Studies in Computational Intelligence 805

Bartłomiej Jacek Kubica

Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems

From Inequalities Systems to Game Solutions

Springer

Studies in Computational Intelligence

Volume 805

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

The series "Studies in Computational Intelligence" (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

The books of this series are submitted to indexing to Web of Science, EI-Compendex, DBLP, SCOPUS, Google Scholar and Springerlink.

More information about this series at http://www.springer.com/series/7092

Bartłomiej Jacek Kubica

Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems

From Inequalities Systems to Game Solutions

Springer

Bartłomiej Jacek Kubica
Department of Applied Informatics
Warsaw University of Life Sciences
Warsaw, Poland

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-030-13794-6 ISBN 978-3-030-13795-3 (eBook)
https://doi.org/10.1007/978-3-030-13795-3

Library of Congress Control Number: 2019931826

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my beloved son, Stefanek, the joy of my life.

Preface

This book contains recent research on interval methods for solving nonlinear constraint satisfaction, optimization and similar problems. It presents a comprehensive survey of applications: it includes various branches of robotics, artificial intelligence systems, economy, control theory, dynamical systems theory and others. The book is completed with three Appendices, describing the notation, representation of numbers, used as intervals' endpoints and example implementations of the interval data type in a few programming languages.

Warsaw, Poland — Bartłomiej Jacek Kubica

Acknowledgements

The author is grateful to everyone, who helped him in the process of preparation of this monograph.

Many thanks to Prof. Roman Wyrzykowski for all his help and for providing the `MICLab` project, crucial in performing several of the practical experiments.

I would like to acknowledge my friend and 'closest approximation to mentor I have ever had', Adam Woźniak.

Thanks to Prof. Sergey P. Shary, my great friend, and his wife, Irene A. Sharaya, who, unfortunately, is no longer between us.

Particular thanks to my Mother, my Wife and all my family.

And—last, but not least—thanks to my Lord, Jesus Christ, because without Him we can do nothing (John 15, 5).

Contents

Acronyms

AD	Algorithmic (Automatic) Differentiation
AHP	Analytic Hierarchy Process
AI	Artificial Intelligence
B&B	Branch-and-Bound method
B&BT	Branch-and-Bound-type method
B&P	Branch-and-Prune method
BC	Box-Consistency
CLR	Common Language Runtime (the .NET virtual machine)
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DBN	Deep Belief Network
DP	Dynamic Programming
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
GS	Gauss–Seidel
HC	Hull-Consistency
HIBA_USNE	Heuristical Interval Branch-and-prune Algorithm for Underdetermined and well-determined Systems of Nonlinear Equations
IRR	Internal Rate of Return
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MCDM	Multi-Criteria Decision-Making
MPC	Model Predictive Control
MVA	Mean-Value Analysis
NaN	Not-a-Number or Not-any-Number
NE	Nash Equilibrium/Equilibria
NPV	Net Present Value
ODE	Ordinary Differential Equations
PDE	Partial Differential Equations

PDF	Probability Density Function
PID	Proportional–Integral–Derivative controller
PPS	Partitioning Parameter Space
RPC	Remote Procedure Call
SIVIA	Set Inversion Via Interval Analysis
SLAM	Simultaneous Localization and Mapping
SNE	Strong Nash Equilibrium/Equilibria
SVM	Support Vector Machine
TBB	Threading Building Blocks
TOPSIS	Technique for Order of Preference by Similarity to Ideal Solution
VaR	Value at Risk
XML	eXtensible Markup Language

List of Figures

Chapter 1
Introduction

The generic problem we are considering in this monograph can be formulated as follows:

$$\text{Find } \textit{all } x \in X \text{ such that } P(x) \text{ is fulfilled.} \tag{1.1}$$

Here, $P(x)$ is a formula with a free variable x and $X \subseteq \mathbb{R}^n$.

The above problem is ubiquitous and pretty general; we can think of it as some generalization of a constraint satisfaction problem (CSP). Its instances include:

- systems of equations—then $P(x)$ means "$\big(f(x) = 0\big)$" for some function $f : \mathbb{R}^n \to \mathbb{R}^m$;
- constraint satisfaction problems—then $P(x)$ means "$\big(g(x) \leq 0\big)$" or"$\big((g(x) \leq 0)$ and $(f(x) = 0)\big)$", for $g : \mathbb{R}^n \to \mathbb{R}^{m_1}$, $f : \mathbb{R}^n \to \mathbb{R}^{m_2}$;
- global optimization problems—then $P(x)$ means "$(\forall t \in X)\ \big(f(x) \leq f(t)\big)$", for $f : \mathbb{R}^n \to \mathbb{R}$;
- problems of seeking Pareto-optimal points of a multicriteria problem—then $P(x)$ means "$(\forall t \in X)\ \big((\forall i = 1, \ldots, N)\ (f_i(x) \leq f_i(t))$ or $(\exists j \in [1..N])\ (f_j(x) < f_j(t))\big)$", for $f : \mathbb{R}^n \to \mathbb{R}^N$;
- and many other problems discussed, i.a., in Chap. 4 (cf. also [6, 7]).

As we can easily tell, the problem under consideration might be a decision problem, an optimization problem, an approximation problem or virtually any other type of problems encountered in several branches of science and engineering.

Rarely have such problems been considered, in the past. Typically, the problem under consideration has been formulated as an optimization problem or a CSP, or an instance of another well-known type of problems. Yet, in this monograph, an approach to solve the generic problem is proposed.

What can the formula P be like, in general? It is a formula, in the first-order logic [1]. But in what formal system?

As a first-order formula, P consists of variables (ranging over some domain X, or over a few domains), constants (with numerical values), comparison signs ($=, \leq, \geq, <, >$), logical conjunctions (and, or, not), logical quantifiers ($\forall$ and $\exists$;

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_1

generalizations, like Mostowski's quantifiers [8], may be considered, as well), and functions (like $f(x)$, in the above examples).

To define the formal system precisely, we should clarify, what function symbols can be used in our formulae. As it has been stated in many former papers (in particular, [3, 5]), these functions have to be computable, i.e., to allow bounding their value for each argument, with a given accuracy. This is certainly possible for Lipschitz-continuous functions, i.e., functions satisfying the condition:

$$(\exists L \geq 0) \quad (\forall x \in X) \quad (\forall y \in X) \qquad ||f(x) - f(y)|| \leq L \cdot ||x - y|| \,,$$

for some norm $|| \cdot ||$.

Yet, it is worth noting that Lipschitz functions are not the only ones that can be bounded. As pointed in [4] or [10], even functions that are nowhere smooth or differentiable can sometimes be bounded (using interval methods, as we shall see).

It is possible that so-called Hausdorff-continuous functions [11] are the most general class of functions that can be used to construct the formula P, but this important conjecture has not been proven yet, and it will not be further studied in this monograph. What is relevant is *how* to solve problems of type (1.1) for several commonly encountered classes of functions: polynomials, rational functions, common transcendental functions (like exp, sin, cos, etc.), and possibly also other functions.

When restricting to polynomial functions, symbolic quantifier elimination methods (like cylindrical algebraic decomposition) may be sufficient for solving them. This property does not hold for more general classes of functions [9], and purely symbolic techniques tend to be inefficient, even if applicable; yet there are other approaches, as well [2].

But why are problems of type (1.1) so important to us? Firstly, because, as we have said, they are very common in several branches of science and engineering (cf., e.g., Chap. 9, for the survey of applications). Secondly, because we have a proper tool for solving them. This tool, the so-called *interval analysis*, is going to be presented in the next chapter.

The remainder of the book is organized as follows. After introducing interval calculus in Chap. 2, we present algorithmic differentiation techniques, in Chap. 3. These techniques help us to bound not only functions, discussed above, but also their derivatives. In particular, the author's library ADHC is briefly presented.

Then, in Chap. 4, the author proposes the main generic branch-and-bound-type algorithm for solving problems of type (1.1). Its logical structure and basic features are analyzed and discussed.

In Chap. 5, the simplest instance of the generic algorithm is presented: the branch-and-prune algorithm. This version is adequate for solving equations systems and constraint satisfaction problems. Results of the author and other researchers are reviewed and the author's HIBA_USNE solver is presented.

Chapter 6 reviews three other types of quantified problems that can be solved using the proposed interval B&BT algorithm: global optimization, seeking Pareto-sets of multicriteria problems and seeking game solutions.

Next, in Chap. 7, problems and issues of parallelization of B&BT algorithms are presented. Both shared-memory and distributed-memory environments are considered. Several tools and features are mentioned.

Libraries and other software available for interval B&BT algorithms are the subject of Chap. 8. Then, in Chap. 9, applications are discussed. The survey tends to be comprehensive: it includes various branches of robotics, artificial intelligence systems, economy, control theory, dynamical systems theory, and others.

The book is completed with three Appendices, describing the notation (to make it easy for checking, while reading the monograph), representation of numbers, used as intervals' endpoints (floating-point numbers, fixed-point numbers and unums) and example implementations of the interval data type in a few programming languages.

References

1. Adler, J., Schmid, J.: Introduction to Mathematical Logic. University of Bern (2007)
2. Franek, P., Ratschan, S., Zgliczynski, P.: Satisfiability of systems of equations of real analytic functions is quasi-decidable. In: International Symposium on Mathematical Foundations of Computer Science, pp. 315–326. Springer (2011)
3. G-Tóth, B., Kreinovich, V.: Verified methods for computing Pareto sets: General algorithmic analysis. Int. J. Appl. Math. Comput. Sci. **19**(3), 369–380 (2009)
4. Gutowski, M.W.: Introduction to interval calculi and methods (in Polish). BEL Studio, Warszawa (2004)
5. Kreinovich, V., Kubica, B.J.: From computing sets of optima, Pareto sets and sets of Nash equilibria to general decision-related set computations. J. Univers. Comput. Sci. **16**, 2657–2685 (2010)
6. Kubica, B.J.: A class of problems that can be solved using interval algorithms. Computing **94**, 271–280 (2012). (SCAN 2010 (14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics) Proceedings)
7. Kubica, B.J.: Interval methods for solving various kinds of quantified nonlinear problems. In: O. Kosheleva (ed.) Beyond Traditional Probabilistic Data Processing Techniques: Interval, Fuzzy, etc. Methods and Their Applications (2018)
8. Mostowski, A.: On a generalization of quantifiers. Fundamenta Mathematicae **44**(1), 12–36 (1957)
9. Ratschan, S.: Efficient solving of quantified inequality constraints over the real numbers. ACM Trans. Comput. Logic (TOCL) **7**(4), 723–748 (2006)
10. Rump, S.M.: Inclusion of zeros of nowhere differentiable n-dimensional functions. Reliab. Comput. **3**(1), 5–16 (1997)
11. Sendov, B.: Hausdorff Approximations, vol. 50. Springer Science & Business Media (1990)

Chapter 2
Interval Calculus

2.1 Introduction

Interval calculus is a branch of numerical analysis and mathematics that operates on sets rather than numbers; specifically, it operates on *intervals*, obviously. A domain strictly conjuncted with the interval analysis is the so-called *reliable computing* that performs verified (or "certified", as many authors say) computations, rigorously bounding the numerical and any other errors.

Why do we need such computations? The exhaustive answer to this question is more complicated than it seems. Let us postpone it until Sect. 2.9. Now, let us just briefly say, it finds applications in description of several kinds of errors and uncertainty, including numerical inaccuracy and measurement errors. But, as we shall see later, this is only a small part of fields, where intervals can be (and often are) used.

First works and concepts on interval calculus have been developed (independently) by Norbert Wiener, Warmus [36], Moore [26] and other researchers, in the first decades of the XX-th century. The first comprehensive study is due to Ramon E. Moore, in his Ph.D. dissertation, defended in 1962 at Stanford University, while his first published book [25] occurred in 1966. Also, Moore is usually considered to be the "founding father" of the interval calculus.

Since than, this approach slowly becomes more and more widely accepted in many branches of science and engineering.

Libraries and other software working with intervals will be surveyed in Chap. 8. Here, let us present the theory.

2.2 Basics of Interval Computations

The idea of interval analysis can be found in several textbooks; e.g., [10, 15, 17, 22, 26, 27, 35].

We define the (closed) interval $[\underline{x}, \overline{x}]$ as a set $\{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$. We denote all intervals by brackets; open ones will be denoted as $]\underline{x}, \overline{x}[$ and partially open as:

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_2

$[\underline{x}, \overline{x}[$, $]\underline{x}, \overline{x}]$. (We prefer this notation to using the parenthesis, that are used also to denote sequences, vectors, etc.)

In computer programs, we (typically) represent an interval usually as a pair of numbers: its endpoints. Should we add some flags to indicate, if the interval is closed, open or half-open? We discuss this in more details below, in Sect. 2.8; here let us just say that usually, we restrict ourselves to representing closed intervals.

Following [19], we use boldface lowercase letters to denote interval variables, e.g., $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbb{IR}$ denotes the set of all real intervals.

2.3 Operations on Intervals

We design arithmetic (and other) operations on intervals so that the following condition was fulfilled:

$$\odot \in \{+, -, \cdot, /\},\ a \in \mathbf{a},\ b \in \mathbf{b} \text{ implies } a \odot b \in \mathbf{a} \odot \mathbf{b}\,. \tag{2.1}$$

2.3.1 Interval Arithmetic

The actual formulae for arithmetic operations—see, e.g., [10, 15, 17]—are as follows:

$$\begin{aligned}
[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}]\,,\\
[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}]\,,\\
[\underline{a}, \overline{a}] \cdot [\underline{b}, \overline{b}] &= [\min(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}), \max(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab})]\,,\\
[\underline{a}, \overline{a}] \,/\, [\underline{b}, \overline{b}] &= [\underline{a}, \overline{a}] \cdot \left[1 \,/\, \overline{b}, 1 \,/\, \underline{b}\right], \qquad 0 \notin [\underline{b}, \overline{b}]\,.
\end{aligned} \tag{2.2}$$

It is worth noting, that division by an interval containing zero is also possible—in the extended Kahan-Novoa-Ratz arithmetic [17]:

$$\mathbf{a}/\mathbf{b} = \begin{cases}
\mathbf{a} \cdot [1/\overline{b}, 1/\underline{b}] & \text{for } 0 \notin \mathbf{b}\\
[-\infty, +\infty] & \text{for } 0 \in \mathbf{a} \text{ and } 0 \in \mathbf{b}\\
[\overline{a}/\underline{b}, +\infty] & \text{for } \overline{a} < 0 \text{ and } \underline{b} < \overline{b} = 0\\
[-\infty, \overline{a}/\overline{b}] \cup [\overline{a}/\underline{b}, +\infty] & \text{for } \overline{a} < 0 \text{ and } \underline{b} < 0 < \overline{b}\\
[-\infty, \overline{a}/\overline{b}] & \text{for } \overline{a} < 0 \text{ and } 0 = \underline{b} < \overline{b}\\
[-\infty, \underline{a}/\underline{b}] & \text{for } 0 < \underline{a} \text{ and } \underline{b} < \overline{b} = 0\\
[-\infty, \underline{a}/\underline{b}] \cup [\underline{a}/\overline{b}, +\infty] & \text{for } 0 < \underline{a} \text{ and } \underline{b} < 0 < \overline{b}\\
[\underline{a}/\overline{b}, +\infty] & \text{for } \underline{a} < 0 \text{ and } 0 = \underline{b} < \overline{b}\\
\emptyset & \text{for } 0 \notin \mathbf{a} \text{ and } 0 = \mathbf{b}
\end{cases}. \tag{2.3}$$

Remark 2.1 Formulae (2.2), although well-known, are less universal than they would seem. If the endpoints are represented using the IEEE 754 Standard for floating-point numbers [13], these endpoints can have infinite values. So what would be the result of the following multiplication: $[0, 1] \cdot [1, +\infty]$? According to Formulae (2.2), we obtain a NaN (Not any Number) for the right endpoint, but we can simply bound the set:

$$\{z = x \cdot y \mid x \in [0, 1],\ \ y \in [1, +\infty]\}\ ;$$

its bounds are obviously: $[0, +\infty]$.

Various interval libraries and packages implement such operations in different manners; some unification has been provided by the IEEE Standard 1788–2015 [14]. We get back to this topic in Chap. 8.

The definition of interval vector $\mathbf{x}$, a subset of $\mathbb{R}^n$ is straightforward: $\mathbb{R}^n \supset \mathbf{x} = \mathbf{x}_1 \times \cdots \times \mathbf{x}_n$. Traditionally, an interval vector is called a *box*.

2.3.2 Interval Enclosures of Other Operations and Functions

We can obtain similar formulae for power of an interval:

$$[\underline{a}, \overline{a}]^n = \begin{cases} [\underline{a}^n, \overline{a}^n] & \text{for odd } n \\ [\min\{\underline{a}^n, \overline{a}^n\}, \max\{\underline{a}^n, \overline{a}^n\}] & \text{for even } n \text{ and } 0 \notin [\underline{a}, \overline{a}] \\ [0, \max\{\underline{a}^n, \overline{a}^n\}] & \text{for even } n \text{ and } 0 \in [\underline{a}, \overline{a}] \end{cases} . \tag{2.4}$$

and other transcendental functions, e.g.:

$$\begin{aligned} \exp\left([\underline{a}, \overline{a}\right) &= [\exp(\underline{a}), \exp(\overline{a})], \\ \log\left([\underline{a}, \overline{a}\right) &= [\log(\underline{a}), \log(\overline{a})], \text{ for } \underline{a} > 0, \\ &\ldots \end{aligned}$$

Links between real and interval functions are set by the notion of an *inclusion function*: see; e.g., [15]; also called an *interval extension*; e.g., [17]. But before we describe this notion, let us devote a few words to properties of interval operations.

2.3.3 Auxiliary Operations

For each interval, we can define its width (sometimes called its *diameter*): $\operatorname{wid} \mathbf{x} = \overline{x} - \underline{x}$ and midpoint: $\operatorname{mid} \mathbf{x} = \frac{\underline{x}+\overline{x}}{2}$.

The interval hull $\square S$ of the set $S \subseteq \mathbb{R}^n$ is the smallest box $\mathbf{x} \in \mathbb{IR}^n$ such that $\mathbf{x} \supseteq S$. This notion should not be confused with a *convex hull*, that is in general not a box.

This terminology and notation follows [19].

2.4 Properties and Features of the Interval Calculus

The arithmetic defined by formule (2.2) has properties very different than the arithmetic of real numbers.

Let us consider the simplest example: what is the value of $\mathbf{x} - \mathbf{x}$? According to (2.2), we obtain $[\underline{x} - \overline{x}, \overline{x} - \underline{x}]$, which is not necessarily zero; it would be zero only for the degenerate case of $\underline{x} = \overline{x}$. This means, intervals of the set $\mathbb{IR}$ do not form a group, not to mention a ring, field or linear space.

Remark 2.2 It is worth noting that there are some extensions of the traditional interval arithmetic, making some superset $\mathbb{KR}$ of $\mathbb{IR}$ a group. This is obtained by introducing so-called improper intervals, i.e., "intervals" $[\underline{x}, \overline{x}]$ such that $\underline{x} > \overline{x}$. Discussion of Kaucher arithmetic (see [16]; cf. [35]) or modal interval arithmetic [3, 4], while interesting and important, is beyond the scope of this monograph.

Also, the law of distributivity is in general not fulfilled for intervals. Instead, we have the so-called *subdistributivity* principle:

$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) \subseteq \mathbf{ab} + \mathbf{ac} \,.$$

Inclusion in the above formula becomes equality only under some specific circumstances, e.g., when $\mathbf{a}$ is a degenerate interval $[a, a]$. Other cases are discussed in a very interesting paper [34].

As these properties are very different from the ones known from the arithmetic of real-numbers, they are not as much different from the arithmetic of *floating-point* numbers. It is well-known that the numerical accuracy of an expression depends on the formula that is used. Yet for intervals of arbitrary length (numerical errors are usually assumed to be "small"), the problem is even more important.

Details will become clear in the next section, where we shall discuss various interval extensions of a function.

2.5 Interval Extension of a Function

Definition 2.1 A function $\mathsf{f}\colon \mathbb{IR} \to \mathbb{IR}$ is an *inclusion function* of $f : \mathbb{R} \to \mathbb{R}$, if for each interval $\mathbf{x}$ within the domain of f the following condition is satisfied:

$$\{f(x) \mid x \in \mathbf{x}\} \subseteq \mathsf{f}(\mathbf{x}) \,.$$

The definition is analogous for functions $f : \mathbb{R}^n \to \mathbb{R}^m$.

2.5.1 Most Common Forms of Interval Extensions

Obviously, a so-called *natural interval extension*, consisting of the same operations as the real-valued function, is its interval extension. This property is a direct consequence of (2.1).

Example 2.1 Consider the function $f(x) = x^2 + x + 1$. Its *natural interval extension* is, obviously, $\mathsf{f}(\mathbf{x}) = \mathbf{x}^2 + \mathbf{x} + 1$.

Obviously, so are other interval extensions, derived from other expressions, that would be equivalent for real numbers, but may be more (or less) precise for intervals: $\mathsf{f}_1(\mathbf{x}) = \mathbf{x} \cdot (\mathbf{x} + 1) + 1$ and $\mathsf{f}_2(\mathbf{x}) = \left(\mathbf{x} + \frac{1}{2}\right)^2 + \frac{3}{4}$.

Another commonly used interval extension or, more precisely, class of interval extensions, are so-called *centered forms* of the inclusion function. They appear as follows:

$$\mathsf{f}_c(\mathbf{x}) = \mathsf{f}(c) + \nabla\mathsf{f}(\mathbf{x}) \cdot (\mathbf{x} - c), \text{ where } c \in \mathbf{x}\,. \tag{2.5}$$

Usually $c = \operatorname{mid}\mathbf{x}$ is taken (and the centered form is then called the *meanvalue form*). Other points might be also useful; so-called *corner forms* have even been patented.

