudenberg, *Hardware Support for Interval Arithmetic, Extended Version*, Report No. 125, Institut für Informatik, Universität Würzburg, 1995, and in [26], pp. 32–27, 1996.

[631] J. Wolff von Gudenberg, *Proceedings of Interval'96*, International Conference on Interval Methods and Computer Aided Proofs in Science and Engineering, Würzburg, Germany, September 30–October 2, 1996. Special issue 3/97 of the journal Reliable Computing, 1997.

[632] J. Wolff von Gudenberg, Java for scientific computation. Pros and cons, *J. Universal Computer Science* 4:1 (1998), 11–15.

[633] J. Wolff von Gudenberg, OOP and interval arithmetic – language support and libraries, in: *Numerical Software with Result Verification*, edited by R. Alt, A. Frommer, B. Kearfott and W. Luther, Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, 2000.

[634] J. Wolff von Gudenberg, Interval arithmetic on multimedia architectures, *Reliable Computing* 8:4 (2002), 307–312.

[635] T. Yilmaz, J. F. M. Theeuwen, R. J. W. T. Tangelder and J. A. G. Jess, The design of a chip for scientific computation, Eindhoven University of Technology, 1989, and in: *Proceedings of the Euro-Asic Symposium, Grenoble, Jan. 25–27, 1989*, pp. 335–346.

[636] J. M. Yohe, Roundings in floating-point arithmetic, *IEEE Trans. on Computers* C-22:6 (1973), 577–586.

[637] G. Zielke and V. Drygalla, Genaue Lösung linearer Gleichungssysteme, GAMM Mitt., *Ges. Angew. Math. Mech.* 26 (2003), 7–107.

[638] A. Ziv, Fast evaluation of elementary mathematical functions with correctly rounded last bit, *ACM Trans. on Math. Software* 17:3 (1991), 410–423.

[639] Y.-K. Zhu, J.-H. Yong and G.-Q. Zheng, A new distillation algorithm for floating-point summation, *SIAM J. Sci. Comput.* 26:6 (2005), 2066–2078.

[640] K. Zuse, *Der Plankalkül*, GMD Berichte, Bonn, 1972.

[641] K. Zuse, *Der Computer – Mein Lebenswerk*, Springer, Berlin, 1984.

[642] K. Zuse, *The Plankalkül*, Oldenbourg, München Wien, 1989.

[643] American National Standards Institute/Institute of Electrical and Electronics Engineers, *IEEE Standard Pascal Computer Programming Language*, ANSI/IEEE Std. 770 X3.97-1983, New York, 1983; J. Wiley & Sons Inc., 1983.

[644] American National Standards Institute/Institute of Electrical and Electronics Engineers, *A Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985, New York, 1985 (reprinted in *SIGPLAN* 22:2 (1987), 9–25).

[645] American National Standards Institute/Institute of Electrical and Electronics Engineers, *A Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std. 854-1987, New York, 1987.

[646] Cyrix, *FasMath CX-83D87 User's Manual*, Cyrix Corporation, P.O. Box 850118, Richardson, TX 75085-0118, 1990.

[647] IAM, *PASCAL-XR: PASCAL for eXtended Real arithmetic*, Joint research project with Nixdorf Computer AG, Institute of Applied Mathematics, Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, 1980.

[648] IAM, *FORTRAN-SC: A FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH*, Institute of Applied Mathematics, Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, January 1989.
1. Language Reference and User's Guide, second edition.
2. General Information Notes and Sample Programs.

[649] IAM, *ACRITH–XSC, A Programming Language for Scientific Computation*, Syntax Diagrams, Institute of Applied Mathematics, Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, 1990.

[650] IBM, *IBM System/370 RPQ. High Accuracy Arithmetic*, SA 22-7093-0, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1984.

[651] IBM, *IBM High-Accuracy Arithmetic Subroutine Library (ACRITH)*, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), third edition, 1986.
1. General Information Manual, GC 33-6163-02.
2. Program Description and User's Guide, SC 33-6164-02.
3. Reference Summary, GX 33-9009-02.

[652] IBM, *Verfahren und Schaltungsanordnung zur Addition von Gleitkommazahlen*, Europäische Patentanmeldung, EP 0 265 555 A1, 1986.