Example 2.2 Consider the function $f(x)$ from Example 2.1. The *meanvalue form* of its interval extension is: $\mathsf{f}_c(\mathbf{x}) = \check{x}^2 + \check{x} + 1 + \nabla\mathsf{f}(\mathbf{x}) \cdot (\mathbf{x} - \check{x})$, where $\check{x} = \operatorname{mid}\mathbf{x}$ and $\nabla\mathsf{f}(\mathbf{x}) = 2 \cdot \mathbf{x} + 1$.

When computing interval operations—either the ones above or computing the enclosure for a transcendental function—we can round the lower bound downward and the upper bound upward. This will result in an interval that will be overestimated, but will be *guaranteed to contain the true result of the real-number operation*. An interesting theoretical analysis of rounded computations can be found, e.g., in [21] or, in a more modern approach, [23], where it is considered with relation to current architectures of Intel processors.

2.5.2 How to Construct Formulae for Interval-Valued Functions?

So, there can be several interval extensions of the same real-valued function f. Accuracies of these extensions may differ to the high extent; also on various subareas of the domain of f, various interval extensions can be more precise. Which of them to choose? What criteria to use?

There are no simple answers to these questions. Let us consider a univariate polynomial: $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$. Its natural interval extension:

$$\mathsf{f}(\mathbf{x}) = \mathbf{a}_0 + \mathbf{a}_1 \mathbf{x} + \mathbf{a}_2 \mathbf{x}^2 + \cdots + \mathbf{a}_n \mathbf{x}^n \,, \tag{2.6}$$

may certainly be severely overestimated.

For floating-point numbers, it is well-known that the optimal—both in terms of accuracy and efficiency—manner to compute the value of a polynomial is to use the Horner's form. But the interval Horner-form of the extension function:

$$\mathsf{f}(\mathbf{x}) = \mathbf{a}_0 + \mathbf{x} \cdot (\mathbf{a}_1 + \mathbf{x} \cdot (\ldots)) \,, \tag{2.7}$$

is not necessarily a good option. As we already know, a sequence of multiplications of the same interval is, in general, less precise than the power of an interval.

In [11], so-called *remainder forms* are proposed, instead. But they are no panacea and for more sophisticated functions (or even for multivariate polynomials), we may need something different.

It seems very worthwhile to seek some symbolic techniques to transform expressions before their evaluation in interval arithmetic. Some efforts have been performed in this area, particularly using the Gröbner bases theory for polynomial systems—see, e.g., [1, 5, 28]. Nevertheless it seems, much can (and should) be improved in this area.

2.6 Comparison of Intervals

What order (or orders) does the space of intervals have? A few partial orders can be considered.

The one defined by the inclusion relation $\mathbf{x} \subseteq \mathbf{y}$ is particularly important in several algorithms. We shall meet such algorithms, in particular, in Sects. 5.3 and 5.5.

Another order is enforced by the relation "$<$": $\mathbf{x} < \mathbf{y}$, iff $(\forall x \in \mathbf{x})\ (\forall y \in \mathbf{y})\ (x < y)$. Obviously, this simply means $\overline{x} < \underline{y}$.

Some authors (e.g., [22]) consider yet other po-relations, e.g., $\mathbf{x} \preceq \mathbf{y}$, iff $\underline{x} \leq \underline{y}$ and $\overline{x} \leq \overline{y}$. This relation seems to have lower importance and it will not be considered here.

All of the above relations have been po-relations, which means some pairs of intervals are incomparable with respect to these orders. Nevertheless, for some practical problems, we need some policy (or at least heuristic) to compare arbitrary pairs of intervals. The following example will show the need for such relation.

Example 2.3 Assume we have $x \in \mathbf{x} = [0, 4]$ and $y \in \mathbf{y} = [1, 5]$. Which of these two quantities is larger (cf. Fig. 2.1)? Obviously, we cannot determine whether $x \leq y$ or $y \leq x$.

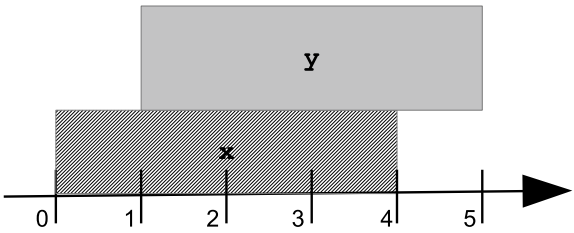

Fig. 2.1 Comparison of intervals $\mathbf{x} = [0, 4]$ and $\mathbf{y} = [1, 5]$ Example (2.3)

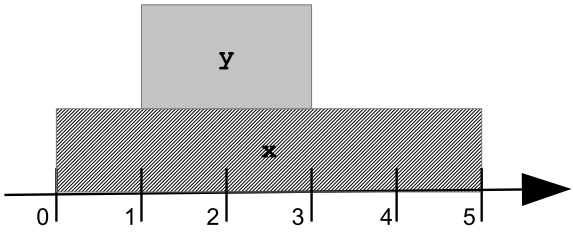

Fig. 2.2 Comparison of intervals $\mathbf{x} = [0, 5]$ and $\mathbf{y} = [1, 3]$

But if we were to guess, what would we choose? Suppose, we have two choose between two technologies to use in our projects and experts are to estimate the costs. They can provide intervals only: one has the cost $x \in \mathbf{x}$ and the other one $y \in \mathbf{y}$.

It is intuitively obvious, we should choose the technology with cost $x \in \mathbf{x} = [0, 4]$.

In Example 2.3, the result was obvious; although, it might not be obvious to verbalize the rationale behind our choice. For other pairs of intervals, say $\mathbf{x} = [0, 5]$ and $\mathbf{y} = [1, 3]$ (cf. Fig. 2.2), the decision would be less intuitive. How can we compare arbitrary intervals? There are a few heuristics for that. In a very interesting survey, Sevastjanov et al. [32] distinguish three classes of approaches to the intervals' comparison problem:

- "qualitative" heuristics, based on diagrammatic representation of intervals [24],
- "quantitative" heuristics, based on measuring distance between intervals,
- hybrid approaches.

In a series of papers, various of these approaches are elaborated and generalized to the case of fuzzy intervals [30]. Some sort of summary has been done in Sect. 3.2 of [31].

Dempster-Shafer theory

One of the approaches is based on the Dempster-Shafer theory [2]; cf. also [29, 30, 33]. It allows us to describe the degree of certainty (so-called belief function) and possibility (so-called plausibility function) of $x \leq y$ being fulfilled. This way, we obtain, the so-called *belief interval* of a condition, having a probabilistic interpretation (lower and upper probability of $x \leq y$).

For instance, let us consider intervals $\mathbf{x} = [0, 4]$ and $\mathbf{y} = [1, 5]$ from Example 2.3. Define two independent random variables: X, uniformly distributed on $\mathbf{x}$, and Y, uniformly distributed on $\mathbf{y}$. Analyzing the likelihood of $X \leq Y$, we obtain four possibilities (cf. Fig. 2.1):

(a) $H_1 = \{X \in [0, 1] \text{ and } Y \in [1, 4]\}$,
(b) $H_2 = \{X \in [0, 1] \text{ and } Y \in [4, 5]\}$,
(c) $H_3 = \{X \in [1, 4] \text{ and } Y \in [1, 4]\}$,
(d) $H_4 = \{X \in [1, 4] \text{ and } Y \in [4, 5]\}$.

Please note that $X \preceq Y$ is *certain* for cases (a), (b) and (d) and *possible* for (c). The *basic probability assignment* to these events are:

$$P(H_1) = \frac{1-0}{4-0} \cdot \frac{4-1}{5-1} = \frac{1}{4} \cdot \frac{3}{4} = \frac{3}{16} ,$$
$$P(H_2) = \frac{1-0}{4-0} \cdot \frac{5-4}{5-1} = \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16} ,$$
$$P(H_3) = \frac{4-1}{4-0} \cdot \frac{4-1}{5-1} = \frac{3}{4} \cdot \frac{3}{4} = \frac{9}{16} ,$$
$$P(H_4) = \frac{4-1}{4-0} \cdot \frac{5-4}{5-1} = \frac{3}{4} \cdot \frac{1}{4} = \frac{3}{16} .$$

And we obtain:

$$Bel\big(\{X \preceq Y\}\big) = P(H_1) + P(H_2) + P(H_4) = \frac{7}{16} ,$$
$$Pl\big(\{X \preceq Y\}\big) = P(H_1) + P(H_2) + P(H_3) + P(H_4) = 1 .$$

Please note that, in the case presented in Fig. 2.2, we have some evidence for both $x \preceq y$ and $x \succeq y$. Details can be found in [33].

Other approaches

Some of the approaches attempt to use the metric on the $\mathbb{IR}$ space (see the next section); the metric describes how far the intervals are from each other, but does not specify, which of them is "higher" and which is"lower".

Finally, it turns out, that usually the most convenient and proper manner of comparing intervals is to simply compare their midpoints, i.e.,:

$$\mathbf{x} \preceq \mathbf{y}, \text{ iff } (\underline{x} + \overline{x}) \leq (\underline{y} + \overline{y}) . \tag{2.8}$$

This approach is compatible with our intuition for Example 2.3 and results in $\mathbf{y} \preceq \mathbf{x}$ for $\mathbf{x} = [0, 5]$ and $\mathbf{y} = [1, 3]$.

Obviously, comparing midpoints of intervals is a specific case of using the Hurwicz criterion [12], i.e., comparing weighted sums (mixtures) of endpoints:

$$\alpha_H \cdot \overline{x} + (1 - \alpha_H) \cdot \underline{x} , \tag{2.9}$$

where α_H denotes the level of optimism-pessimism of the decision maker. It is easy to observe that for $\alpha_H = \frac{1}{2}$, we compare the midpoints.

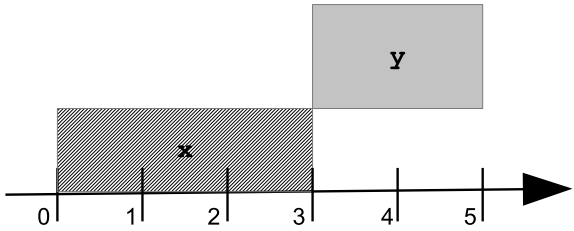

Fig. 2.3 Comparison of intervals $\mathbf{x} = [0, 3]$ and $\mathbf{y} = [3, 5]$

Anyway, all the above approaches for comparing intervals have been heuristical and more or less prone to failures. The only situation when we can *reliably* compare intervals is when $(\forall x \in \mathbf{x})\ (\forall y \in \mathbf{y})\ (x < y)$, like in Fig. 2.3.

2.7 A Metric on the Space of Intervals

We can define a metric on the space of intervals. The most commonly used one is:

$$d(\mathbf{x}, \mathbf{y}) = \max(|\underline{x} - \underline{y}|, |\overline{x} - \overline{y}|)\ . \tag{2.10}$$

Please note, however, that, in spaces of intervals, the metric has a different role, than in spaces of numbers. It should be the most convenient to explain it, using an example.

Example 2.4 Assume we have a real-valued quantity u that we want to approximate. We obtained two floating-point estimates (e.g., using two different algorithms): u_1 and u_2. The metric d in the space real numbers (or real vectors) gives us the hint about the quality of these estimates; if $|u_1 - u_2|$ is small enough, then the estimates are (probably) good.

Now, assume, we have interval enclosures of u: $\mathbf{u}_1$ and $\mathbf{u}_2$. Consider two situations:

(a) $d(\mathbf{u}_1, \mathbf{u}_2) \leq \varepsilon$, but $\mathbf{u}_1 \cap \mathbf{u}_2 = \emptyset$,
(b) we have relatively high $d(\mathbf{u}_1, \mathbf{u}_2)$, but $\mathbf{u}_1 \cap \mathbf{u}_2$ is nonempty, yet narrow.

In contrast to the non-interval case, the in the first situation, we have no good estimate (although, we know that the algorithms or their input have not been correct!), but in the second case, we got a quite good enclosure of u: $\mathbf{u}_1 \cap \mathbf{u}_2$.

2.8 Open or Closed Intervals?

We have already stated that usually, we use only closed intervals in our computations. What about open and half-open ones?

The answer to this question is not trivial. As it will be presented in Chap. 8, there are some packages that allow us to represent non-closed intervals as well. Nevertheless, most authors agree about representing closed intervals only; also the IEEE 1788–2015 Standard [14] is consistent with it.

The reasons are thoroughly discussed by Shary in Sect. 1.11B of his book [35]. Briefly summarizing this discussion, the space of closed intervals is compact, while the space of all possible intervals is not. Performing computations in a non-compact space would result in several subtle difficulties; for instance, a sequence in such a space does not necessarily achieve its limit. Also, the metric (2.10) would become a pseudo-metric, as intervals $[\underline{x}, \overline{x}]$, $]\underline{x}, \overline{x}]$, $[\underline{x}, \overline{x}[$ and $]\underline{x}, \overline{x}[$ would not be distinguishable by it.

2.9 Purposes of the Interval Calculus

Now, let us answer the question, why and for what to use interval methods? The answer to this question is more complicated than it might seem at the first glance.

To the best knowledge of the author, initially, the approach has been developed as a tool to deal with numerical inaccuracy; at least these were the considerations of Warmus and Moore. However, in this context, their usefulness is questionable. For floating-point computations, the traditional Wilkinson's error analysis [37] is sufficient, usually. Obviously, there are exceptions to this rule (see e.g., the Rump's example in [10]), but they are relatively rare.

Nevertheless, there are several other kinds of imprecision; in addition to numerical errors, we have:

- discretization errors,
- truncation errors (e.g., for infinite series),
- inexact data; e.g., results of measurements,
- human-related uncertainty: e.g., precise description of decision-makers' preferences,
- uncertainty related to decisions taken by another decision-makers,
- …

Intervals give us the tool to bound all these kinds of uncertainty. Indeed, robust control, processing measurements and game theory are areas, where interval methods have found some important applications; cf., e.g., [6–9, 15, 18, 20].

Yet, in the opinion of the author, interval calculus should *not* be understood as a tool of uncertainty description, but rather as an approach to seek points satisfying a certain logical condition. Uncertainty description is a subclass of this category of problems—we seek some parameter values such that either *for each* uncertain value a condition is fulfilled or there *exists* such value in the range of an uncertain quantity that a condition is fulfilled. But there are also important applications not related to uncertainty description. This approach is described in Chap. 4 of this volume.

Problems

2.1 Given $f(x) = x^3 + x^2$ compute $\mathsf{f}(\mathbf{x})$ for $\mathbf{x} = [-2, -1], [1, 2], [-2, 2]$. Which of these values are sharp?

2.2 Check if the distributivity law $\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{ab} + \mathbf{ac}$ is fulfilled for:

(a) $\mathbf{a} = [2, 2]$, $\mathbf{b} = [3, 5]$, $\mathbf{c} = [-1, 1]$,
(b) $\mathbf{a} = [-2, 2]$, $\mathbf{b} = [3, 5]$, $\mathbf{c} = [-1, 1]$,
(c) $\mathbf{a} = [-2, 2]$, $\mathbf{b} = [3, 5]$, $\mathbf{c} = [1, 4]$,
(d) $\mathbf{a} = [-2, 2]$, $\mathbf{b} = [-3, 3]$, $\mathbf{c} = [-1, 1]$,
(e) $\mathbf{a} = [1, 2]$, $\mathbf{b} = [3, 5]$, $\mathbf{c} = [-4, -1]$,
(f) $\mathbf{a} = [1, 2]$, $\mathbf{b} = [-5, -3]$, $\mathbf{c} = [-4, 1]$.

2.3 Compute belief and plausibility functions for $x \leq y$ from Figs. 2.2 and 2.3.

2.4 Implement a simple class `interval` and basic interval arithmetic operations in one of the object-oriented programming languages. Outward rounding can be neglected in the solution of this exercise.

References

1. Benhamou, F., Granvilliers, L.: Combining local consistency, symbolic rewriting and interval methods. In: Artificial Intelligence and Symbolic Mathematical Computation, pp. 144–159 (1996)
2. Fedrizzi, M., Kacprzyk, J., Yager, R.R. (eds.): Advances in the Dempster-Shafer Theory of Evidence (1994)
3. Gardenes, E., Sainz, M.A., Jorba, L., Calm, R., Estela, R., Mielgo, H., Trepat, A.: Modal intervals. Reliab. Comput. **7**(2), 77–111 (2001)
4. Goldsztejn, A.: Modal intervals revisited, part 1: a generalized interval natural extension. Reliab. Comput. **16**, 130–183 (2012)
5. Granvilliers, L., Monfroy, E., Benhamou, F.: Symbolic-interval cooperation in constraint programming. In: Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation, pp. 150–166. ACM (2001)
6. Gutowski, M.W.: Interval straight line fitting (2001). arXiv:math/0108163
7. Gutowski, M.W.: Introduction to Interval Calculi and Methods (in Polish). BEL Studio, Warszawa (2004)
8. Gutowski, M.W.: Breakthrough in interval data fitting I. The role of Hausdorff distance (2009). arXiv:0903.0188
9. Gutowski, M.W.: Breakthrough in interval data fitting II. From ranges to means and standard deviations (2009). arXiv:0903.0365
10. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
11. Hansen, P., Jaumard, B., Xiong, J.: Decomposition and interval arithmetic applied to global minimization of polynomial and rational functions. J. Glob. Optim. **3**(4), 421–437 (1993)
12. Hurwicz, L.: Optimality criteria for decision making under ignorance. In: Cowles Commission Discussion Paper, Statistics, 370 (1951)
13. IEEE: 754-2008–IEEE standard for floating-point arithmetic (2008). http://ieeexplore.ieee.org/document/4610935/

14. IEEE: 1788-2015–IEEE standard for interval arithmetic (2015). http://standards.ieee.org/findstds/standard/1788-2015.html
15. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
16. Kaucher, E.: Interval analysis in the extended interval space $\mathbb{IR}$. In: Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis), pp. 33–49. Springer (1980)
17. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
18. Kearfott, R.B., Kreinovich, V.: Applications of Interval Computations, vol. 3. Springer Science & Business Media (2013)
19. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Vychislennyie Tiehnologii (Computational Technologies) **15**(1), 7–13 (2010)
20. Kubica, B.J., Woźniak, A.: Applying an interval method for a four agent economy analysis. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 477–483 (2012)
21. Kulisch, U.: An axiomatic approach to rounded computations. Numerische Mathematik **18**(1), 1–17 (1971)
22. Kulisch, U.: Computer Arithmetic and Validity-Theory. Implementation and Applications. De Gruyter, Berlin, New York (2008)
23. Kulisch, U.: An axiomatic approach to computer arithmetic with an appendix on interval hardware. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 484–495 (2012)
24. Kulpa, Z.: Diagrammatic representation of interval space in proving theorems about interval relations. Reliab. Comput. **3**(3), 209–217 (1997)
25. Moore, R.E.: Interval Analysis. Prentice Hall (1966)
26. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
27. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press (1990)
28. Pedamallu, C.S., Özdamar, L., Csendes, T.: Symbolic interval inference approach for subdivision direction selection in interval partitioning algorithms. J. Glob. Optim. **37**(2), 177–194 (2007)
29. Sevastianov, P.: Numerical methods for interval and fuzzy number comparison based on the probabilistic approach and Dempster-Shafer theory. Inf. Sci. **177**(21), 4645–4661 (2007)
30. Sevastjanov, P., Bartosiewicz, P., Tkacz, K.: A method for comparing intervals with interval bounds. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 496–503 (2012)
31. Sevastjanov, P., Tikhonenko, A.: Direct interval extension of TOPSIS method. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 504–512 (2012)
32. Sevastjanov, P.V., Róg, P., Venberg, A.V.: A constructive numerical method for the comparison of intervals. In: PPAM 2001 (4th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 2328, pp. 756–761 (2003)
33. Sevastjanow, P.: Interval comparison based on Dempster–Shafer theory of evidence. In: PPAM 2003 (5th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 3019, pp. 668–675 (2004)
34. Sharaya, I.A.: On the distributivity in classical interval arithmetic. Vychislennyie Tiehnologii (Computational Technologies) **2**(1), 71–83 (1997). (in Russian)
35. Shary, S.P.: Finite-Dimensional Interval Analysis. Institute of Computational Technologies, SB RAS, Novosibirsk (2013)
36. Warmus, M.: Calculus of approximations. Bulletin de l' Academie Polonaise de Sciences **4**(5), 253–257 (1956)
37. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice Hall, Englewood Cliffs, NJ (1963)

Chapter 3
Bounding Derivatives by Algorithmic Differentiation

3.1 Interval Algorithms and Derivatives Computation

On the face of it, this is not obvious that derivatives are useful in solving problems of type (1.1). Actually, there might be some instances of this problem for which derivatives would be of little help. Nevertheless, in many practical cases (cf. any of the quoted papers of the author or any textbook on interval analysis), derivatives (or slopes, or yet another equivalent of derivatives; cf., e.g., [8, 11, 17]) are very useful in interval algorithms.

In general, we can state that:

- first-order necessary conditions for several problems are formulated using derivatives, e.g., Fritz John conditions for optimization or analogous conditions for Pareto-optimization or seeking game solutions,
- the interval Newton operator makes use of some version of derivatives; this operator is the main tool in bounding solutions of equations and inequalities, which—as explained in Chap. 4—is crucial in both phases of Algorithm 1.

But to utilize derivatives (or slopes) in an interval solver, their values have to be bounded somehow. How to compute bounds on the derivatives of a function?

3.1.1 Basic Approaches

Classical *numerical differentiation* tools, based on finite-differences, are of little (or virtually no) help. They are rather inaccurate (especially, when trying to compute higher derivatives) and there is no obvious way to bound the error, which is crucial for interval methods (cf. Chap. 2).

Symbolic differentiation is more adequate, but it has several drawbacks, also. It requires a separate (and complicated) toolset to transform formulae and the resulting expressions may be complicated and impractical; cf. [11].

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_3

Algorithmic differentiation (often called "automatic differentiation") turns out to be a feasible alternative to the both above approaches (see, e.g., [9–11]).

3.2 Algorithmic Differentiation

This technique is based on the following observation: each function evaluated by a computer is described by a computer program, that consists of several elementary operations (arithmetic operations, transcendental functions, etc.). And it is known, how to compute derivatives of such elementary operations. So, we can enhance this program, so that it computed derivative(s) together with the original function.