[653] IBM, *ACRITH–XSC: IBM High Accuracy Arithmetic – Extended Scientific Computation. Version 1, Release 1*, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1990.
1. General Information, GC33-6461-01.
2. Reference, SC33-6462-00.
3. Sample Programs, SC33-6463-00.
4. How To Use, SC33-6464-00.
5. Syntax Diagrams, SC33-6466-00.

[654] Institute of Electrical and Electronics Engineers (IEEE), *Proceedings of* x*-th Symposium on Computer Arithmetic ARITH*, IEEE Computer Society Press, IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

Editors of proceedings, place of conference, date of conference.

1. R. R. Shively, Minneapolis, June 16, 1969.
2. H. L. Garner and D. E. Atkins, Univ Maryland, College Park, May 15–16, 1972.
3. T. R. N. Rao and D. W. Matula, SMU, Dallas, Nov. 19–20, 1975.
4. A. Avizienis and M. D. Ercegovac, UCLA, Los Angeles, October 25–27, 1978.
5. K. S. Trivedi and D. E. Atkins, Univ Michigan, Ann Arbor, May 18–19, 1981.
6. T. R. N. Rao and P. Kornerup, Univ Aarhus, Denmark, June 20–22, 1983.
7. K. Hwang, Univ Illinois, Urbana, June 4–6, 1985.
8. M. J. Irwin and R. Stefanelli, Como, Italy, May 19–21, 1987.
9. M. Ercegovac and E. E. Swartzlander Jr., Santa Monica, September 6–8, 1989.
10. P. Kornerup and D. Matula, Grenoble, France, June 26–28, 1991.
11. E. E. Swartzlander Jr., M. J. Irwin and G. Jullien, Windsor, Ontario, June 29–July 2, 1993.
12. S. Knowles and W. H. Mc Allister, Bath, England, July 19–21, 1995.
13. Th. Lang, J.-M. Muller and N. Takagi, Asilomar, California, July 6–9, 1997.

[655] IEEE, *A Proposed Standard for Binary Floating-Point Arithmetic*, IEEE Computer, March 1981.

[656] IMACS and GAMM, IMACS-GAMM resolution on computer arithmetic, *Mathematics and Computers in Simulation* 31 (1989), 297–298, or in *Zeitschrift für Angewandte Mathematik und Mechanik* 70:4 (1990), T5, or in [591], pp. 301–302, 1990, or in [592], pp. 523–524, 1990, or in [271], pp. 477–478, 1991.

[657] IMACS and GAMM, IMACS-GAMM proposal for accurate floating-point vector arithmetic, *GAMM Rundbrief* 2 (1993), 9–16. Mathematics and Computers in Simulation, Vol. 35, IMACS, North Holland, 1993. News of IMACS, Vol. 35, No. 4, 375–382, Oct. 1993.

[658] ISO/IEC, *ISO/IEC 7185:1990 – Information Technology – Programming Languages – Pascal*, second edition, 1990.

[659] ISO/IEC, *ISO/IEC 10206:1991 – Information Technology – Programming Languages – Extended Pascal*, 1991.

[660] ISO, *Language Independent Arithmetic Standard (LIA-1)*, Second Committee Draft Standard (Version 4.0), ISO/IEC CD 10967-1, 1992.

[661] *MAPLE, Reference Manual*, Symbolic Computation Group, University of Waterloo, Ontario, Canada, 1988.

[662] Numerik Software GmbH, *PASCAL-XSC: A PASCAL Extension for Scientific Computation. User's Guide*, Numerik Software GmbH, Haid-und-Neu-Straße 7, D-76131 Karlsruhe, Germany / Postfach 2232, D-76492 Baden-Baden, Germany, 1991.

[663] SIEMENS, *ARITHMOS (BS 2000) Unterprogrammbibliothek für Hochpräzisionsarithmetik. Kurzbeschreibung, Tabellenheft, Benutzerhandbuch*, SIEMENS AG, Bereich Datentechnik, Postfach 83 09 51, D-8000 München 83, Bestellnummer U2900-J-Z87-1, September 1986.

[664] Sun Microsystems, *Interval Arithmetic Programming Reference, Fortran 95*, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA, 2000.

List of Figures

List of Tables

Index