The basic rule is called the *chain rule*:

$$\frac{\partial}{\partial x} f\big(g(x)\big) = \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial x} \,. \tag{3.1}$$

Using (3.1), it is possible to decompose complicated expressions to "atoms" that can be differentiated, and "assemble" the derivative from the "building blocks".

For instance, for basic arithmetic operations, we have the following formulae:

$$\begin{aligned}
\langle \mathbf{u}, \mathbf{u}' \rangle + \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u} + \mathbf{v},\ \mathbf{u}' + \mathbf{v}' \rangle \,, \\
\langle \mathbf{u}, \mathbf{u}' \rangle - \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u} - \mathbf{v},\ \mathbf{u}' - \mathbf{v}' \rangle \,, \\
\langle \mathbf{u}, \mathbf{u}' \rangle \cdot \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u} \cdot \mathbf{v},\ \mathbf{u} \cdot \mathbf{v}' + \mathbf{u}' \cdot \mathbf{v} \rangle \,, \\
\langle \mathbf{u}, \mathbf{u}' \rangle \,/\, \langle \mathbf{v}, \mathbf{v}' \rangle &= \langle \mathbf{u}/\mathbf{v},\ (\mathbf{u}' \cdot \mathbf{v} - \mathbf{u} \cdot \mathbf{v}')/\mathbf{v}^2 \rangle \,.
\end{aligned}$$

Other operations, e.g., power or transcendental functions, can be extended in an analogous manner, e.g.:

$$\langle \mathbf{u}, \mathbf{u}' \rangle^n = \big\langle \mathbf{u}^n, n\mathbf{u}^{n-1}\mathbf{u}' \big\rangle$$

Let us consider a simple example, to make the things more explicit.

Example 3.1

$$\begin{aligned}
&f(x) = x^2 - 3x + 2 \,, \\
&\mathbf{x} = [-1, 2] \,, \\
&\big\langle \mathbf{x}, \mathbf{x}' \big\rangle = \big\langle [-1, 2], [1, 1] \big\rangle \,, \\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \mathsf{f}\Big(\langle \mathbf{x}, \mathbf{x}' \rangle \Big) \,, \\
&\mathsf{f}\Big(\langle \mathbf{x}, \mathbf{x}' \rangle \Big) = \big\langle \mathbf{x}, \mathbf{x}' \big\rangle^2 - 3 \cdot \big\langle \mathbf{x}, \mathbf{x}' \big\rangle + \big\langle 2, 0 \big\rangle \,, \\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \big\langle [-1, 2], [1, 1] \big\rangle^2 - 3 \cdot \big\langle [-1, 2], [1, 1] \big\rangle + \big\langle 2, 0 \big\rangle \,, \\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \big\langle [0, 4], [-2, 4] \big\rangle + \big\langle [-6, 3], [-3, -3] \big\rangle + \big\langle [2, 2], [0, 0] \big\rangle \,, \\
&\langle \mathbf{y}, \mathbf{y}' \rangle = \big\langle [-4, 9], [-5, 1] \big\rangle \,.
\end{aligned}$$

We can consider also computing higher derivatives—then the record has more fields, e.g., $\langle \mathbf{u}, \mathbf{u}', \mathbf{u}'', \mathbf{u}''' \rangle$; the formulae are analogous (they may become quite complicated for higher derivatives, though).

The differentiated function may be multivariate—then higher derivatives are represented by some containers: vectors, matrices, etc., instead of singleton values.

3.3 Implementation of AD

Algorithmic differentiation can be implemented in a few manners. The above schema is called the *forward mode algorithmic differentiation* (see, e.g., [11]). It can be realized either by operator overloading or using so-called dual numbers (cf. [9]). An alternative schema, is called the *reverse mode algorithmic differentiation*. Let us present these approaches, briefly.

3.3.1 Forward AD with Operator Overloading

This approach can be used in object-oriented programming—if only they allow operator overloading. We need to create a class representing an algebraic expression. Actually, this seems to be the most common approach; the author's library ADHC, presented below, in Sect. 3.4.1, adopts this approach, as well. Hence, we defer the presentation of details, until then.

3.3.2 Forward AD with Dual Numbers

Dual numbers, introduced by Clifford already in XIX century, are conceptually similar to complex numbers. We have:

$$x = a + b \cdot \varepsilon \,,$$

where $\varepsilon \notin \mathbb{R}$. The difference with complex numbers is that $\varepsilon^2 = 0$ and not -1.

Similarly to complex numbers, each dual number has a conjugate one:

$$(a + b \cdot \varepsilon) \cdot (a - b \cdot \varepsilon) = a^2 + (ab - ab) \cdot \varepsilon + b^2 \cdot \varepsilon^2 = a^2 \,.$$

What is the interpretation of ε? Actually, because $\varepsilon^2 = 0$, it can be interpreted as "a value close to zero", i.e., an approximation of the *infinitesimal number*.

We can simply obtain the arithmetic rules for dual numbers:

$$
\begin{aligned}
(a + b \cdot \varepsilon) + (c + d \cdot \varepsilon) &= (a + b) + (c + d) \cdot \varepsilon \,, \\
(a + b \cdot \varepsilon) - (c + d \cdot \varepsilon) &= (a - b) + (c - d) \cdot \varepsilon \,, \\
(a + b \cdot \varepsilon) \cdot (c + d \cdot \varepsilon) &= (ac) + (ad + bc) \cdot \varepsilon \,, \\
(a + b \cdot \varepsilon) / (c + d \cdot \varepsilon) &= \frac{a}{c} + \frac{ad - bc}{c^2} \cdot \varepsilon \,.
\end{aligned}
$$

Similarity to the rules of differentiation is obvious. Obviously, using the Taylor series expansion, proper definitions of transcendental functions can be provided, as well.

As this approach to AD is elegant and relatively easy to implement, it seems to have few advantages with respect to the former approach: the direct use of operator overloading. There exist some software packages using dual numbers ([9] presents some Haskell code), but none of them (at least, to the best knowledge of the author) uses interval data types.

3.3.3 *Reverse Mode AD*

The reverse mode is more efficient (at least when differentiating functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m < n$), but also more difficult and cumbersome to implement and to use.

As the forward mode, the reverse mode is based on (3.1), but the accumulation is performed in the opposite direction. We need to store all intermediate quantities (in a so-called Wengert's list) and we compute their values, by solving an equations system. It will be the simplest to explain it, using the example we already know.

Example 3.2 As in Example 3.1, let us consider the function:

$$
\begin{aligned}
f(x) &= x^2 - 3x + 2 \,, \\
\mathbf{x} &= [-1, 2] \,,
\end{aligned}
$$

Its expression can be decomposed to the following form:

$$
\begin{aligned}
x_1 &= x \,, \\
x_2 &= x_1^2 \,, \\
x_3 &= 3 \cdot x_1 \,, \\
x_4 &= x_2 - x_3 \,, \\
x_5 &= x_4 + 2 \,.
\end{aligned}
$$

Derivatives of all terms are:

$$\begin{aligned} x_1' &= 1\,, \\ x_2' &= 2 \cdot x \in [-2, 4]\,, \\ x_3' &= 3 \cdot x_1'\,, \\ x_4' &= x_2' - x_3'\,, \\ x_5' &= x_4'\,. \end{aligned}$$

This equations system can be written in the matrix form:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ -3 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \\ x_5' \end{pmatrix} = \begin{pmatrix} 1 \\ [-2, 4] \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

The result of solving the above system is:

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \\ x_5' \end{pmatrix} = \begin{pmatrix} 1 \\ [-2, 4] \\ 3 \\ [-5, 1] \\ [-5, 1] \end{pmatrix},$$

which is consistent with the result obtained in Example 3.1.

As it has been stated, the reverse mode AD is more difficult and cumbersome than the forward mode and, although it is used by some researcher, it will not be considered further, in this monograph.

3.4 State-of-the-Art Libraries

It is not simple to find an appropriate library for AD. The most popular packages, like ADOL-C [6], are not compatible with using interval data types and arithmetic. For several years, the author has been using the code, from C-XSC library [1]. Unfortunately, this package has several drawbacks:

- there are distinct classes (`GradType`, `HessType`, `DerivType`), implemented in distinct files (`grad_ari.cpp`, `hess_ari.cpp`, `ddf_ari.cpp`) for computing first or second derivatives and for univariate or multivariate functions;
- there are global variables (`GradOrder`, `HessOrder`, `DerivOrder`) to distinguish the order of computed derivative—these variables have to be checked

at run-time, several times during the computation; they also affect multithreaded implementations [15];
- extended interval division (for divisors containing zero) is not supported in the algorithmic differentiation library;
- computing derivatives of higher order would require to implement a separate (but analogous) class;
- although, the developers of C-XSC have provided several useful classes for sparse matrices and vectors, their AD code makes no use of it;
- the compound assignment operators (e.g., "+="), that might have a significant impact on performance of the code, have not been implemented for AD classes.

Fortunately, there is a way to overcome all (or at least most) of these issues, yet it is necessary to apply template metaprogramming.

3.4.1 ADHC Library

In 2016, the author has provided a novel automatic differentiation library, based on C++ templates (see, e.g., [7]). The package has been named ADHC, which stands for Algorithmic Differentiation and Hull Consistency [2].

Virtues of template meta-programming allowed obtaining several useful features of the ADHC library. This includes efficiency and versatility. The same source code can be used to generate (with no penalty at runtime) distinct procedures for computing function values, gradients, Hesse matrices and—potentially—higher derivatives. Also, we can use the same source code to differentiate uni- and multivariate functions and to use sparse or dense representations of vectors and matrices of partial derivatives. And C-XSC library provides us pretty nice implementations of sparse vectors and matrices (cf. [12]), that can directly be applied in ADHC.

Proper types are generated, using the so-called typelist (see, e.g., [7]). They are specializations of the following template:

```
template<int level, sparsity_t sparse_mode, int num_vars>
struct adhc_ari {
    // ...
};
```

The three template arguments are:

- `level`—information on what should be computed; number of computed derivatives (for nonnegative values) or construction of the syntactic tree (for value -1),
- `sparse_mode`—should sparse or dense matrix/vector representation be used,
- `num_vars`—the number of variables.

Please note, `num_vars` is an argument of the template, so expressions using potentially different number of variables are of inherently different types. Hence, naïvely

performing an operation on such incompatible objects will be detected at compile-time already. For instance, the following code:

```
adhc::adhc_ari<2, sparse, 2> x;
adhc::adhc_ari<2, sparse, 3> y;
//...
z = x + y;
```

will not compile. In contrast to that, the AD code from C-XSC library has to use a dedicated function `TestSize()` while performing virtually any arithmetic operation, which results in a certain overhead at runtime.

The type of the second template argument, `sparse_mode`, is an instance of enumerable type `sparsity_t` and can have three values:

```
typedef enum sparsity_t {dense, sparse, highly_sparse};
```

As it has already been stated, the C-XSC library contains useful classes for sparse matrices and vectors (cf. [12]): `cxsc::srvector`, `cxsc::scvector`, `cxsc::sivector`, `cxsc::scivector`, `cxsc::srmatrix`, `cxsc::scmatrix`, `cxsc::simatrix`, `cxsc::scimatrix`.

Sparse vectors are represented as a pair of `std::vector` objects: one storing values and the other one—indices of non-zero elements. Sparse matrices are stored in a *compressed column storage* format. Details can be found, i.a., in [14].

For instance, the function considered in previous examples, in ADHC could look as follows:

```
const int lev = 1;
const int n = 1;

adhc_ari<lev, sparse_mode, n>
f(const adhc_ari<lev, sparse_mode, n> &x) {
    adhc_ari<lev, sparse_mode, n> result;
    result = sqr(x) - 3.0*x + 2.0;
    return result;
}
```

The library has been used in the HIBA_USNE solver [5] and in the solver for game solutions, described in [13] (cf. also Sect. 6.3); the author plans to use it in other solvers, also.

3.4.2 Computing Arbitrary Many Derivatives

The current implementation of ADHC does not allow computing derivatives of order higher than two. This is caused by two issues that require solution:

- implementation of classes representing higher-dimensionality tensors, not only vectors and matrices;
- providing universal enough formulae for higher derivatives of various operations.

But even after these issues will have been solved, users of ADHC will need to decide at compile-time, how many derivatives they would like to compute. Actually, this is an inherent limitation of template meta-programming: conditions have to be known at compile-time.

Actually, for most problems, it is reasonable to assume, we make such decisions prior to running the solver. Yet, in some cases, it would be beneficial to delay the decision on the number of computed derivatives, until runtime. An example of such situation is seeking local optima of the function $f(x) = x^{2 \cdot n}$ (or a similar function, having a very flat optimum—with several derivatives equal to zero). If we do not know n at compile-time, we shall not know how many derivatives to compute.

It would seem beneficial to be able to increase the number of computed derivatives subsequently, during runtime. Unfortunately, according to the best knowledge of the author, no currently available package allows this; ADHC is no exception to this rule.

How could such a procedure be implemented? Obviously, it is possible, but rather cumbersome. For instance, sticking to types of the C-XSC library [1], we could have a list, say `std::vector<>` of pointers to some abstract class, representing an arbitrary derivative value. We would need to define this abstract class, because classes representing a single interval, interval vectors, matrices, etc. are not related in the inheritance hierarchy. Hence, we would have to use some wrapper-classes, all derived from the abstract base class, that we would define.

Yet another possibility is to represent the list as `std::vector<void *>` and cast the pointer to a `void` to a pointer to specific classes, using the `reinterpret_cast<>`.

Both approaches seem inconvenient, but feasible.

The new C++17 standard offers us yet another, more proper, possibility: the type `std::any` [4]. A variable of this type can store a value of arbitrary-type; it is similar to a C-style pointer to a `void`, but safe casts are available, using the `std::any_cast` operator. Unfortunately, the C++17 standard is not widely supported, yet, but Boost libraries [3] offer a similar concept of `boost::any`.

Nevertheless, it is questionable, if computing higher derivatives, using automatic differentiation, would really be useful. The computed values would be very likely to be highly overestimated. Possibly, using Taylor arithmetic [16] in such case would be a better approach. Verifying it experimentally, would be pretty interesting.

3.5 Summary

This survey of AD techniques was very brief and incomplete. Nevertheless, even such a brief description was necessary for the completeness of this monograph, as AD techniques are very often used together with interval analysis (cf. the textbooks [10, 11]). In all author's experiments, his ADHC library will be used.

References

1. C++ eXtended Scientific Computing library (2015). http://www.xsc.de
2. ADHC, C++ library (2017). https://www.researchgate.net/publication/316610415_ADHC_Algorithmic_Differentiation_and_Hull_Consistency_Alfa-05
3. Boost C++ libraries (2017). http://www.boost.org/
4. C++ documentation for `std::any` (2017). http://en.cppreference.com/w/cpp/utility/any
5. HIBA_USNE, C++ library (2017). https://www.researchgate.net/publication/316687827_HIBA_USNE_Heuristical_Interval_Branch-and-prune_Algorithm_for_Underdetermined_and_well-determined_Systems_of_Nonlinear_Equations_-_Beta_25
6. ADOL-C: ADOL-C library (2016)
7. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
8. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
9. Hoffmann, P.H.: A hitchhiker's guide to automatic differentiation. Numer. Algorithms **72**(3), 775–811 (2016)
10. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
11. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
12. Krämer, W., Zimmer, M., Hofschuster, W.: Using C-XSC for high performance verified computing. In: PARA 2010 Proceedings. Lecture Notes in Computer Science, vol. 7134, pp. 168–178 (2012)
13. Kubica, B.J.: Advanced interval tools for computing solutions of continuous games. Vychislennyie Tiehnologii (Computational Technologies) **23**(1), 3–18 (2018)
14. Kubica, B.J., Kurek, J.: Interval arithmetic, hull-consistency enforcing and algorithmic differentiation using a template-based package. In: CPEE 2018 Proceedings (2018)
15. Kubica, B.J., Woźniak, A.: A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment. In: PARA 2008 Proceedings. Lecture Notes in Computer Science, vol. 6126/6127. Accepted for Publication (2010)
16. Nedialkov, N., Kreinovich, V., Starks, S.A.: Interval arithmetic, affine arithmetic, Taylor series methods: why, what next? Numer. Algorithms **37**(1), 325–336 (2004)
17. Shary, S.P.: Finite-dimensional Interval Analysis. Institute of Computational Technologies, SB RAS, Novosibirsk (2013)

Chapter 4
Branch-and-Bound-Type Methods

4.1 Preliminary Remarks

This chapter presents the main application of interval methods (at least in the author's opinion): seeking points that satisfy a certain logical condition. The generic algorithm to solve such problems is presented: the interval branch-and-bound-type algorithm. This generic algorithm has several instances, in particular:

- classical B&B methods, used in optimization (e.g., [15]), but also other problems (e.g., seeking Pareto sets [28] or Nash equilibria [29]);
- branch-and-prune methods (B&P)—for systems of equations and/or inequalities [10, 12, 13];
- partitioning parameter space (PPS)—for interval linear systems [45];
- SIVIA (Set Inversion Via Interval Analysis)—for various constraint satisfaction problems (CSPs) [13, 14];
- …

Presentation in this chapter tries to be as general and abstract as possible. Readers who are not familiar with any of the above approaches might find this presentation hard to follow. Possibly, it would be more convenient for them to have a look at Chaps. 5 and 6 first.

4.2 Introduction

Let us recall Problem (1.1):

$$\text{Find } \textit{all } x \in X \text{ such that } P(x) \text{ is fulfilled.}$$

Here, $P(x)$ is a formula with a free variable x and $X \subseteq \mathbb{R}^n$; often X is a single box $\mathbf{x}^{(0)}$ (as in other chapters, the standard notation from [16] is adopted).

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_4

In what formal system is P a formula? We shall not answer this interesting open question here. Instead, let us present a few well-known problems that can be formulated as instances of (1.1).

The problem of solving interval linear systems is often formulated that way (e.g., [2, 43, 44]). Also:

- a system of (nonlinear) equations: $\{x \in X \mid f(x) = 0\}$,
- a constraint satisfaction problem (CSP) with inequality constraints: $\{x \in X \mid f(x) \in [\underline{y}, \overline{y}]\}$,
- global optimization: $\{x \in X \mid (\forall t \in X) \big(f(x) \leq f(t)\big)\}$,
- seeking Pareto set of a multicriterion problem: $\{x \in X \mid (\forall t \in X) \big((\forall i = 1, \ldots, N) (f_i(x) \leq f_i(t)) \text{ or } (\exists j \in [1..N])\ (f_j(x) < f_j(t))\big)\}$,
- seeking Nash equilibrium points of a non-cooperative game: $\{x \in X \mid (\forall i = 1, \ldots, N)\ (\forall t_i \in \mathbf{x}_i \subseteq \mathbb{R}^{k_i})\ (f_i(x_{\setminus i}, t_i) \geq f_i(x))\}$.

Some other, less commonly studied, examples include:

- seeking all optima of a function—local and global ones (studied, e.g., for a very specific case in [47]): $\{x \in X \mid (\exists \varepsilon > 0)\ \big((\forall t \in X) \text{ and } (d(x,t) < \varepsilon)\ (f(x) \leq f(t))\big)\}$,
- seeking local (but non-global) minimizers of a function: $\{x \in X \mid (\exists \varepsilon > 0)\ \big((\forall t \in X) \text{ and } (d(x,t) < \varepsilon)\ (f(x) \leq f(t))\big) \text{ and } \big((\exists s \in X)\ (f(s) < f(x))\big)\}$,
- seeking local minimizers of a function, for which the optima have a sufficiently large "attraction basin", e.g., containing at least an interval of the given range: $\{x \in X \mid (\exists \varepsilon > \delta)\ \big((\forall t \in X) \text{ and } (d(x,t) < \varepsilon)\big)\ (f(x) \leq f(t))\}$,
- and many other similar problems.

The quantity $d(\cdot, \cdot)$, used in above formulae, denotes the metric.

All above problems are instances of (1.1) and can be solved using a similar algorithm, that will be described in this Chapter (cf. also [22, 27]).

It is worth noting that Problem (1.1) has some variants, e.g.:

- Find *a single* solution point $x \in X$ such that $P(x)$.
- Find *the inner approximation* of the solution set, i.e., a box $\mathbf{x} \subseteq X$ such that for all $x \in \mathbf{x}$ we have $P(x)$.
- Find *the outer approximation* of the solution set, i.e., a box $\mathbf{x}$ such that all $x \in X$ for which $P(x)$ satisfy $x \in \mathbf{x}$.

Above problems, often being solved, e.g., for interval linear equations [43, 44], are not going to be considered in this monograph.

As for problems of type (1.1) in its original form, the author has formulated a general meta-algorithm to solve them. Let us call it the *branch-and-bound-type* method (B&BT method), or *generalized branch-and-bound method*.

This meta-algorithm has several well-known instances for solving various problems (see Chaps. 5 and 6, and references therein). All these algorithms differ in some significant details, but also they have several similarities:

- they are based on subsequent subdivision of the search domain, i.e., they are instances of the so-called divide-and-conquer approach (which is an inexact translation of the Latin phrase *divide et impera*);

- they bound values of some functions on obtained subboxes, using the interval calculus;
- they use the same kind of tools to process subboxes, e.g., interval Newton operators, consistency enforcing operators, linear relaxations, initial exclusion phases, etc. (see, e.g., [13, 15]);
- they can be parallelized in a similar manner;
- they face similar problems in storing and distributing boxes, collecting and storing results, load balancing, etc. [22].

Actually, there is some confusion in naming the algorithms: for instance, Kearfott in his classical book [15] calls "a branch-and-bound method" the procedure that is called "branch-and-prune" by other researchers (see, e.g., [10, 12]). It seems useful to indicate actual common features and differences between these algorithms.

4.3 The Solution Set

What can the interval branch-and-bound procedure compute? Usually, two lists of boxes are constructed; and so do the pseudocodes presented below, in Sect. 4.4:

- *verified solutions*—boxes that are *guaranteed* to contain some points within the solution set,
- *possible solutions*—boxes that (under the given accuracy) could not be verified either to contain solutions or not to contain them.

What specifically does the list of verified boxes contain? It is related to the geometry of the solution set and hence problem-dependent.

In general, the solution set can have three forms:

- a countable set (a set of isolated points)—then verified solutions are boxes guaranteed to contain a single solution (see Fig. 4.1),
- an uncountable set with an empty interior (a manifold)—then verified solutions are boxes guaranteed to contain a segment of the solution set (see Fig. 4.2),
- an uncountable set with a nonempty interior—then verified solutions are boxes guaranteed to lie in the interior of the solution set, i.e., to contain only actual solutions (see Fig. 4.3).

Please note that each of the above possibilities is quite often encountered in practical problems:

- a countable set is the solution to (non-singular) global optimization problems and well-determined nonlinear equations systems,
- an uncountable set with an empty interior is the solution to Pareto set seeking problems and underdetermined systems of equations,
- an uncountable set with a nonempty interior is often the solution to constraint satisfaction problems with inequality constraints.

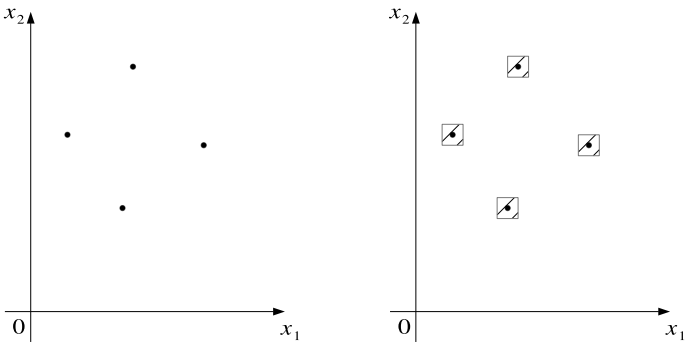

Fig. 4.1 Solution set consisting of separate points

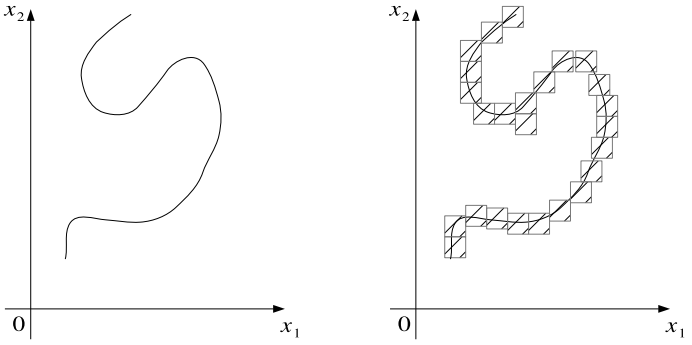

Fig. 4.2 Solution set of measure zero—a manifold

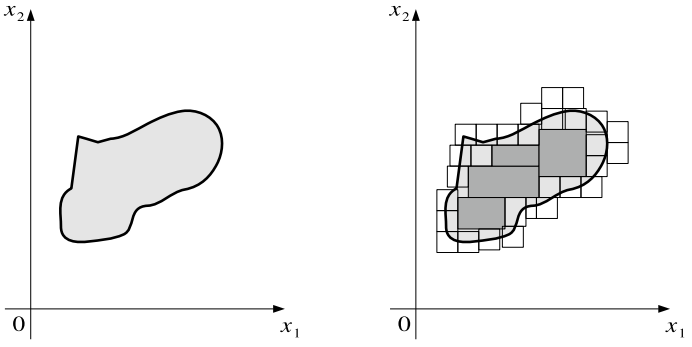

Fig. 4.3 Solution set with a non-empty interior

In any case, "possible" solutions are small boxes (usually their diameter is smaller than some predefined accuracy parameter ε; cf. Algorithm 2, line 16) that have not been proved either to contain a solution(s) or not to contain any.

Now, that we know what the algorithm is supposed to compute, let us present it in the from of a pseudocode.

4.4 Generic Algorithm

The main algorithm to solve problems of type (1.1), the B&BT method, can be expressed by Algorithm 1.

Algorithm 1 The overall algorithm

Require: L, P
1: perform the essential B&BT method (i.e., Algorithm 2) for (L, P), storing the results in L_{ver}, L_{pos}, L_{check}
2: {The second phase}
3: perform the verification (i.e., Algorithm 3) for L_{ver}, L_{check}, P
4: perform the verification (i.e., Algorithm 3) for L_{pos}, L_{check}, P

This algorithm consists of two phases: the actual B&BT method (Algorithm 2) and the second phase, when the results are verified (if it is necessary; Algorithm 3). Why and when do we need the verification in the second phase, will be explained in Sect. 4.6.

Algorithm 2 The essential generalized branch-and-bound method

Require: L, P
$L_{ver} = L_{pos} = L_{check} = \emptyset$
x = pop (L)
loop
  process the box **x**, using the rejection/reduction tests
  update the *shared quantities* (if any; see explanation in Sect. 4.6.2)
  if (**x** does not contain solutions) **then**
    if CHECK(P, **x**) **then**
      push (L_{check}, **x**)
    discard **x**
  else if VERIF(P, **x**) **then**
    push (L_{ver}, **x**)
  else if (the tests resulted in two subboxes of **x**: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) **then**
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push (L, $\mathbf{x}^{(2)}$)
    cycle loop
  else if (**x** is small enough) **then**
    push (L_{pos}, **x**)
  if (**x** was discarded **or** **x** was stored) **then**
    x = pop (L)
    if (L was empty) **then**
      return L_{ver}, L_{pos}, L_{check}
  else
    bisect (**x**), obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push (L, $\mathbf{x}^{(2)}$)

Algorithm 3 Verification of solutions

Require: L_{sol}, L_{check}, P
1: **for all** ($\mathbf{x} \in L_{sol}$) **do**
2: discard $\mathbf{x}$ if it contains no points $x \in \mathbf{x}$, satisfying $P(x)$
3: {details of the verification depend on P, but the *shared quantities* and, possibly, the boxes from L_{check} are useful there; cf. Sect. 4.6}

Words "push" and "pop" in Algorithm 2 name operations of inserting and removing elements to/from the set (independent of the representation of the set—it can be a stack, a queue or a more sophisticated data structure). This depends on the problem under consideration and other features of the specific implementation.

The following notation is used in the algorithms:

- L—the list/set of initial boxes, often containing a single box $\mathbf{x}^{(0)}$;
- $P(x)$—the predicate formula, defining the problem under solution;
- the lists/sets of solutions: L_{ver}—verified solution boxes and L_{pos}—possible solution boxes;
- L_{check}—the list/set of boxes (possibly, with some additional information) representing the *shared quantities*, that will be used in the second phase to verify boxes from L_{ver} and L_{pos}; for examples see Sect. 4.6.3 or [33], where seeking strong Nash equilibria is considered;
- VERIF(P, $\mathbf{x}$) states that the box $\mathbf{x}$ has been verified to contain a solution (or a point satisfying some necessary conditions to be a solution); by a solution we mean a point x satisfying $P(x)$;
- CHECK(P, $\mathbf{x}$) states that the box $\mathbf{x}$ does not contains a solution, yet it can be useful to verify P for some other box in the second phase.

In general, CHECK and VERIF are predicates in a second-order logic: they depend on P, so they are functions of a formula. Obviously, it might be pretty difficult to develop them for some P (cf., e.g., [22, 29, 33] for examples). In the remainder, we discuss this issue in more details.

Remark 4.1 A comment is necessary about subdividing a box. In lines 13–14 and 24–25 of Algorithm 2 it is implied that we always store one of the resulting boxes and process the other one in the next step.

Often, this is so, but there are exceptions to this rule. There might be various policies for box selection and for some algorithm versions, we might push both boxes $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ and pop the new value of $\mathbf{x}$. A good example is global optimization: cf. pseudocodes and remarks in Sect. 6.1.1. Actually, such a formulation might be considered more general to what is presented in Algorithm 2.

Nevertheless, such a situation is—according to the author's experience—relatively rare and, for several algorithm versions (in particular, for multithreaded implementations), it is crucial to reduce the number of push/pop operations. So, the author considers the above presentation to be proper, even if requiring this comment. We shall consider this problem also in Chaps. 6 and 7.

Remark 4.2 For some algorithms, we might need bisection in the second phase, as well as in the first one. This is the case, e.g., for some algorithms computing the Pareto-sets, e.g., [31]; precisely, this may happen when the computation of *shared quantities* does not require exhaustive search of X. We do not reflect this possibility in the pseudocode of Algorithm 3; such situations seem rare, but the possibility requires mentioning.

Remark 4.3 As we have been able to formulate a generic algorithm to solve problems of type (1.1), it follows that, under some technical assumptions about P, such problems are computable (see also [9, 18] and references therein). Probably, they are often NP-hard (cf., e.g., [19, 41] and, especially, the book of Kreinovich et alii [20]), but we are still able to devise proper heuristics [46]. We shall get back to that in Sect. 4.9.

4.5 Analysis of the B&BT Algorithm

What happens in both phases of the algorithm? This varies to some extent, depending on the problem under solution. Yet, in general, we can distinguish two objectives of the first phase:

- removing boxes that do not satisfy some necessary conditions of P (see Sect. 4.7),
- computing some quantities that will be necessary for solutions verification, in the second phase; they will be called *shared quantities* in the remainder (see Sect. 4.6.2).

The importance of these two objectives varies, depending on the specific problem being addressed; as we shall see, for some problems one of these tasks might even be irrelevant.

For global optimization, most of the work is performed in the first phase—we check first-order necessary conditions of optimality (usually: Fritz John conditions [15]) and also we check if $\underline{y} > y^*$, where the upper bound on the global minimum y^* is successively improved.

Hence, for the problem of approximating the whole ε-optimality region, we can discard few boxes in the first phase. ε-optimal points do not have to satisfy any first-order necessary conditions (they are solutions of an inequality!) and y^* is still overestimated in this phase. The main objective of the first phase of the algorithm solving this problem, is—actually—to compute y^* as precisely as possible; boxes will be rejected in the second phase.

For other problems, we have to compute much more complicated quantities for use in the second phase: e.g., for a multicriteria problem, we have to obtain a set of boxes representing the Pareto frontier (see [32] and references therein). These boxes are inverted in the second phase, using another B&BT procedure.

An analogous situation is encountered for computing (strong) Nash equilibria of a game [33].

Obviously, for many B&BT methods the second phase is not necessary. For what problems is it the case and why? This question will be addressed in the next section. But to answer it, let us describe in more details, what operation is performed in the second phase. Actually, it is some kind of *quantifier elimination*—but a very specific kind.

4.6 The Second Phase—Quantifier Elimination

So, why and when do we need the second phase? As already stated, for some problems (equations systems, CSPs) the first phase suffices, but for other ones (e.g., global optimization) the second phase is inevitable. Please note that for solving equations or inequalities systems, formula P is non-quantified.

Actually, solving equations and inequalities is the task best suited for interval methods. Using classical interval tools (including the Newton operator), we can verify boxes containing the solutions—in both cases; and for too wide boxes, we can subdivide them.

But how do we apply interval methods for problems of type (1.1) with quantified P? What we need there is *quantifier elimination* and the partition of Algorithm 1 into two phases is related to this task. To the best knowledge of the author, this fact has not been recognized before (not to count the author's paper [27]).

Other words, in the first phase (apart from discarding boxes that do not satisfy necessary conditions), we need to obtain some quantities that would allow to verify $P(x)$ for boxes $\mathbf{x} \ni x$, *using equations and inequalities, only*. We refer to them as *shared quantities* as they are stored independently of the boxes and often they can be used to verify several of these boxes.

For instance, in global optimization, we need to compute a single *shared quantity* in the first phase: an upper bound on the global minimum. In the problems of Pareto sets or Nash equilibria seeking, we need to extract more *shared quantities* in the first phase (the set of approximate Pareto-optimal points, etc.).

But what *shared quantities* are (in general) necessary to verify a specific predicate P?

To give a comprehensive answer to this, we have to introduce the notion of obtaining the *Herbrand expansion* of P; see, e.g., [6] and references therein; see also [1].

4.6.1 Herbrand Expansion

Let us consider formula P from (1.1), having the form either $P(x) \equiv (\forall t \in X)\ P'(x,t)$ or $P(x) \equiv (\exists t \in X)\ P'(x,t)$.

If the quantifier is universal, we have to verify that for all values in the domain a property holds. If the quantifier is existential, we have to find a value for which the property holds.

This is related to obtaining the *Herbrand expansion* of the quantified formula P, i.e., transforming it to a non-quantified alternative (or conjunction) of formulae for specific values $t_1, t_2, \ldots, t_k$. Formula "$\exists t$ such that $P'(t)$" can be transformed into a Herbrand disjunction: "$P'(t_1) \vee P'(t_2) \vee \ldots \vee P'(t_k)$". Hence formula "$\forall t$ we have $P'(t)$" can be transformed into a Herbrand conjunction: "$P'(t_1) \wedge P'(t_2) \wedge \ldots \wedge P'(t_k)$".

In the original theorem of Jacques Herbrand, these expansions have been used to determine the provability of a formula (see, e.g., [6]). Yet, they can be used to approximate the formulae, also. Actually, while these Herbrand expansions are not equivalent to the original formulae, they can provide good *approximations*, if the values $t_1, \ldots, t_k$ are chosen properly.

What are these "proper" values of $t_1, \ldots, t_k$ and how many of them do we need (i.e., what is k)? This strongly depends on P. In general, the t_i's depend also on x; we have $t_1(x), \ldots, t_k(x)$ and we seek:

$$x \in X \text{ such that } P'\big(x, t_1(x)\big) \wedge P'\big(x, t_2(x)\big) \wedge \ldots \wedge P'\big(x, t_k(x)\big) \,. \tag{4.1}$$

The above conjunction is obtained for the universal quantifier; for an existential one, we would use a disjunction.

For specific problems, the structure of (4.1) might get pretty simple; in particular values of t_i are often independent on x, $t_i(x) = t_i$. A good example is global optimization, where a single value t (the approximate global minimizer) is sufficient (actually, we need to store $y^* = f(t)$, only). For other problems more t_i's are needed. We get back to this topic in Sect. 4.6.2. Now, let us consider the relation between the original formula and its non-quantified form.

Relation between a formula and its Herbrand disjunction/conjunction.

As it has already been mentioned, a quantified formula is not (in general) equivalent to its Herbrand form. Actually, the relations are as follows:

$$\begin{aligned} \forall t \; P'(t) &\implies P'(t_1) \wedge P'(t_2) \wedge \ldots \wedge P'(t_k) \,, \\ \exists t \; P'(t) &\impliedby P'(t_1) \vee P'(t_2) \vee \ldots \vee P'(t_k) \,. \end{aligned}$$

What does that mean?

For a universal quantifier, the transformed formula is weaker than the initial one. We are not able to verify the initial problem strictly, but a weaker one, e.g., solving a global optimization problem, actually, we seek ε-optimal points, satisfying some necessary optimality conditions (see also [9, 18]). It is up to us to choose points $t_1, \ldots, t_k$ so that weakening of the original problem is as small as possible.

Hence, for an existential quantifier, the transformed formula is *stronger* than the initial one. We can verify the initial formula directly—at least for some points—but values of $t_1, \ldots, t_k$ have to be chosen carefully, so that as many solutions could be verified, as possible.

What can be the structure of formula P?

Up to now, we have considered computing the Herbrand expansion of the formula P starting with a universal or existential quantifier. Also, we know that for a non-quantified P, computing the Herbrand expansion is not necessary; we can consider such P its own Herbrand expansion.

What about other cases? Can P start with a non-quantified sub-formula, but contain quantifiers in the remainder? In general, it could, but we can assume P to be in the *prenex normal form*, i.e., a sequence of quantifiers followed by a non-quantified expression.

This case is sufficient, as all first-order formulae can be transformed to such form (see, e.g., [1], Theorem 7.1.9). So, we only need to consider P in the prenex normal form (cf. [18]).

Example 4.1 Let us consider the following problem:

$$\text{Find all elements of } \left\{x \in X \mid (g(x) \le 0) \implies (\forall t \in X)\big(f(x) \le f(t)\big)\right\}. \quad (4.2)$$

In this case, the formula P is not in the prenex normal form. Yet it can easily be converted to such form: we make use of the fact that the implication $(p \implies q)$ is equivalent to $\neq p \vee q$ and that the quantifier can be taken before the implication if the predecessor is independent on the bound variable. Hence, we obtain:

$$\text{Find all elements of } \left\{x \in X \mid (\forall t \in X)\Big((g(x) > 0) \vee \big(f(x) \le f(t)\big)\Big)\right\}. \quad (4.3)$$

4.6.2 Shared Quantities

Now, let us explain the notion of *shared quantities*, which we have been using before. Actually, to verify a box to contain (or not to contain) a solution, we need the values $t_1, \ldots, t_k$ for the Herbrand conjunction/disjunction.

So, these values can be considered the *shared quantities* from Algorithm 2. Please note, these values can be stored in the list L_{check}, mentioned in Algorithms 2 and 3. Yet, not necessarily should these values be represented explicitly.

For several problems, it is some function of t_i's, and not them themselves, that we are interested in. The simplest example is global optimization. The Herbrand representation of this problem would be as follows:

$$\text{Find } all\ x \in X \text{ such that } \big(f(x) \le f(t)\big),$$

where t is the approximate global optimizer, i.e., the best point found. But what we actually need for verification is $y^* = f(t)$ and not t itself; y^* can be computed from t, but it takes time and is unnecessary.

Obviously, the same applies to the problem of seeking Pareto-optimal solutions of a multicriteria problem; just we have several *shared quantities* then.

Hence, for seeking (ordinary or strong) Nash equilibria of a game, t_i's have to be represented directly: verification of an equilibrium requires comparison of its values with values at specific points of the domain (see [29, 33], for details).

To sum up, we can state that:

- In theory, the *shared quantities* can always be represented by the list L_{check}.
- In practice, rarely is this list directly used.
- Usually, some more specific quantities are kept instead of the list L_{check}, e.g., the values of (or bounds on) some function of points from boxes that would be stored in L_{check}.

What quantities should be used for a specific problem? It seems, this has to be decided for each problem individually (cf. also Sect. 7 of [22]). Also, finding the proper formulation can hardly be automated.

Investigating general conditions for simplification of the Herbrand form, might be an interesting subject of future investigations.

4.6.3 Existentially Quantified Formulae

All well-known problems mentioned so far, had P starting with a universal quantifier. Problems with existential quantifiers:

$$\text{Find } all\ x \in X \text{ such that } (\exists t \in X)\ P'(x,t) \text{ is fulfilled,}$$

seem to be less frequently encountered. As an example—possibly an artificial one—let us consider, as in [27], the problem of seeking all points from the domain of function f, such that the function values are identical as in some other point, i.e., points where f is *not* an injection:

$$\text{Find } all\ x \in X \text{ such that } (\exists t \in X)\ (t \neq x)\ \big(f(x) = f(t)\big)\,.$$

How to solve it? Certainly, we need two phases: in phase 1 we partition X into several boxes $\mathbf{x} \subseteq X$ and we compute for each of them both inner and outer approximations of $\mathsf{f}(\mathbf{x})$. The twins arithmetic might be convenient here (see, e.g., [38]). The place where we store boxes $\mathbf{x}$, together with inner and outer approximations of $\mathsf{f}(\mathbf{x})$, is the aforementioned set L_{check}.

In the second phase, we try to *verify* each box $\mathbf{x}$ to have or have not a counterpart $\mathbf{x}'$ such that:

$$\big(\mathbf{x} \cap \mathbf{x}' = \emptyset\big) \text{ and } \big(\mathsf{f}(\mathbf{x}) \cap \mathsf{f}(\mathbf{x}') \neq \emptyset\big)\,. \tag{4.4}$$

It is worth noting that we do not really have to assemble lists L_{ver} and L_{pos}: information they contain would be redundant with this already contained in L_{check}.

Furthermore, it is worth investigation, what data structure to use for storing the records in L_{check}, to find pairs of boxes that satisfy (4.4). Possibly, an interval tree would be appropriate here (cf. [22]), but it is not a panacea. A detailed discussion is out of the scope of this book, but it might be an interesting subject of further research.

4.6.4 When is the Second Phase Not Necessary?

We do not need to use such Herbrand expansions (and hence to perform the second phase) at least in the following three cases:

- formula P is non-quantified, itself—e.g., for equations systems and constraint satisfaction problems;
- formula P can be transformed to a non-quantified form symbolically, without the necessity of computing specific values—we have such a situation for computing all local minima of a function, a problem discussed in Sect. 4.8;
- the quantifier(s) in formula P ranges over other domains than X—it is a so-called quantified constraint problem (see, e.g., [4, 11, 39]); in this case we do not need any second phase, but a "nested" B&BT method in phase one—to process all feasible values of the parameter.

Obviously, in some cases we have quantifiers ranging over both: X and some other domain. Then, we might need the second phase, but some of the quantifiers will still be present; a good example is solving the min-max problem (e.g., [48]).

As already mentioned, for global optimization of a smooth function, we seek points that fulfill some first-order necessary conditions ($\nabla f(x) = 0$ for the unconstrained case or Fritz John conditions otherwise) *and* $f(x) \leq y^* + \varepsilon$. We could replace the latter by $f(x) = y^*$, but it would be pretty ill-conditioned and hard to verify.

A similar example will be encountered in the Sect. 4.8.

4.7 Necessary Conditions

Earlier, we stated that determining how many *shared quantities* $t_1, \ldots, t_k$ should be computed and what is their adequate representation is hard to be automated; probably it has to be done by a human expert. Possibly, it is related to the fact that CHECK from Algorithm 2 is not a formula in first-order logic, but in the second (or even higher) order one.

What is more, there is another very important feature of each B&BT algorithm that can hardly be provided by an algorithm: determining the necessary conditions of P, to be used in the first phase. Indeed, the predicate VERIF, as well as CHECK, goes beyond the first-order logic, certainly.

The necessary conditions of P can be classified in a few categories:

- 0th-order conditions: check the Herbrand expansion of P for current estimates of $t_1, \ldots, t_k$.
- 1st-order conditions, e.g., checking if the gradient is equal to zero for unconstrained global optimization, Fritz John conditions for constrained global optimization or analogous conditions for Pareto-optimal points [31], or game solutions, e.g., [29, 33], etc.
- 2nd-order conditions, e.g., checking the eigenvalues (or simply diagonal elements; cf., e.g., [15]) of the Hesse matrix for unconstrained global optimization; see, e.g., [15].
- Higher-order conditions, rarely used, so far.

It is worth noting that some pretty similar problems may have quite different necessary conditions. A good example is the dissimilarity between the well-known problem of global optimization and the problem of seeking ε-optimal solutions, which we already mentioned. In the latter, solutions are points satisfying the inequality $f(x) \leq y^* + \varepsilon$, thus there are no 1st-order necessary conditions, like the Fritz John ones.

Hence, for another pretty similar problem of seeking local optima, the 1st-order conditions are of high importance. This problem will be detailed in the next section.

4.8 Seeking Local Optima of a Function

The problem of finding all local minima of a function is very specific, it has interesting properties and—in contrast to, e.g., global optimization or seeking ε-optimal solutions—so far, rarely has it been considered directly (as a distinct problem from, e.g., solving a system of equations $\nabla f(x) = 0$). Notable exceptions include [8, 34, 47].

Let us formulate the problem as follows: find all elements of the set:

$$\{x \in [\underline{x}, \overline{x}] \subseteq \mathbb{R}^n \mid (\exists \varepsilon > 0)\, (\forall t \in [\underline{x}, \overline{x}] \text{ and } d(x,t) < \varepsilon)\, \big(f(x) \leq f(t)\big)\}\,. \quad (4.5)$$

The formulation is very similar to the global optimization problem, but its features are very different:

- the local optimization problem requires, as the name says, only local information; in particular, we do not need to process the objective's values, but only derivatives;
- consequently, there is no global information stored in the B&BT algorithm, no *shared quantities*;
- also, the order of processing boxes is irrelevant, while it was quite important for global optimization (cf., e.g., [15, 35, 37]);
- finally, no second phase is needed for Problem (4.5), while for global optimization it was necessary to distinguish global from local optima.

The main difference boils down to the fact that, for Problem (4.5), quantifiers in the formula can be removed symbolically, without performing their "numerical remov-

ing" (i.e., without computing any *shared quantities*) in the two phases of Algorithm 1. To be succinct: all necessary information is local, so no *shared quantities* are needed.

Remark 4.4 It is worth noticing that properties of (4.5) are more similar to properties of the problem of solving a system of equations than to global optimization—although, the formulation of (4.5) is very different from $\{x \in [\underline{x}, \overline{x}] \mid f(x) = 0\}$.

How to produce the non-quantified formulation of (4.5)? Let us consider the case of smooth functions. For unconstrained optimization, we can formulate the problem as follows:

$$\{x \in [\underline{x}, \overline{x}] \mid (\nabla f(x) = 0) \text{ and } (\text{Hesse matrix of } f(x) \text{ has no negative eigenvalues})\}\,. \tag{4.6}$$

Please note, that—also for this problem—formula (4.6) is not equivalent to (4.5), but weaker. If an eigenvalue of the second derivatives matrix is equal to zero, the point may or may not be a local minimum.

Actually, a local minimum can be singular and have arbitrarily many derivatives equal to zero, e.g., function $f(x) = x^{2 \cdot n}$, where $n \geq 2$ and $x \in [-10, 10]$. If we do not know n in advance, we cannot determine how many derivatives to compute and investigate.

A more precise formulation than (4.6) is possible; for the univariate case, it can go as follows:

$$\Big\{x \in [\underline{x}, \overline{x}] \subseteq \mathbb{R} \mid \ (f'(x) = 0) \text{ and } \big((f''(x) > 0) \text{ or } (f''(x) = 0 \text{ and } f''(\cdot) \text{ has a local minimum at } x)\big)\Big\}\,. \tag{4.7}$$

Such a formulation allows to verify local minima of an arbitrary f, provided it has no plateau, i.e., it is not constant. If f was constant on some subregions, no numerical algorithm would be able to verify it in a finite number of steps; at least not in the general case.

Also, formulation (4.7) is impractical, even for a problem with no plateau. Computing higher derivatives is difficult, both, from the theoretical (tensor algebra) and practical (lack of interval automatic differentiation libraries, with higher derivatives) point of view.

Solving Problem (4.5) for a function with a plateau seems hard, indeed. We can check if $|\mathbf{f}'(\mathbf{x})|$ is lower than some threshold value, but it does not allow to distinguish a constant function and a function changing slowly on some region (or even having a local minimum there!).

The situation is different if we state a related but distinct problem:

$$\{x \in [\underline{x}, \overline{x}] \subseteq \mathbb{R}^n \mid (\exists \varepsilon > \varepsilon_{\min})\ (\forall t \in [\underline{x}, \overline{x}] \text{ and } d(x,t) < \varepsilon)\ \big(f(x) \leq f(t)\big)\}. \tag{4.5'}$$

It is the problem of seeking "significant" local optima, i.e., the deepest optima in a "sufficiently large" subdomain. The threshold value of the subdomain radius is $\varepsilon_{\min}$. When solving such a problem, checking $|\mathbf{f}'(\mathbf{x})|$ becomes a useful tool.

By the way, please note, as quite different tools occur useful for pretty similar problems. Although, the same meta-algorithm can be applied for several problems, choosing proper tools (and heuristics to parameterize them) is pretty hard. It does not seem, this decision can be automated—only a human can choose proper tools and heuristics to make the algorithm *efficient* for a specific class of problems.

Finally, let us note that formulation (4.5') might be better for practical applications than (4.5). And the problem of enclosing all local optima of a function is of high interest as it can find several practical applications, i.a., in the game theory (so-called *potential games* [42]), NMR (nuclear magnetic resonance) spectroscopy or radio-astronomy [47].

4.9 Example Heuristics

What tools should we use to process boxes in the B&BT algorithm? Details depend on the specific problem (cf. Chaps. 5 and 6), obviously, but interval analysis provides us a variety of common tools, in particular:

- various interval Newton operators for solving equations (or inequalities) systems,
- various local consistency notions (hull-consistency, box-consistency, bound-consistency, etc.) and methods for their enforcing,
- other specific tests, e.g., checking monotonicity of a function, checking positive definiteness of a matrix, etc.

Which of these tools should be used for a specific problem (and for a specific box)? How to apply them? How should they be parameterized?

There are no general answers to these questions. Instead, we have to rely on some *heuristics*, tailored for a specific class of problems.

Many such heuristics for various problems have been developed—both, by the author and by other researchers.

Most of these tools accelerate the actual B&BT algorithm. Yet, some of them can be applied prior to it. This is the case, in particular, for initial exclusion phases, proposed, e.g., in Caprani et alii [7], Kolev [17] and a few papers of the author [23, 25, 26]. Also, prior to Algorithm 1 we can perform some symbolic preprocessing of the problem; for instance the Gröner basis theory can be applied here.

In the remainder of this section, we shall concentrate on heuristics for box subdivision.

Bisection

The most common form of box subdivision in the B&BT process is its bisection (in one of the coordinates). Some researchers (e.g., [5]; see also [15], Paragraph 5.1.2)

suggest using multisection, but according to the author's experiences (see [21]), it does not seem worthwhile.

Possibly in the future, heuristics will be developed to choose between bi- and multisection for specific classes of problems.

And which of the variables to bisect? A common idea is to bisect the longest edge of the box; it is called the maximal diameter bisection. Several other approaches have been proposed to choose the variable for bisection; see [3, 15, 45] and, in particular, [40]. Most of them are efficient on some problems, but fail not on other ones; some, like [36], have been tailored for very specific problem classes only. How to obtain more universal heuristics?

In [24] the author observed that the proper approach is to create boxes *suitable for reduction* by the rejection/reduction tools, used in the current instance of the B&BT algorithm.

For optimization problems (at least unconstrained ones) minimizing the diameter of objectives on resulting boxes is proper, usually.

But, e.g., for the problem of solving nonlinear systems, the main rejection/reduction tool is some version (or versions) of the interval Newton operator and proper heuristics should be tuned to produce boxes suitable for this procedure. Such heuristics are proposed, i.a., by the author in [24, 26].

For the problem of Pareto sets seeking the situation is yet different. The procedure to process a box is more sophisticated (the multiobjective version of the monotonicity test [28], consistency checking of the criteria [30], etc.) and the proper heuristic to choose the variable for bisection has to be adequate to these features. It is described in [32].

Remark 4.5 Some authors devise the heuristics for bisection to minimize the diameter of objectives on resulting boxes (see, in particular, [3]). In the author's opinion, this approach is in general wrong, as it does not have to lead to producing boxes suitable for further processing.

The difference is particularly important for higher problem dimensions. Please note, $\mathbb{R}^n$ for $n >> 2$ can have properties much different than these of $\mathbb{R}$ or $\mathbb{R}^2$. Distances are higher in such spaces and bisecting a single component of a box does not result in changing these distances significantly. That is why bisections should be used for separating different solution points and not for reducing the range of the function, which would require an outrageous amount of bisections.

4.10 Conclusions

In this chapter, we have considered interval branch-and-bound-type algorithms as a tool for solving a wide class of problems, described using a formula in the first-order logic. Similarities and differences between various instances of this type of algorithms have been discussed. We have tried to clear some confusion in the terminology used in the area.

We have shown, how the necessity of quantifier elimination forces splitting some versions of these algorithms into two phases. The quantifier elimination process has been linked to obtaining the Herbrand expansion of the formula.

It has been stated that, although, Algorithm 1, for solving problems of type (1.1), is pretty general, adapting it for a specific problem and tuning to be efficient is a difficult process, hard (or impossible) to be automated. Probably, at least three features have to be determined by a human: the number and nature of the *shared quantities* used by the algorithm, their adequate representation and heuristics used by the rejection/reduction tests.

Similar problems might need quite different heuristics and an expert's knowledge is necessary to choose and tune them. Artificial intelligence and self-tuning methods might be of some use, but in general, these details have to be designed by a human.

It is worth noting that B&BT algorithms are natural candidates for parallelization and—as they are usually time consuming—proper parallelization is often crucial for achieving satisfying efficiency. Parallelization of B&BT algorithms will be considered in Chap. 7.

References

1. Adler, J., Schmid, J.: Introduction to Mathematical Logic. University of Bern (2007)
2. Alefeld, G., Kreinovich, V., Mayer, J.: The shape of the solution set for systems of interval linear equations with dependent coefficients. Math. Nachr. **192**, 23–36 (1998)
3. Beelitz, T., Bischof, C.H., Lang, B.: A hybrid subdivision strategy for result-verifying nonlinear solvers. Technical Report 04/8, Bergische Universität Wuppertal (2004)
4. Benhamou, F., Goualard, F.: Universally quantified interval constraints. In: Principles and Practice of Constraint Programming–CP 2000, pp. 67–82. Springer (2000)
5. Berner, S.: New results on verified global optimization. Computing **57**(4), 323–343 (1996)
6. Buss, S.R.: On Herbrand's theorem. In: Logic and Computational Complexity, pp. 195–209. Springer (1995)
7. Caprani, O., Godthaab, B., Madsen, K.: Use of a real-valued local minimum in parallel interval global optimization. Interval Comput. **2**, 71–82 (1993)
8. Eick, C., Villaverde, K.: Robust algorithms that locate local extrema of a function of one variable from interval measurement results: a remark. Reliab. Comput. **2**(3), 213–218 (1996)
9. G-Tóth, B., Kreinovich, V.: Verified methods for computing Pareto sets: general algorithmic analysis. Int. J. Appl. Math. Comput. Sci. **19**(3), 369–380 (2009)
10. Gau, C.Y., Stadtherr, M.A.: Dynamic load balancing for parallel interval-Newton using message passing. Comput. Chem. Eng. **26**(6), 811–825 (2002)
11. Hao, F., Merlet, J.P.: Multi-criteria optimal design of parallel manipulators based on interval analysis. Mech. Mach. Theory **40**(2), 157–171 (2005)
12. Ishii, D., Goldsztejn, A., Jermann, C.: Interval-based projection method for under-constrained numerical systems. Constraints **17**(4), 432–460 (2012)
13. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
14. Jaulin, L., Walter, É.: Set inversion via interval analysis for nonlinear bounded-error estimation. Automatica **29**(4), 1053–1064 (1993)
15. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
16. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Vychislennyie Tiehnologii (Computational Technologies) **15**(1), 7–13 (2010)

17. Kolev, L.V.: Some ideas towards global optimization of improved efficiency. In: GICOLAG Workshop, Wien, Austria, pp. 4–8 (2006)
18. Kreinovich, V., Kubica, B.J.: From computing sets of optima, Pareto sets and sets of Nash equilibria to general decision-related set computations. J. Univers. Comput. Sci. **16**, 2657–2685 (2010)
19. Kreinovich, V., Lakeyev, A.V.: Linear interval equations: computing enclosures with bounded relative or absolute overestimation is NP-hard. Reliab. Comput. **2**(4), 341–350 (1996)
20. Kreinovich, V., Lakeyev, A.V., Rohn, J.: Computational Complexity and Feasibility of Data Processing and Interval Computations, vol. 10. Springer Science & Business Media (2013)
21. Kubica, B.J.: Performance inversion of interval Newton narrowing operators. Prace Naukowe Politechniki Warszawskiej. Elektronika, **169**, 111–119 (2009). KAEiOG 2009 (Konferencja Algorytmy Ewolucyjne i Optymalizacja Globalna) Proceedings
22. Kubica, B.J.: A class of problems that can be solved using interval algorithms. Computing **94**, 271–280 (2012). SCAN 2010 (14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics) Proceedings
23. Kubica, B.J.: Exclusion regions in the interval solver of underdetemined nonlinear systems. Technical Report 12-01, ICCE WUT (2012)
24. Kubica, B.J.: Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 467–476 (2012)
25. Kubica, B.J.: Excluding regions using Sobol sequences in an interval branch-and-prune method for nonlinear systems. Reliab. Comput. **19**(4), 385–397 (2014). SCAN 2012 (15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics) Proceedings
26. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. Numer. Algorithms **70**(4), 929–963 (2015). https://doi.org/10.1007/s11075-015-9980-y
27. Kubica, B.J.: Interval methods for solving various kinds of quantified nonlinear problems. In: Kosheleva, O. (ed.) Beyond Traditional Probabilistic Data Processing Techniques: Interval, Fuzzy, etc. Methods and Their Applications (2018)
28. Kubica, B.J., Woźniak, A.: Interval methods for computing the Pareto-front of a multicriterial problem. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, pp. 1382–1391 (2009)
29. Kubica, B.J., Woźniak, A.: An interval method for seeking the Nash equilibria of non-cooperative games. In: PPAM 2009 Proceedings. Lecture Notes in Computer Science, vol. 6068, pp. 446–455 (2010)
30. Kubica, B.J., Woźniak, A.: Optimization of the multi-threaded interval algorithm for the Pareto-set computation. J. Telecommun. Inf. Technol. **1**, 70–75 (2010)
31. Kubica, B.J., Woźniak, A.: Using the second-order information in Pareto-set computations of a multi-criteria problem. In: PARA 2010 Proceedings. Lecture Notes in Computer Science, vol. 7134, pp. 137–147 (2012)
32. Kubica, B.J., Woźniak, A.: Tuning the interval algorithm for seeking Pareto sets of multi-criteria problems. In: PARA 2012 Proceedings. Lecture Notes in Computer Science, vol. 7782, pp. 504–517 (2013)
33. Kubica, B.J., Woźniak, A.: Interval methods for computing strong Nash equilibria of continuous games. Decis. Mak. Manuf. Serv. **9**(1), 63–78 (2015). SING10 Proceedings
34. Lyager, E.: Finding local extremal points by using parallel interval methods. Interval Comput. **3**, 63–80 (1994)
35. Lyudvin, D.Y., Shary, S.P.: Testing implementations of PPS-methods for interval linear systems. Reliab. Comput. **19**(2), 176–196 (2013). SCAN 2012 Proceedings
36. Majumdar, S.: Application of relational interval arithmetic to computer performance analysis: a survey. Constraints **2**(2), 215–235 (1997)

37. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
38. Nesterov, V.M.: Interval and twin arithmetics. Reliab. Comput. **3**(4), 369–380 (1997)
39. Ratschan, S.: Continuous first-order constraint satisfaction. Lecture Notes in Computer Science, vol. 2385, pp. 181–195 (2002)
40. Ratz, D., Csendes, T.: On the selection of subdivision directions in interval branch-and-bound methods for global optimization. J. Glob. Optim. **7**, 183–207 (1995)
41. Rohn, J., Kreinovich, V.: Computing exact componentwise bounds on solutions of linear systems with interval data is NP-hard. SIAM J. Matrix Anal. Appl. **16**(2), 415–420 (1995)
42. Rosenthal, R.W.: A class of games possessing pure-strategy Nash equilibria. Int. J. Game Theory **2**(1), 65–67 (1973)
43. Sharaya, I.A.: On maximal inner estimation of the solution sets of linear systems with interval parameters. Reliab. Comput. **7**(5), 409–424 (2001)
44. Shary, S.P.: Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of Kaucher arithmetic. Reliab. Comput. **2**(1), 3–33 (1996)
45. Shary, S.P.: Finite-dimensional Interval Analysis. Institute of Computational Technologies, SB RAS, Novosibirsk (2013)
46. Traylor, B., Kreinovich, V.: A bright side of NP-hardness of interval computations: interval heuristics applied to NP-problems. Reliab. Comput. **1**(3), 343–359 (1995)
47. Villaverde, K., Kreinovich, V.: A linear-time algorithm that locates local extrema of a function of one variable from interval measurement results. Interval Comput. **4**, 176–194 (1993)
48. Zuhe, S., Neumaier, A., Eiermann, M.: Solving minimax problems by interval methods. BIT Numer. Math. **30**(4), 742–751 (1990)

Chapter 5
Solving Equations and Inequalities Systems Using Interval B&Bt Methods

Let us consider solving the following problem:

$$\text{Find } all\ x \in X \text{ such that } c_i(x) \text{ are fulfilled for } i = 1, \ldots, m\ . \tag{5.1}$$

Such a problem is called a Constraint Satisfaction Problem (or CSP for short) and constraints can be either inequalities "$g_i(x) \leq 0$" or equations "$f_i(x) = 0$".

Let us start our considerations with CSPs having inequalities only.

5.1 Constraint Satisfaction Problems

So, we are trying to compute the set:

$$S = \{x \in X \mid g_i(x) \leq 0, i = 1, \ldots, m\}.$$

We can also write $S = \{x \in X \mid g(x) \leq 0\}$, where $g = (g_1, \ldots, g_m)$.

What can we compute using interval methods? Two lists of boxes, as for any B&BT algorithm.

In case of a system of inequalities, the interior of the solution set S is nonempty and the *verified* solutions are boxes contained in this interior, i.e., boxes that contain solutions only. *Possible* boxes will lie on the boundaries and contain some points both from S and its complement $\mathbb{R}^n - S$. Typically there will be several possible boxes, unless S is a box itself (which would be highly unlikely).

The branch-and-prune algorithm for a CSP can be formulated as follows:

The "rejection/reduction tests", mentioned in the algorithm are going to be described later in this chapter; see also the book [24] or several papers, e.g., [3, 4, 7, 8, 18, 19, 40, 41].

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_5

Remark 5.1 It is worth noting that there exist several algorithms similar to Algorithm 4. They differ in some minor details or are devoted to problems slightly different than (5.1). A good example is SIVIA—Set Inversion Via Interval Analysis (see [24, 25]; cf. [38]). This algorithm is devoted to seek an inversion of the set Y, i.e., the set $\{x \in X \mid g(x) \subseteq Y \leq 0\}$, where Y is usually an interval, but can be a more sophisticated set, also. SIVIA is pretty similar to Algorithm 4; necessary modifications are left to the reader as a simple exercise.

Both SIVIA and the B&P algorithm for CSPs that consist of inequalities only, are relatively simple. We can verify the solutions directly by enclosing the range of g functions on obtained boxes; no more complicated tools are needed for the verification. Consequently, all more sophisticated tests applied to a box are optional; they can accelerate discarding infeasible boxes, though.

These tests (interval Newton operators, consistency operators, etc.) are very similar to the ones used for nonlinear equations systems; as they are optional for inequalities systems, we shall not discuss them until later in this chapter (Sect. 5.3 and next).

5.2 Solving Systems of Nonlinear Equations

Now, let us consider the problem of solving equations or systems of equations. So, we are trying to compute the set:

Algorithm 4 Interval branch-and-prune algorithm for a system of inequelities

Require: $\mathbf{x}^{(0)}, \mathsf{g}$
1: $\{\mathbf{x}^{(0)}$ is the initial box, $\mathsf{g}(\cdot)$ is the interval extension of the function $g\colon \mathbb{R}^n \to \mathbb{R}^m\}$
2: $\{L_{ver}$ – verified solution boxes, L_{pos} – possible solution boxes$\}$
3: $L_{ver} = L_{pos} = \emptyset$
4: $\mathbf{x} = \mathbf{x}^{(0)}$
5: **loop**
6: compute $\mathbf{y} = \mathsf{g}(\mathbf{x})$
7: optionally, process the box $\mathbf{x}$, using additional rejection/reduction tests
8: **if** $(\underline{y} > 0)$ **then**
9: discard $\mathbf{x}$
10: **else if** $(\overline{y} \leq 0])$ **then**
11: push $(L_{ver}, \mathbf{x})$
12: **else if** $(\text{wid}\,\mathbf{x} < \varepsilon)$ **then**
13: push $(L_{pos}, \mathbf{x})$ {The box $\mathbf{x}$ is too small for bisection}
14: **if** ($\mathbf{x}$ was discarded **or** $\mathbf{x}$ was stored) **then**
15: **if** $(L == \emptyset)$ **then**
16: **return** L_{ver}, L_{pos} {All boxes have been considered}
17: $\mathbf{x} = \text{pop}\,(L)$
18: **else**
19: bisect $(\mathbf{x})$, obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
20: $\mathbf{x} = \mathbf{x}^{(1)}$
21: push $(L, \mathbf{x}^{(2)})$

$$S = \{x \in X \mid f_i(x) = 0, i = 1, \ldots, m\},$$

which we can also denote as $S = \{x \in X \mid f(x) = 0\}$, where $f = (f_1, \ldots, f_m)$.

Let us present the B&P algorithm for equations systems, based on the HIBA_USNE solver [2]. It can be expressed by the pseudocode presented in Algorithm 5.

Algorithm 5 Interval branch-and-prune algorithm

Require: L, f, ε
1: {L – the list of initial boxes, often containing a single box $\mathbf{x}^{(0)}$}
2: {L_{ver} – verified solution boxes, L_{pos} – possible solution boxes}
3: $L_{ver} = L_{pos} = \emptyset$
4: $\mathbf{x}$ = pop (L)
5: **loop**
6: process the box $\mathbf{x}$, using the rejection/reduction tests
7: **if** ($\mathbf{x}$ does not contain solutions) **then**
8: discard $\mathbf{x}$
9: **else if** ($\mathbf{x}$ is verified to contain a segment of the solution manifold) **then**
10: push (L_{ver}, $\mathbf{x}$)
11: **else if** (the tests resulted in two subboxes of $\mathbf{x}$: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) **then**
12: $\mathbf{x} = \mathbf{x}^{(1)}$
13: push (L, $\mathbf{x}^{(2)}$)
14: **cycle loop**
15: **else if** (wid $\mathbf{x} < \varepsilon$) **then**
16: push (L_{pos}, $\mathbf{x}$) {The box $\mathbf{x}$ is too small for bisection}
17: **if** ($\mathbf{x}$ was discarded **or** $\mathbf{x}$ was stored) **then**
18: **if** ($L == \emptyset$) **then**
19: **return** L_{ver}, L_{pos} {All boxes have been considered}
20: $\mathbf{x}$ = pop (L)
21: **else**
22: bisect ($\mathbf{x}$), obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
23: $\mathbf{x} = \mathbf{x}^{(1)}$
24: push (L, $\mathbf{x}^{(2)}$)

The "rejection/reduction tests", mentioned in line 6 of the algorithm, include, in particular:

- switching between the componentwise Newton operator (for larger boxes) and Gauss-Seidel with inverse-midpoint preconditioner, for smaller ones,
- box-consistency enforcing [35],
- bound-consistency enforcing [36],
- hull-consistency and 3B-consistency enforcing [37],
- an additional second-order approximation procedure [34],
- sophisticated heuristics to choose the bisected component [32, 35],
- an initial exclusion phase of the algorithm (deleting some regions, not containing solutions)—based on Sobol sequences [33, 35].

The above list is not exhaustive.

What is the purpose of processing the box using these tests? We can say, there are two main purposes:

- discarding boxes (or subregions of boxes) that do not contain solutions,
- proving the existence of solutions in boxes that do contain them.

The first goal is crucial for efficiency of the solver. As Gutowski states in his book [20], what we are trying to do is "picking holes" in the solution set; in the author's opinion, it is a very good description of B&BT methods.

The second goal is more subtle. In some applications, it might be more important to us to localize solutions than to verify them. Yet, interval methods allow us to *prove* that some boxes contain solution points.

For equations systems, proving it is more difficult than for inequalities systems. It does not suffice to find a box $\mathbf{x}$ such that $\mathsf{f}(\mathbf{x}) \subseteq [-\varepsilon, +\varepsilon]$ for some small ε; as the computed value is overestimated, it might be the case that all values of actual $f(x)$, $x \in \mathbf{x}$ are positive (or all of them are negative) and zero is not reached anywhere.

Nevertheless, there are procedures allowing to verify at least some of the solutions. The most important one (but not the only one!) of them is the interval Newton operator.

5.3 Interval Newton Operators

The interval Newton operator (see, e.g., [21, 27]) is one of the most celebrated achievements of the interval analysis. It can be understood as a generalization of the traditional (point) Newton operator (Fig. 5.1).

In the univariate case, the operator can be expressed by the following formula:

$$N(\mathbf{x}, \mathsf{f}, \check{x}) = \check{x} - \frac{\mathsf{f}(\check{x})}{\mathsf{f}'(\mathbf{x})} .$$

Usually $\check{x} = \operatorname{mid} \mathbf{x}$ is used; in general, another $\check{x} \in \mathbf{x}$ could be applied, also.

In the multidimensional case, instead of division by an interval, we have to solve a linear equations system with an interval coefficient matrix:

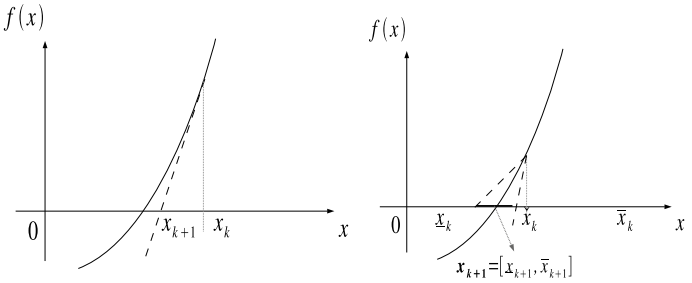

Fig. 5.1 Illustration of the Newton operators: pointwise (left) and interval (right) ones

$$\mathbf{A} \cdot (\mathbf{x} - \check{x}) = -\mathsf{f}(\check{x}) \,, \tag{5.2}$$

where $\mathbf{A}$ is the so-called Lipschitz matrix of $f(\cdot)$, i.e., the interval matrix satisfying the condition:

$$(\forall x \in \mathbf{x})\,(\forall y \in \mathbf{x})\,(\exists A \in \mathbf{A})\,\big(f(x) - f(y) = A \cdot (x - y)\big) \,.$$

The interval linear system (5.2) can be solved in a few ways. Among other versions of the interval Newton operator, it is worth to note the Krawczyk operator (see, e.g., [27]) and the componentwise Newton operator [22]. Several versions and details have been discussed in [31]; see also [29]. Yet, nowadays, the interval Gauss-Seidel method is the most commonly used. Before we present it, let us say a few words about preconditioning the Eq. (5.2).

Preconditioners

Multiplying both sides of an equations system by a nonsingular matrix does not change the roots of this system.

So the equations system:

$$Y \cdot \mathbf{A} \cdot (\mathbf{x} - \check{x}) = -Y \cdot \mathsf{f}(\check{x}) \,,$$

has the same solutions as (5.2), for each nonsingular matrix Y (with real-valued components).

Such *preconditioning* is often applied for traditional floating-point linear systems, to improve the numerical conditioning. For interval systems, preconditioning is even more important, as it can dramatically reduce (or enlarge) the diameter of resulting enclosures for the solution (cf. the description of interval calculus in Chap. 2). Multiplying the sides of the equation by a suitable *preconditioning matrix* Y, may lead to a significant improvement in the obtained results.

In the context of the Newton method, a quite commonly used preconditioner is the *inverse midpoint preconditioner*:

$$Y = (\operatorname{mid} \mathbf{A})^{-1} \,.$$

The operation "mid" of interval matrices and vectors is understood componentwise, i.e. it produces real matrices (vectors) of midpoints.

It is worth noting that the inverse midpoint preconditioner is not the only one in use. In particular, Kearfott and other authors recommend using so-called LP-preconditioners [27]. They are obtained by solving a proper linear programming problem. The author does not use this approach; reasons will be given in Chap. 7.

The Interval Gauss-Seidel Step

Formulae used in the preconditioned GS step are similar to the ones of the traditional GS step. Let us present the pseudocode for the case of well-determined systems; in

the underdetermined case, we have to choose a submatrix of **A**. Details can be found in [31]; cf. also [39].

But for the well-determined case, we get Algorithm 6.

Algorithm 6 Interval Gauss-Seidel step

Require: $\mathbf{x}^{\text{old}}, \check{x}$
1: $\mathbf{x} = \mathbf{x}^{\text{old}}$
2: **for** $(i = 1, \ldots, n)$ **do**
3: $Y_{i:}$, the i-th row of the preconditioning matrix Y
4: $\mathbf{x}_i^{\text{new}} = \check{x}_i - \left(Y_{i:} \cdot \mathsf{f}(\check{x}) + \sum_{j=1}^{i-1} Y_{i:} \cdot \mathbf{A}_{j:} \cdot (\mathbf{x}_j^{\text{new}} - \check{x}_j) + \sum_{j=i+1}^{N} Y_{i:} \cdot \mathbf{A}_{j:} \cdot (\mathbf{x}_j - \check{x}_j)\right) / (Y_{i:} \cdot A_{i:})$
5: **if** $\left(\mathbf{x} \cap \mathbf{x}^{\text{new}} == \emptyset\right)$ **then**
6: **return** $\emptyset$ {there are no solutions in $\mathbf{x}^{\text{old}}$}
7: replace $\mathbf{x}$ by $\left(\mathbf{x} \cap \mathbf{x}^{\text{new}}\right)$
8: **return** $\mathbf{x}$

Algorithm 6 returns the box in which all roots of the Eq. (5.2) must lie; it returns an empty set, if it can be verified that there are no roots in $\mathbf{x}^{\text{old}}$.

Properties of the Interval Newton Operator

Why is the interval Newton operator such a useful tool? Because its properties are much stronger than the features of a point Newton operator.

The interval Newton step allows not only to discard or reduce boxes by proving the non-existence of roots; it allows also to prove the uniqueness of solutions in some boxes.

The following propositions state these properties (cf., e.g., [21, 27]).

Proposition 5.1 *Suppose we are trying to find the roots of the equation (equations system)* $f(x) = 0$ *in the box* $\mathbf{x}$*; we use the interval extension* $\mathsf{f}(\cdot)$ *of the function* $h(\cdot)$*. Let computation of the interval Newton operator lead to the result* $N(\mathbf{x}, \mathsf{f}, \check{x}) \cap \mathbf{x} = \emptyset$*, where* $\check{x}$ *is any point from the box* $\mathbf{x}$*.*

Then $f(x)$ *has* no roots *in the box* $\mathbf{x}$*.*

Proposition 5.2 *Suppose we are trying to find the roots of the equation (equations system)* $f(x) = 0$ *in the box* $\mathbf{x}$*; we use the interval extension* $\mathsf{f}(\cdot)$ *of the function* $f(\cdot)$*.*

Each root of $f(x)$ *belonging to the box* $\mathbf{x}$ *(if any), belongs to the set* $\left(N(\mathbf{x}, \mathsf{f}, \check{x}) \cap \mathbf{x}\right)$*.*

Proposition 5.3 *Suppose we are trying to find the roots of the equation (equations system)* $f(x) = 0$ *in the box* $\mathbf{x}$*; we use the interval extension* $\mathsf{f}(\cdot)$ *of the function* $f(\cdot)$*. Let computation of the interval Newton operator lead to the result* $N(\mathbf{x}, \mathsf{f}, \check{x}) \subset \operatorname{int} \mathbf{x}$*, where* $\check{x}$ *is any point from the box* $\mathbf{x}$*.*

Then there is a single unique root *of* $f(x)$ *in the box* $\mathbf{x}$*.*

Remark 5.2 We can say that the interval Newton operator is that important as it can achieve both goals of tools processing boxes in B&P algorithms: it can discard non-solutions and verify actual solutions. Most other tools allow only doing one of these operations; e.g., consistency methods prune boxes, but do not verify the existence, while several verification tests do not allow pruning or discarding boxes.

5.4 Other Verification Tests

As said above, the Newton operator is the most famous and the most celebrated tool for boxes verification. Indeed, as stated in Remark 5.2, its features are very attractive, but it is certainly not the only verification tool that can be used. There are several others—some of them are very simple and some are very sophisticated and based on advanced mathematical theories, most notably on algebraic topology.

5.4.1 Miranda Test

This test, presented, e.g., in [27], is one of the simplest. Consider a continuous function $f : X \to \mathbb{R}$. Assume, we have found two points $a, b \in X$ such that $f(a) > 0$ and $f(b) < 0$. It is well known (from the Bolzano's intermediate value theorem), that any curve connecting a and b and lying inside X, on which f is continuous, will contain a point x such that $f(x) = 0$.

The Miranda's theorem generalizes this result to a function $f : \mathbb{R}^n \to \mathbb{R}^n$.

Another similar test is also presented in [13].

5.4.2 Using Quadratic Approximation

This test has been proposed by the author in [34]. As the Newton test is based on linear approximation of f, a quadratic approximation can be used, also. Yet, unlike in the Newton case, we do not approximate the whole $f = (f_1, \ldots, f_m)$, but only one of its components f_i.

As we obtain a representation in the form of a quadratic function, we can solve the resulting quadratic equation, to find its zeros. Obviously, this is a quadratic equation with interval-valued coefficients. In Chapter 8 of the book of Hansen and Walster [21], a procedure to enclose zeros of such an equation is presented.

Experiments in [34] show that this test can dramatically improve the performance on some problems, but is rather inefficient on other ones. As it is expensive to use (it requires computing second derivatives), it should be used with caution. An adequate heuristic is proposed in [34]; cf. also Sect. 5.6.

Remark 5.3 The test using the quadratic approximation is one of the few that share the feature of the interval Newton operator, described in Remark 5.2: it can both—discard and narrow boxes not containing solutions and verify boxes that contain the actual solutions.

5.4.3 Borsuk Test

This tool, proposed in [12], is based on one of the theorems of Karol Borsuk. The theorem states (slightly simplifying) that the function $f(\cdot)$ must have a zero on the box $\mathbf{x}$, if:

$$f(\operatorname{mid}\mathbf{x} + r) \neq \lambda \cdot f(\operatorname{mid}\mathbf{x} - r), \ (\forall\lambda > 0) \text{ and } (\operatorname{mid}\mathbf{x} + r \in \partial\mathbf{x}) \ . \tag{5.3}$$

For each pair of faces $\mathbf{x}_{i+}$, $\mathbf{x}_{i-}$ of $\mathbf{x}$, we have to compute the intersection of interval expressions:

$$]0, +\infty[\cap\Big(\cap_{j=1}^{m} \frac{\mathbf{f}_j(\mathbf{x}_{i+})}{\mathbf{f}_j(\mathbf{x}_{i-})}\Big) \neq \emptyset \ . \tag{5.4}$$

If the intersection is empty for at least m pairs of faces, then there is no λ for which the disequality (5.3) becomes an equality; hence, according to the theorem, f has a zero in $\mathbf{x}$.

Remark 5.4 In its original formulation [12], the test is used for well-determined problems, i.e., such where $n = m$. Hence, the intersection (5.4) must be nonempty for all $i = 1, \ldots, n$. In the underdetermined case, it suffices that the intersection is nonempty for m arbitrary values of i.

Remark 5.5 Instead of whole faces $\mathbf{x}_{i+}$ and $\mathbf{x}_{i-}$, we could use their sub-faces—or, more precisely: pairs of sub-faces, symmetric with respect to the center of the box. Such subdivision of faces allows more precise approximation of f's values, but at additional computational cost. Details can be found, e.g., in [11] or the Master's thesis [5]; the author's HIBA_USNE solver does not use this approach.

5.4.4 Computing Topological Degree

Topological degree is one of the most general tools to verify existence of equations' zeros.

Let us define the degree: $deg(f, y, B)$, where $f\colon \mathbb{R}^n \to \mathbb{R}^n$ is a function, y a value from its codomain (in all our considerations we shall take $y = 0$) and B is a closed connected n-dimensional subset of the f's domain.

Definition of the degree is complicated (cf. [10, 26, 27]), but when y is not a singular value of f and $y \notin \text{range}(f, \partial B)$, we have:

$$deg(f, y, B) = \sum_{x \in f^{-1}(y)} \operatorname{sgn} \det f'(x) \,, \tag{5.5}$$

where $f'(x)$ represents the Jacobi matrix of the system $f(x) = 0$.

A few algorithms have been developed to compute the topological degree: [9, 10, 26]. Some of them (in particular, [26]) base on computing the formula (5.5), other ones adopt different approaches. As important property of the degree is its composability: if we have two sets B_1, B_2 such that $B_1 \cap B_2 = \emptyset$ or even int $B_1 \cap$ int $B_2 = \emptyset$, then:

$$deg(f, y, B_1 \cup B_2) = deg(f, y, B_1) + deg(f, y, B_2) \,. \tag{5.6}$$

The algorithm of Franek and Ratschan [10] is based on this feature. It subdivides the set under consideration to obtain sets for which the degrees can be computed more easily.

Why would we want to compute this degree, anyway? Because of its most important property:

$$\text{If } deg(f, y, B) \neq 0, \text{ then } \exists x_0 \in B \; f(x_0) = 0 \,. \tag{5.7}$$

5.4.5 *Obstruction Theory Test*

In a series of papers (see, e.g., [9, 10] and the references therein), Franek et alii propose a fascinating family of methods targeted specifically at underdetermined systems.

Assume, we have $f : \mathbb{R}^n \to \mathbb{R}^m$, where $m < n$ and we want to verify that a box $\mathbf{x}$ contains a solution point (a segment of the solution manifold). Assume the boundary region of $\mathbf{x}$ does not contain zeros: $\forall x \in \partial\mathbf{x} \; f(x) \neq 0$.

The question is, if f can be extended from $\partial\mathbf{x}$ to the whole $\mathbf{x}$ without containing a zero.

Let us formulate it differently. As the boundary region of $\mathbf{x}$ does not contain zeros, we can consider the image of f on the boundary as a space homeomorphic to the subset of the $(m-1)$-dimensional sphere S^{m-1}. The boundary ∂x of $\mathbf{x} \subset \mathbb{R}^n$ itself, is obviously homeomorphic to S^{n-1}.

So, the problem boils down to checking the *extendability* of some function $f : S^{n-1} \to S^{m-1}$ from S^{n-1} to the whole disk D^n. Abusing the notation, we do not distinguish between the original f and $f : S^{n-1} \to S^{m-1}$. This should not lead to any confusion.

To be succinct, the methods of Franek et alii try to approximate the boundary ∂x as a cell complex or a simplicial set and they construct a Postnikov complex, build of Eilenberg-MacLane spaces. Basing on this representation, we can check possible extendability of a function for subsequent skeletons of the complex.

This test seems a pretty general tool, suitable for underdetermined problems as well as well-determined ones (in the latter case it is equivalent to using the topological degree). Unfortunately, it is not only based on complicated mathematical notions, but also it seems extremely cumbersome to implement and usually requiring high computational effort. Eilenberg-MacLane spaces have often infinite dimensionality and thus they can only be represented implicitly. And they are only a building block of the algorithm!

Also, please note that existing software such as GUDHI [1] is of little help when implementing this test and some of the useful algorithms might even occur to be unimplementable, like, e.g., the Brown's algorithm [6].

The things would get improved significantly, if we could operate on cubical complexes directly and not convert them to simplicial sets. Unfortunately, it is not obvious if required operations can be performed for cubical complexes. The topic can be subject of an interesting research in the future.

5.5 Consistency Enforcing

Enforcing some kind of consistency on a box is one of the most common pruning operations, allowing to reduce or discard boxes in the B&P process.

The notion of "consistency" has been used in Constraint Logic Programming: a constraint is *arc-consistent*, if for each values in all variables' domains, there are values of other variables such that the constraint is fulfilled.

For discrete domains, we can directly use arc-consistency. For continuous domains, this kind of consistency would be too restrictive, so there are various relaxations of this notion [4]. Some of them take into account only the numerical accuracy, but some are even weaker, but easier to compute, in particular: hull-consistency and box-consistency.

5.5.1 Hull-Consistency

Hull-consistency (also known under the name of 2B-consistency) has been used in several interval programs over the years; see, e.g., [3, 4]. It can be defined as follows.

Definition 5.1 A box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ is hull-consistent with respect to a constraint $c(x_1, \ldots, x_n)$, iff:

$$\forall i \ \mathbf{x}_i = \square\{s \in \mathbf{x}_i \mid \exists x_1 \in \mathbf{x}_1, \cdots \exists x_{i-1} \in \mathbf{x}_{i-1}, \exists x_{i+1} \in \mathbf{x}_{i+1} \cdots \exists x_n \in \mathbf{x}_n \\ c(x_1, \ldots, x_{i-1}, s, x_{i+1}, \ldots, x_n)\} \ .$$

Following [28], the symbol "$\square$" denotes the interval hull.

Other words, $\mathbf{x}$ is hull-consistent iff for each i we can find two points x^a and x^b, satisfying the property c, for which $x_i^a = \underline{x}_i$ and $x_i^b = \overline{x}_i$.

Remark 5.6 Slightly simplifying, hull-consistency is a relaxation of arc-consistency requiring only existence of solutions on the endpoints of each variable's domain, but not necessarily for all interior points.

Now, let us describe, how to check if a box is hull-consistent and how to enforce hull-consistency on a box.

5.5.1.1 Algorithms for Enforcing Hull-consistency

For simple constraints, checking and/or enforcing hull-consistency is relatively simple.

As a simple example, let us consider an equation $x_1 + x_2 - 3 = 0$. By obvious symbolic transformations, we obtain formulae for both variables that can be used to obtain their consistent domains:

$$\begin{aligned} \mathbf{x}_1 &= 3 - \mathbf{x}_2 \text{ and} \\ \mathbf{x}_2 &= 3 - \mathbf{x}_1. \end{aligned}$$

Using the above consistency operators, we can simply check consistency for any box or compute its sub-box containing all consistent values. For instance, a box $[-4, 2] \times [-2, 4]$ is not hull-consistent, but it can be reduced to the hull consistent one, by applying:

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{x}_1 \cap (3 - \mathbf{x}_2) = [-4, 2] \cap [-1, 5] = [-1, 2], \\ \mathbf{x}_2 &= \mathbf{x}_2 \cap (3 - \mathbf{x}_1) = [-2, 4] \cap [1, 7] = [1, 4]. \end{aligned}$$

This box is hull-consistent indeed, as points $(-1, 4)$ and $(1, 2)$ are solutions of the initial constraint $x_1 + x_2 - 3 = 0$.

However, for a more sophisticated constraint, obtaining a consistent box is not as straightforward. Let us consider the constraint:

$$x_1^3 + 2 \cdot x_1^2 - \sin(x_2) = 0. \tag{5.8}$$

Again, by relatively simple symbolic transformations we can extract x_2 from Eq. (5.8), but not x_1. The solution is to decompose such an equation into primitive ones, by adding additional variables and apply hull-consistency to such a decomposed system. For the constraint (5.8), we could obtain:

$$
\begin{aligned}
t_1 - x_1^3 &= 0,\\
t_2 - x_1^2 &= 0,\\
t_3 - 2 \cdot t_2 &= 0,\\
t_4 - t_1 - t_3 &= 0,\\
t_5 - \sin(x_2) &= 0,\\
t_4 - t_5 &= 0.
\end{aligned}
$$

The algorithm HC4 [3] (cf. also [19]) performs such a decomposition, creating a tree of the initial constraint, where a variable corresponds to each node (Fig. 5.2).

5.5.2 *Box-Consistency*

Definition 5.2 A box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ is box-consistent, iff for each i:

$$
\begin{aligned}
&\mathsf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, [\underline{x}_i, \underline{x}_i^+], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n) \ni 0 \text{ and}\\
&\mathsf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, [\overline{x}_i^-, \overline{x}_i], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n) \ni 0\,.
\end{aligned}
$$

Let us consider a simple example: the system of equations

$$C = \{x_1 + x_2 = 0,\ x_1 - x_2 = 0\}\,.$$

The box $[-1, 1]^2$ contains a single solution only: $(0, 0)$. Nevertheless, it is box-consistent with respect to the system C.

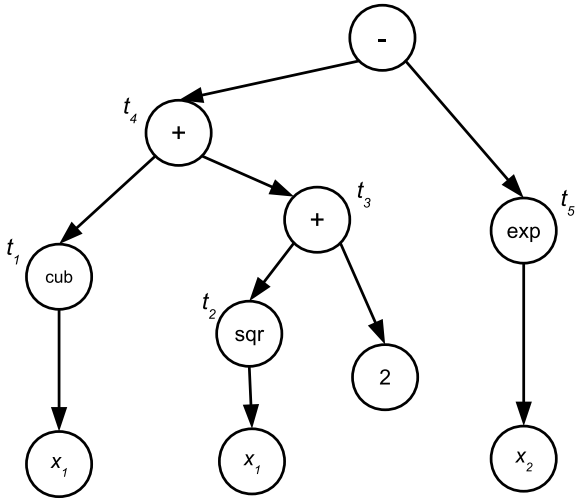

Fig. 5.2 Expression tree of constraint (5.8)

5.5.2.1 Algorithms for Enforcing Box-Consistency

A possible procedure enforcing box-consistency is described by Algorithm 7. There can be also other procedures, differing in some significant details (cf., e.g., [3, 4, 8, 15]).

Algorithm 7 Procedure bc_enforce

Require: $\mathbf{x}$, f, L_{pairs}, ε, ε_{equal}
1: **repeat**
2: store $\mathbf{x}^{old} = \mathbf{x}$
3: modified = **false**
4: **for all** $(j, i) \in L_{pairs}$ **do**
5: **if** (bc3revise($\mathbf{x}$, f_j, i, $\mathbf{x}_i^{old}$, ε, ε_{equal}, modified) results in "no solutions") **then**
6: **return** "no solutions"
7: **until** (**not** modified)
8: **return** $\mathbf{x}$

Procedure "bc3revise" performs the reduction for a specific variable with respect to a specific equation. It is described by Algorithm 8.

Algorithm 8 Procedure bc3revise

Require: $\mathbf{x}$, f, i, $\mathbf{x}_i^{old}$, ε, ε_{equal}, modified
1: **if** (left_narrow ($\mathbf{x}$, f, i, ε, ε_{equal}) results in "no solutions") **then**
2: **return** "no solutions"
3: **if** (right_narrow ($\mathbf{x}$, f, i, ε, ε_{equal}) results in "no solutions") **then**
4: **return** "no solutions"
5: **if** (dist($\mathbf{x}_i$, $\mathbf{x}_i^{old}$) $>= \varepsilon_{equal}$) **then**
6: modified = **true**
7: **return** $\mathbf{x}$

Procedure "left_narrow" is described by Algorithm 9 ("right_narrow" is analogous). The code can be found (with comments and a discussion) in [35, 36], but we give it here for the sake of completeness.

5.5.3 Higher-Order Consistencies

The most important drawback of both hull- and box-consistency is that it considers only a single constraint. There are however higher-order-consistency notions that do not have this limitation.

Most authors (e.g., [7, 40]) use the following notions of 3B-consistency and bound-consistency (or "box(2)-consistency"):

Algorithm 9 Procedure left_narrow

Require: $\mathbf{x}, \mathsf{f}, i, \varepsilon, \varepsilon_{equal}$
1: $\mathbf{x}_{left} = [\underline{x}_i, \underline{x}_i^+]$
2: **if** $(0 \in \mathsf{f}(\mathbf{x}_{left}))$ **then**
3: **return** $\underline{x}_{left}$, "found a pseudo-solution"
4: compute the interval extension of $\mathbf{g}_i = \frac{\partial f}{\partial x_i}(\mathbf{x})$
{Using the automatic differentiation arithmetic makes us compute the whole interval gradient $\mathbf{g}$, but only one component is going to be used}
5: update $\underline{x} = \overline{x}_{left}$
6: compute $\mathbf{x}_{new} = \mathbf{x}_{left} - \frac{\mathsf{f}(\mathbf{x}_{left})}{\mathbf{g}_i}$ {Perform the interval Newton step using ordinary or extended interval arithmetic, depending on whether $0 \in \mathbf{g}_i$ or not}
7: **if** $(\mathbf{x}_i \cap \mathbf{x}_{new} = \emptyset)$ **then**
8: **return** "no solution"
9: **if** $(\mathrm{dist}(\mathbf{x}_i, \mathbf{x}_{new}) < \varepsilon_{equal})$ **then**
10: update $\mathbf{x}_i = \mathbf{x}_i \cap \mathbf{x}_{new}$
11: **return** $\underline{x}_i$, "found a pseudo-solution"
12: update $\mathbf{x}_i = \mathbf{x}_i \cap \mathbf{x}_{new}$
13: **if** $(\mathrm{wid}\,\mathbf{x}_i \leq \varepsilon)$ **then**
14: **return** $\underline{x}_i$, "found a pseudo-solution" {The component $\mathbf{x}_i$ too narrow for bisection}
15: bisect $\mathbf{x}_i$, obtaining $\mathbf{x}_i^{(1)}$ and $\mathbf{x}_i^{(2)}$
16: **if** (left_narrow $(\mathbf{x}^{(1)}, \mathbf{f}, i, \varepsilon)$ results in $(x^*$, "found a pseudo-solution")) **then**
17: **return** x^*, "found a pseudo-solution"
18: **if** (left_narrow $(\mathbf{x}^{(2)}, \mathbf{f}, i, \varepsilon)$ results in $(x^*$, "found a pseudo-solution")) **then**
19: **return** x^*, "found a pseudo-solution"
20: **return** "no solution"

Definition 5.3 A box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ is 3B-consistent, iff all its facets, lower: $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, [\underline{x}_i, \underline{x}_i^+], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^T$ and upper ones: $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, [\overline{x_i}^-, \overline{x}_i], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^T$, contain a non-empty 2B-consistent subbox.

Definition 5.4 A box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ is bound-consistent, iff all its facets, lower: $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, [\underline{x}_i, \underline{x}_i^+], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^T$ and upper ones: $(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, [\overline{x_i}^-, \overline{x}_i], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^T$, contain a non-empty box-consistent subbox.

Yet higher order kB consistency is also used; to the best knowledge of the author, an analog for box-consistency has never been used (or named).

Let us set these notions in order:

- the simplest consistencies we shall call by their traditional names: box-consistency, hull-consistency, interval-consistency (cf. [4]), etc.,
- higher order consistencies will be called kB-consistency(c), where c denotes the simplest consistency used.

So, we have:

- HC: hull-consistency,
- 3B-consistency(HC), traditionally called just 3B-consistency,
- kB-consistency(HC), traditionally called just kB-consistency,
- BC: box-consistency,

- 3B-consistency(BC), traditionally called bound-consistency,
- kB-consistency(BC), which has not have a name, formerly,
- IC: interval-consistency,
- 3B-consistency(IC),
- …

The procedure enforcing the higher-order consistency, has been proposed by the author. Up to now, it has been implemented for the case of 3B-consistency(BC) (i.e., bound-consistency) [36]. We present it as Algorithm 10.

Algorithm 10 Higher-order-consistency enforcing procedure

Require: $\mathbf{x}$, $\mathbf{f}$, ε, ε_{equal}, λ_{min}, consistency_proc
{$\mathbf{f} = (\mathbf{f}_1, \ldots, \mathbf{f}_m)$, consistency_proc is the procedure enforcing lower-order consistency; this procedure might require also some other arguments}
1: let $\lambda = \lambda_0$
2: **repeat**
3: modified = **false**
4: **for** $(i = 1, \ldots, n)$ **do**
5: store $\mathbf{x}^{cur} = \mathbf{x}$
6: store $\mathbf{x}^{old} = \mathbf{x}_i^{cur}$
7: $c_1 = (1 - \lambda) \cdot \underline{x}_i^{cur} + \lambda \cdot \overline{x}_i^{cur}$
8: $c_2 = \lambda \cdot \underline{x}_i^{cur} + (1 - \lambda) \cdot \overline{x}_i^{cur}$
9: change $\mathbf{x}_i^{cur} = [c_2, \overline{x}_i^{cur}]$
10: apply consistency_proc to $\mathbf{x}^{cur}$
11: **if** (it resulted in "no solutions") **then**
12: change $\mathbf{x}_i = [\underline{x}_i, c_2]$
13: **else**
14: change $\mathbf{x}_i = [\underline{x}_i, \overline{x}_i^{cur}]$
15: store $\mathbf{x}^{cur} = \mathbf{x}$
16: change $\mathbf{x}_i^{cur} = [\underline{x}_i^{cur}, c_1]$
17: apply consistency_proc to $\mathbf{x}^{cur}$
18: **if** (it resulted in "no solutions") **then**
19: change $\mathbf{x}_i = [c_2, \overline{x}_i]$
20: **else**
21: change $\mathbf{x}_i = [\underline{x}_i^{cur}, \overline{x}_i]$
22: **if** $(\text{wid}\,\mathbf{x}_i + \varepsilon_{equal} \leq \mathbf{x}_i^{old})$ **then**
23: modified = **true**
24: **if** (**not** modified) **then**
25: change $\lambda = \lambda/2$
26: **until** $(\lambda \leq \lambda_{min})$
27: **return** $\mathbf{x}$

5.6 Heuristics for Choosing and Parameterizing the Tools

As we could see, interval analysis provides us a rich set of procedures and techniques to process boxes in B&BT algorithms. To develop an efficient algorithm for a specific problem, we need to choose these tools that will have good performance for this problem (and for a specific box!) and parameterize these tools properly.

For instance, the GS operator usually will not perform well on relatively large boxes; consistency operators will be much more efficient. On the other hand, for sufficiently small boxes, the GS operator will often be unmatched—unless for boxes where f is singular or at least ill-conditioned.

Hull-consistency outperforms box-consistency for simple constraints—especially, when each variable occurs only once in the formula. For more complicated constraints, the dependency problem (cf. Chap. 2) often makes HC inefficient. Yet, there are problems with relatively complicated formulae, for which HC still performs better than BC (see, e.g., [37]).

So, what should be applied in practice? There is no simple answer to this question; we need to develop proper heuristics and policies. Several papers of the author have been devoted to this goal: [30, 32–37]. For efforts taken by other researchers, see, e.g., [14, 16, 18, 23, 24, 27].

We shall not present these heuristics here in details. What should be emphasized is that static heuristics and policies have only a limited potential of performance improvement. As the author observed in [37], intelligent heuristics, based on machine learning techniques should be applied, so that the solver could adapt to the features of a specific problem. Up to now, very little has been done in this field; the only paper known to the author is [17].

References

1. GUDHI C++ library (2017). http://gudhi.gforge.inria.fr/
2. HIBA_USNE, C++ library (2017). https://www.researchgate.net/publication/316687827_HIBA_USNE_Heuristical_Interval_Branch-and-prune_Algorithm_for_Underdetermined_and_well-determined_Systems_of_Nonlinear_Equations_-_Beta_25
3. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: International Conference on Logic Programming, pp. 230–244. The MIT Press (1999)
4. Benhamou, F., McAllester, D., Hentenryck, P.V.: CLP(intervals) revisited. In: Proceedings of the 1994 International Symposium on Logic Programming, pp. 124–138. The MIT Press (1994)
5. Borkowski, T.: Comparison of existence tests of zeros of equations systems in a given region: tests of Miranda, Borsuk and Newton. Master's thesis, ICCE WUT (2013). (under supervision of Bartłomiej J. Kubica). (in Polish)
6. Brown, E.H.: Finite computability of Postnikov complexes. Ann. Math., 1–20 (1957)
7. Collavizza, H., Delobel, F., Rueher, M.: Comparing partial consistencies. In: Csendes, T. (ed.) Developments in Reliable Computing, pp. 213–228. Springer, Netherlands (1999)
8. van Emden, M.H.: Computing functional and relational box consistency by structured propagation in atomic constraint systems (2001). arXiv preprint arXiv:cs/0106008
9. Franek, P., Krčál, M.: Robust satisfiability of systems of equations. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 193–203. SIAM (2014)
10. Franek, P., Ratschan, S.: Effective topological degree computation based on interval arithmetic. Math. Comput. **84**(293), 1265–1290 (2015)
11. Frommer, A., Lang, B.: On preconditioners for the Borsuk existence test. PAMM **4**(1), 638–639 (2004)
12. Frommer, A., Lang, B.: Existence tests for solutions of nonlinear equations using Borsuk's theorem. SIAM J. Numer. Anal. **43**(3), 1348–1361 (2005)

13. Goldsztejn, A.: Comparison of the Hansen-Sengupta and the Frommer-Lang-Schnurr existence tests. Computing **79**(1), 53–60 (2007)
14. Goualard, F.: On considering an interval constraint solving algorithm as a free-steering nonlinear Gauss-Seidel procedure. In: Proceedings of the 2005 ACM Symposium on Applied Computing, pp. 1434–1438. ACM (2005)
15. Goualard, F., Goldsztejn, A.: A data-parallel algorithm to reliably solve systems of nonlinear equations. In: Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2008, pp. 39–46. IEEE (2008)
16. Goualard, F., Jermann, C.: On the selection of a transversal to solve nonlinear systems with interval arithmetic. Comput. Sci. ICCS **2006**, 332–339 (2006)
17. Goualard, F., Jermann, C.: A reinforcement learning approach to interval constraint propagation. Constraints **13**(1–2), 206–226 (2008)
18. Granvilliers, L.: On the combination of interval constraint solvers. Reliab. Comput. **7**(6), 467–483 (2001)
19. Granvilliers, L., Benhamou, F.: Progress in the solving of a circuit design problem. J. Glob. Optim. **20**(2), 155–168 (2001)
20. Gutowski, M.W.: Introduction to interval calculi and methods. BEL Studio, Warszawa (2004). (in Polish)
21. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
22. Herbort, S., Ratz, D.: Improving the efficiency of a nonlinear-system-solver using the componentwise Newton method. Technical Report 02/97, Institut für Angewandte Mathematik, Universität Karslruhe (1997)
23. Ishii, D., Goldsztejn, A., Jermann, C.: Interval-based projection method for under-constrained numerical systems. Constraints **17**(4), 432–460 (2012)
24. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
25. Jaulin, L., Walter, É.: Set inversion via interval analysis for nonlinear bounded-error estimation. Automatica **29**(4), 1053–1064 (1993)
26. Kearfott, R.B.: An efficient degree-computation method for a generalized method of bisection. Numer. Math. **32**(2), 109–127 (1979)
27. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
28. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Vychislennyie Tiehnologii (Computational Technologies) **15**(1), 7–13 (2010)
29. Kolev, L.V.: Some ideas towards global optimization of improved efficiency. In: GICOLAG Workshop, Wien, Austria, pp. 4–8 (2006)
30. Kubica, B.J.: Performance inversion of interval Newton narrowing operators. Prace Naukowe Politechniki Warszawskiej. Elektronika **169**, 111–119 (2009). KAEiOG 2009 (Konferencja Algorytmy Ewolucyjne i Optymalizacja Globalna) Proceedings
31. Kubica, B.J.: Interval methods for solving underdetermined nonlinear equations systems. Reliab. Comput. **15**, 207–217 (2011). SCAN 2008 Proceedings
32. Kubica, B.J.: Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 467–476 (2012)
33. Kubica, B.J.: Excluding regions using Sobol sequences in an interval branch-and-prune method for nonlinear systems. Reliab. Comput. **19**(4), 385–397 (2014). SCAN 2012 (15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics) Proceedings
34. Kubica, B.J.: Using quadratic approximations in an interval method for solving underdetermined and well-determined nonlinear systems. In: PPAM 2013 Proceedings. Lecture Notes in Computer Science, vol. 8385, pp. 623–633 (2014)

35. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. Numer. Algorithms **70**(4), 929–963 (2015). https://doi.org/10.1007/s11075-015-9980-y
36. Kubica, B.J.: Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. J. Parallel Distrib. Comput. **107**, 57–66 (2017). https://doi.org/10.1016/j.jpdc.2017.03.009
37. Kubica, B.J.: Role of hull-consistency in the HIBA_USNE multithreaded solver for nonlinear systems. In: PPAM 2017 Proceedings. Lecture Notes in Computer Science, vol. 10778, pp. 381–390 (2018)
38. Kubica, B.J., Woźniak, A.: Interval methods for computing the Pareto-front of a multicriterial problem. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, pp. 1382–1391 (2009)
39. Neumaier, A.: The enclosure of solutions of parameter-dependent systems of equations. In: Reliability in Computing: The Role of Interval Methods in Scientific Computing, pp. 269–285. Academic Press Professional, Inc. (1988)
40. Puget, J.F., Hentenryck, P.V.: A constraint satisfaction approach to a circuit design problem. J. Glob. Optim. **13**(1), 75–93 (1998)
41. Ratschek, H., Rokne, J.: Experiments using interval analysis for solving a circuit design problem. J. Glob. Optim. **3**(4), 501–518 (1993)

Chapter 6
Solving Quantified Problems Using Interval Methods

6.1 Interval Global Optimization

Global optimization is one of the first problems for which the interval B&BT algorithm has been applied; actually, it has been the eponymous B&B algorithm (see [8, 12, 33] and references therein).

This applies to unconstrained:

$$\begin{aligned} &\min_x f(x)\,, \\ &\text{s.t.} \\ &x \in [\underline{x}, \overline{x}]\,, \end{aligned} \tag{6.1}$$

inequality constrained version:

$$\begin{aligned} &\min_x f(x)\,, \\ &\text{s.t.} \\ &g(x) \leq 0\,, \\ &x \in [\underline{x}, \overline{x}]\,, \end{aligned} \tag{6.2}$$

and equality constrained version:

$$\begin{aligned} &\min_x f(x)\,, \\ &\text{s.t.} \\ &g(x) \leq 0\,, \\ &h(x) = 0\,, \\ &x \in [\underline{x}, \overline{x}]\,, \end{aligned} \tag{6.3}$$

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_6

In all three cases, $\mathbf{x} = [\underline{x}, \overline{x}] \subseteq \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$.

For the first two cases, the feasible set has a nonempty interior (at least typically), while for equality constrained problems, the feasible set is of measure zero.

There are a few algorithms to solve problems (6.1), (6.3) and (6.2), e.g., the Moore-Skelboe algorithm, Hansen-Sengupta, etc. (cf, e.g., Chap. 9 of [9]).

They differ in several important details, but are all instances of the B&B algorithm.

6.1.1 Branch-and-Bound Algorithm

The interval B&B for global optimization is an instance of Algorithm 1 with its subprograms: Algorithms 2 and 3. Let us present this instance as Algorithm 11; this is not the only possible formulation, as we discuss later, but the one most important for our purposes.

Remark 6.1 Interval B&B algorithm for global optimization is often formulated differently than in Algorithm 11. The most important difference are lines 31–36. Often both boxes after bisection are stored in L and a new box $\mathbf{x}$ is always selected from L. This allows to change the main loop of the algorithm to a `while` loop:

```
...
while (L ≠ ∅) do
  choose from L the pair (x, y) with the smallest y
  ...
  bisect (x), obtaining x^(1) and x^(2)
  [y_(1), ȳ^(1)] = f(x^(1))
  store (L, (x^(1), y_(1)))
  [y_(2), ȳ^(2)] = f(x^(2))
  store (L, (x^(2), y_(2)))
...
```

Nevertheless, the author would like to keep the form presented in Algorithm 11. The order of boxes processing seems less relevant to the algorithm's efficiency than avoiding one insertion to a priority queue.

This topic has been discussed also in Chap. 4, in Remark 4.1.

Tools used to process a box depend on problem version (6.1–6.2); we shall discuss them in Sect. 6.1.2.

In any case, the algorithm version has the following features, distinguishing it from other instances of the B&BT algorithm:

- the order of processing boxes is very significant,
- the second phase is simple, but inevitable.

The Order of Processing Boxes

Algorithm 11 The branch-and-bound method for global optimization

Require: $\mathbf{x}^0$, $\mathsf{f}(\cdot)$, $\mathsf{g}(\cdot)$, ε {f – objective, g – constraints}
$L_{ver} = L_{pos} = \emptyset$
$[\underline{y}^{(0)}, \overline{y}^{(0)}] = \mathsf{f}(\mathbf{x}^{(0)})$
$L = \{\}$
$y_{opt} = \overline{y}^{(0)}$
$\mathbf{x} = \mathbf{x}^{(0)}$
new_box = **false**
loop
  if (new_box) **then**
    if ($L == \emptyset$) **then**
      break {All pairs from L have been considered}
    choose from L the pair ($\mathbf{x}$, y) with the smallest y
  new_box = **false**
  process the box $\mathbf{x}$, using the rejection/reduction tests
  update y_{opt} if possible
  if (it was verified that $\mathbf{x}$ contains no solutions) **then**
    new_box = **true**
    continue
  else if (it was verified that $\mathbf{x}$ contains a single stationary point) **then**
    narrow (x) as possible, using interval tools
    push (L_{ver}, $\mathbf{x}$)
    new_box = **true**
  else if (wid ($\mathbf{x}$) $< \varepsilon$) **then**
    push (L_{pos}, $\mathbf{x}$)
    new_box = **true**
  else if (the tests resulted in two subboxes of $\mathbf{x}$: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) **then**
    $\mathbf{x} = \mathbf{x}^{(1)}$
    store (L, ($\mathbf{x}^{(2)}$, $\underline{y}^{(2)}$))
    cycle loop
  else
    bisect ($\mathbf{x}$), obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
  $[\underline{y}^{(1)}, \overline{y}^{(1)}] = \mathsf{f}(\mathbf{x}^{(1)})$
  $[\underline{y}^{(2)}, \overline{y}^{(2)}] = \mathsf{f}(\mathbf{x}^{(2)})$
  if ($\underline{y}^{(2)} < \underline{y}^{(1)}$) **then**
    swap (($\mathbf{x}^{(1)}$, $\underline{y}^{(1)}$), ($\mathbf{x}^{(2)}$, $\underline{y}^{(2)}$))
  $\mathbf{x} = \mathbf{x}^{(1)}$
  store (L, ($\mathbf{x}^{(2)}$, $\underline{y}^{(2)}$))
{Second phase – verification}
for all ($\mathbf{x} \in L_{ver}$) **do**
  if ($\mathsf{f}(\mathbf{x}) > y_{opt}$) **then**
    discard $\mathbf{x}$
for all ($\mathbf{x} \in L_{pos}$) **do**
  if ($\mathsf{f}(\mathbf{x}) > y_{opt}$) **then**
    discard $\mathbf{x}$
return L_{ver}, L_{pos}

Usually, the boxes are sorted according to increasing lower bound of the objective function. Specifically, we store pairs ($\mathbf{x}$, $\underline{y}$), where $[\underline{y}, \overline{y}] = \mathsf{f}(\mathbf{x})$; such pairs are usually stored in some sort of priority queue.

Indeed, such an ordering and taking the pair with the lowest $\underline{y}$ if each iteration guarantees the convergence of the algorithm. However, it is not the optimal approach. According to an interesting paper of Shary [39], it is better to choose a pair with sufficiently small, but not necessarily smallest $\underline{y}$. It even occurred to be beneficial to randomize the chosen box, i.e., to choose the random one of the first few in the priority queue. It is worth noting that compatible results have been obtained for Lipschitz optimization, by Lera and Sergeyev [27].

In [28], yet another approach has been described, replacing the priority queue by a pair of lists: "promising boxes" with sufficiently small $\underline{y}$ and the list of other boxes. This approach seems quite successful and particularly sufficient for multithreaded implementations; cf. Chap. 7.

The Second Phase of Global Optimization Algorithm

The second phase of Algorithm 11 is simple, but important. It boils down to scanning the lists L_{ver} and L_{pos} (resp. other solution list(s) assembled by another versions of this algorithm) and removing from them all solution boxes containing local (and not global) optima.

While the procedure itself is simple, it should be emphasized that such final scan has to be done *after* phase one is completed: otherwise, we would not be certain about the value of y_{opt}. In the case of a distributed implementation (cf. Chap. 7), this phase requires synchronization between all instances performing the first phase, and executing some sort of reduction operation to compute the minimum of all "local" values of y_{opt}.

Bisection

Which variable to subdivied and in what manner? This problem has been extensively studied by several authors. The most common concept of bisecting symmetrically the longest edge of the box (so-called, maximal diameter bisection):

$$k = \operatorname*{argmax}_i \operatorname{wid}\mathbf{x}_i \ ,$$

is widely accepted as convergent, but it is definitely not optimal—at least not for all problems.

Another common approach is to subdivide the component with maximal *smear*:

$$k = \operatorname*{argmax}_i \operatorname{wid}\left(\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}_i} \cdot \mathbf{x}_i\right) .$$

This idea has been independently proposed by Csendes [36] and Shary (cf. [40]). It works well for unconstrained problems, but for constrained ones more subtle policies may become handy (see, e.g., [12]).

Some researchers (e.g., [3, 35]; see also [12], Paragraph 5.1.2) have also suggested to use multisection, but its usefulness is questionable. Details have been given in Sect. 4.9.

Other Possible Interval Global Optimization Algorithms

Section 5.1 of [12] makes an interesting survey of various interval global optimization algorithms. Besides the versions we already know, several other ones are mentioned. Some of them make no use of derivatives or are targeted at very specific class of problems.

One of the algorithms worth mentioning is [4]. The idea is to seek the approximate global minimizer with a non-interval algorithm, prior to the actual B&B procedure. Many professional solvers can be applied here; also multithreaded or distributed ones. The obtained minimizer should then be validated and a very good initial value of y_{opt} gets computed this way. Many subregions can thus be efficiently deleted, using some kind of constraint propagation on the constraint $f(x) \leq y_{opt}$.

Yet another interesting algorithm for global optimization has been proposed by Shary [38]. In this approach, the B&B procedure is performed not in the space $\mathbb{R}^n \ni x$, but $\mathbb{R}^{n+1} \ni (x, y)$, i.e., the objective y is treated as one of the variables, that can be bisected as well as the other variables. A similar idea has been applied by the author for multicriteria problems; see below, Sect. 6.2.

6.1.2 Processing a Box in Interval Global Optimization

Tools to process a single box in interval B&B global optimization algorithms have been extensively studied by several authors. As there exist several textbooks (including [8, 12]) and papers, only a brief description will be given here.

It is also worth noting that many of these techniques are analogous to ones used for solving nonlinear equations systems, presented in Chap. 5. For instance, the interval Newton operators or various constraint propagation techniques can be applied in two manners:

- for solving 0th-order conditions, of the type: $f(x) \leq y_{opt}$;
- for solving 1st-order conditions, like $\nabla f(x) = 0$ for unconstrained problems or Fritz John conditions, for constrained ones.

Nevertheless, efficiency of these tools may be different for global optimization than for equations systems. For instance, choice of the preconditioning matrix in the interval Newton operator turns out to be less important for the former problem than for the latter [11].

In general, it is worth to distinguish two situations, when the used tools will differ:

- unconstrained and inequality-constrained problems: then the feasible set has (usually) a non-empty interior and it is relatively easy to find feasible points;
- equality-constrained problems: then the feasible set is of measure zero and obtaining a verified feasible point requires solving a system of (typically underdetermined) nonlinear equations [13].

Many tools, like the midpoint test or monotonicity test [11] are not applicable in the second case. Quoted books (particularly [12]) and papers (e.g., [15–17, 31, 41]) discuss other (less common) tools, also.

6.2 Pareto Sets of Multicritria Problems

In many branches of engineering and decision-making, we encounter "optimization" problems with several criteria. Other words, we are interested in solving the following problem:

$$\begin{array}{ll} \min\limits_{x} q_k(x) & k = 1, \ldots, N \ , \\ \text{s.t.} & \\ g_j(x) \leq 0 & j = 1, \ldots, m \ , \\ x_i \in [\underline{x}_i, \overline{x}_i] & i = 1, \ldots, n \ . \end{array} \tag{6.4}$$

Often, we can aggregate all criteria $q_k(x)$ into a single value, but sometimes we are interested in finding the whole Pareto frontier, defined below.

Definition 6.1 A feasible point x is *Pareto-optimal* (non-dominated), if there exists no other feasible point x' such that:

$$\begin{array}{ll} (\forall k) & q_k(y) \leq q_k(x) \text{ and} \\ (\exists i) & q_i(y) < q_i(x) \ . \end{array}$$

The set $P \subset \mathbb{R}^n$ of all Pareto-optimal points (Pareto-points) is called the *Pareto-optimal set*. The *Pareto frontier* is the image of the Pareto-optimal set.

For convenience, in the sequel, they both will be called *Pareto sets*.

Computing the Pareto sets (or even approximating them precisely enough) is a hard task, especially for nonlinear problems. Interval methods turned out to be a useful tool to obtains such an approximation. They can be used in at least three manners:

- reducing the problem to repeated unicriterion global optimization;
- B&BT procedure performed in the decision space;
- B&BT procedure performed in the criteria space and resulting boxes inverted to the decision space.

Let us present them briefly.

Reducing to Global Optimization

Fernandez and Toth [6] present a sophisticated algorithm for a bicriteria problem. One of the criteria is treated as the objective and the other as a constraint. A sequence of constrained optimization problems:

$$\begin{aligned} &\min_x f_1(x) \,, \\ &\text{s.t.} \\ &f_2(x) \leq y_2 \,, \\ &x \in X \,, \end{aligned} \tag{6.5}$$

is then solved starting with $y_2 = +\infty$ and restricting the constraint subsequently. Precisely: for each of these optimization problems, the set of ε-optimal solutions is approximated, until this set becomes empty.

Thanks to reducing the problem to global optimization, tools analogous to typical global optimization problems can be used. As much information as possible is extracted from the solution of each (6.5) problems; in particular, boxes that cannot contain solutions of further optimization problems get discarded, not to be considered in subsequent algorithm's stages. For all details, the reader is referred to [6].

The algorithm seems efficient, but it can hardly be generalized to the case of larger number of criteria (and such a generalization would be quite inefficient, probably).

The B&BT Search in Decision Space of the Problem

This approach, most similar to algorithms currently used in global optimization or equations solving, was presented in the paper of Ruetsch [37]. While processing the boxes he uses two kinds of tests to discard them:

- comparing the bounds on criteria values in boxes, to delete dominated ones,
- a "differential approach"—some procedure using the information about gradients, probably equivalent (or almost equivalent) to the "multicriteria variant of the monotonicity test" introduced by Kubica and Woźniak [19].

Unfortunately, the paper [37] lacks several important informations, useful to implement the algorithm:

- How are the boxes stored and chosen for comparisons? In particular, is linear search necessary?
- Details about the "differential procedure".

The B&BT Search in Criteria Space of the Problem

This approach was first presented by Barichard and Hao [2] and than in a series of papers of Kubica and Woźniak [19, 21, 22, 24, 25].

Barichard and Hao proposed an algorithm storing pairs of boxes—the box in decision and in criteria space. They perform bisection in the criteria space and use constraint propagation to narrow the corresponding box in decision space. Also, some algorithm, called "substitution procedure" is used to find a feasible point in each box. Having feasibility of some points verified, boxes dominated by them can be discarded.

Paper [2] lacks several important details about the substitution and the constraint propagation procedures, which makes it difficult to implement it. Also, presented results lack the information about computation time or number of iterations, criteria evaluations.

Kubica and Woźniak [18, 22] proposed an algorithm similar in general assumptions, but different in significant details. Among others, boxes obtained from b&b procedure are inverted to the decision space. This means that with each box $\mathbf{y}$ in the criteria space a set of boxes $\{\mathbf{x}\}$ in the decision space is associated, not a single box. This set is represented by three lists of boxes:

- boxes verified to lie in the interior of reverse image of $\mathbf{y}$,
- boundary boxes of the reverse image of $\mathbf{y}$,
- boxes yet to be checked.

To invert the boxes a version of the SIVIA procedure [10] is used. In contrast to classical SIVIA, the procedure in Kubica and Woźniak's algorithm is broken when the first interior box is found—even, if there are still unchecked boxes. This allows a relatively early approximation of the Pareto front and consequently a possibility to discard dominated boxes in the criteria space.

After obtaining the desired accuracy (given by ε_y parameter) of the Pareto front approximation, we enter the "second phase" in which all boxes from the decision space, yet unchecked (because SIVIA was broken), are investigated until a desired accuracy ε_x is obtained. This second phase does not allow discarding boxes from the criteria space or affect the approximation of the Pareto front.

Algorithm 12 The branch-and-bound method in the criteria space

Require: $\mathbf{x}^0, \mathsf{f}(\cdot), \mathsf{g}(\cdot), \varepsilon_x, \varepsilon_y$ $\{\mathsf{f} = (\mathsf{f}_1, \ldots, \mathsf{f}_N)^T$ – criteria, g – constraints$\}$
1: $\mathbf{y}^{(0)} = \mathsf{f}(\mathbf{x}^{(0)})$
2: $L = \left\{\left(\mathbf{y}^{(0)}, \{\}, \{\}, \{\mathbf{x}^{(0)}\}\right)\right\}$
3: **while** (there is a quadruple in L, for which wid $\mathbf{y} \geq \varepsilon_y$) **do**
4: take this quadruple $(\mathbf{y}, L_{\text{in}}, L_{\text{bound}}, L_{\text{unchecked}})$ from L
5: bisect $\mathbf{y}$ to $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$
6: **for** i = 1, 2 **do**
7: apply `SIVIA` with accuracy ε_x to quadruple $(\mathbf{y}^{(i)}, L_{\text{in}}, L_{\text{bound}}, L_{\text{unchecked}})$, breaking it after finding the first feasible box
8: **if** (the resulting quadruple has a nonempty interior, i.e., $L_{\text{in}} \neq \emptyset$) **then**
9: delete quadruples that are dominated by $\overline{y}^{(i)}$
10: insert the quadruple to the end of L
11: {Second phase – finishing to invert the Pareto frontier}
12: **for all** (quadruple in L) **do**
13: process boxes from $L_{\text{unchecked}}$ until all of them get to L_{in} or L_{bound}
14: **return** L_{ver}, L_{pos}

6.2.1 Tools

Tools that can be applied in all three algorithms to compute Pareto sets are (at least to some extent) distinct in all cases. In his earlier paper, the author has proposed several tools for Algorithm 12. In particular, a multicriterion analog of the monotonicity test

has been introduced [19], 2nd-order Pareto-optimality conditions (analogous to Fritz John conditions) [24] and specialized heuristics for bisection [25].

Other notable tools have been introduced, in particular, in [7, 29, 30].

6.3 Game Solutions

Seeking solutions of continuous games is one of the most sophisticated (yet still practical) instances of Problem (1.1). It has been considered in the author's papers [14, 20, 23, 26].

The game theory tries to predict decisions and/or advise the decision makers on how to behave in a situation when several players (sometimes called "agents") have to choose their behavior that will also influence the others. In the game theory, the behavior of a separate player can be described by its "strategy", and we suppose that the i-th player chooses the strategy $x^i \in X_i$. Usually, it is assumed that each player tends to minimize their cost function (or maximize their utility) represented by $q_i(x^1, \ldots, x^n)$.

So, each of the decision makers solves the following problem:

$$\begin{aligned} &\min_{x^i} q_i(x^1, \ldots, x^n) \,, \\ &\text{s.t.} \\ &x^i \in X_i \,. \end{aligned} \tag{6.6}$$

What solution are they going to choose?

The most commonly considered is the so-called Nash equilibrium point [34]. It can be defined as a situation (an assignment of strategies to all players), when each player's strategy is optimal against those of the others. Formally, the tuple $x^* = (x^{1*}, \ldots, x^{n*})$ is a Nash equilibrium, iff

$$(\forall i \in \{1, \ldots, n\})\, (\forall x^i \in X_i)\; q_i(x^{1*}, \ldots, x^{i-1*}, x^i, x^{i+1*}, \ldots, x^{n*}) \geq q_i(x^{1*}, \ldots, x^{n*}) \,.$$

The notion of a NE has several interesting (and non-obvious) theoretical and practical features (in particular, contrary to a popular belief, they are not necessarily self-enforcing [5]), which are however out of the scope of this monograph.

Also, they have several important drawbacks—both theoretical (rather strong assumptions about the players' knowledge and rationality) and practical (they can be Pareto-inefficient, i.e., it is possible to improve the outcome of one player, without worsening results of the others [32]).

Hence, several "refinements" to the notion of NE have been introduced, including, in particular, the strong Nash equilibrium (SNE); see [1]. For such points, not only none of the players can improve his performance by changing strategy, but also no *coalition* of the players can improve the performance of all of its members, by mutually deviating from the SNE. Formally:

$$(\forall I \subseteq \{1,\ldots,n\})\left(\forall x^I \in \times_{i\in I} X_i\right)(\exists i \in I) \quad q_i(x^{\setminus I*}, x^I) \geq q_i(x^{\setminus I*}, x^{I*})\,. \quad (6.7)$$

6.3.1 Algorithm

Algorithm 13 presents the version of B&BT method (the generic Algorithm 1), specialized for seeking game equilibria. A similar algorithm works for NE and SNE. The differences are in two points:

- details of processing a single box;
- details of verifying the box in the 2nd phase.

We get back to these differences in the next subsection.

Algorithm 13 The branch-and-bound-type method for seeking NE/SNE

Require: $\mathbf{x}^0$, $\mathsf{q}(\cdot)$, ε
$L_{ver} = L_{pos} = L_{check} = L_{small} = \emptyset$
$\mathbf{x} = \mathbf{x}^{(0)}$
loop
  $\mathbf{x}^{old} = \mathbf{x}$
  process the box $\mathbf{x}$, trying to verify if it does or does not contain a point satisfying the necessary conditions of being a solution
  if ($\mathbf{x}$ was discarded, but not all q_i's are monotonous on it) **then**
    push $(L_{check}, \mathbf{x}^{old})$
    discard $\mathbf{x}$
  else if (the tests resulted in two subboxes of $\mathbf{x}$: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) **then**
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push $(L, \mathbf{x}^{(2)})$
    cycle loop
  else if (wid $(\mathbf{x}) < \varepsilon$) **then**
    push $(L_{small}, \mathbf{x})$
  if ($\mathbf{x}$ was discarded **or** $\mathbf{x}$ was stored) **then**
    $\mathbf{x}$ = pop (L)
    if (L was empty) **then**
      break
  else
    bisect $(\mathbf{x})$, obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push $(L, \mathbf{x}^{(2)})$
{Second phase – verification}
for all ($\mathbf{x} \in L_{small}$) **do**
  check if another solution from L_{small} does not invalidate $\mathbf{x}$
  verify if no box from L_{check} contains a point that would invalidate $\mathbf{x}$
  put $\mathbf{x}$ to L_{ver}, L_{pos} or discard it, according to the results
return L_{ver}, L_{pos}

6.3.2 *Tools*

Optimal tools and heuristics for B&BT algorithms seeking NE/SNE still require further studies. The author has done some preliminary research in this area, providing, in particular, the analog of monotonicity test for verifying SNE [26] or emphasizing the importance of testing concavity of players' criterion functions and investigating the possibilities of eigenvalues bounding, for this purpose [14].

Obviously, the tools differ slightly depending on if in request is NE, SNE or possibly yet another kind of game solution. Papers [20, 26] discuss necessary conditions of Nash and strong Nash equilibria (analogous to Karush John conditions [12] or similar conditions for Pareto-optimality [25]). They can be solved using the interval Newton operator or other constraint propagation methods.

It is worth noting that the elaborate character of Algorithm 13 (in particular, its sophisticated 2nd phase) make it particularly hard to parallelize in a distributed environment. Problems with such an implementation, and how the author had dealt with them are discussed in [14]; cf. also Sect. 7.6 of this volume.

6.4 Summary

In this chapter, three important classes of problems have been considered: global optimization, Pareto-sets seeking and game solutions seeking. All of them can be solved using the interval B&BT algorithm. All of them have already been studied, but as for the first problem, these are extensive studies, described in several textbooks, for the second one there are just a few collections of papers. The last problem has been considered mostly by the author of this monograph, in a few of his papers.

All three of the problems still require further studies to develop modern, highly-tuned, parallelized and practical algorithms, by filling the B&BT "skeleton" with proper tools and heuristics, tailored for the given class of problems.

References

1. Aumann, R.J.: Acceptable points in general cooperative games. In: A.W. Tuckar, R.D. Luce (eds.) Contributions to the Theory of Games IV. Princeton University Press (1959)
2. Barichard, V., Hao, J.K.: Population and interval constraint propagation algorithm. Lecture Notes in Computer Science, vol. 2632, pp. 88–101 (2003)
3. Berner, S.: New results on verified global optimization. Computing **57**(4), 323–343 (1996)
4. Caprani, O., Godthaab, B., Madsen, K.: Use of a real-valued local minimum in parallel interval global optimization. Interval Comput. **2**, 71–82 (1993)
5. Clark, K., Kay, S., Sefton, M.: When are Nash equilibria self-enforcing? An experimental analysis. Int. J. Game Theory **29**(4), 495–515 (2001)
6. Fernandez, J., Toth, B.: Obtaining an outer approximation of the efficient set of nonlinear biobjective problems. J. Glob. Optim. **38**, 315–331 (2007)

7. Goldsztejn, A., Domes, F., Chevalier, B.: First order rejection tests for multiple-objective optimization. J. Glob. Optim. **58**(4), 653–672 (2014)
8. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
9. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
10. Jaulin, L., Walter, É.: Set inversion via interval analysis for nonlinear bounded-error estimation. Automatica **29**(4), 1053–1064 (1993)
11. Kearfott, R.B.: An interval branch and bound algorithm for bound constrained optimization problems. J. Glob. Optim. **2**(3), 259–280 (1992)
12. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
13. Kearfott, R.B.: On proving existence of feasible points in equality constrained optimization problems. Math. Program. **83**(1–3), 89–100 (1998)
14. Kubica, B.J.: Advanced interval tools for computing solutions of continuous games. Vychislennyie Tiehnologii (Computational Technologies) **23**(1), 3–18 (2018)
15. Kubica, B.J., Malinowski, K.: An interval global optimization algorithm combining symbolic rewriting and componentwise Newton method applied to control a class of queueing systems. Reliab. Comput. **11**(5), 393–411 (2005)
16. Kubica, B.J., Malinowski, K.: Optimization of performance of queuing systems with long-tailed service times. Prace Naukowe Politechniki Warszawskiej. Elektronika **156**, 237–245 (2006)
17. Kubica, B.J., Niewiadomska-Szynkiewicz, E.: An improved interval global optimization algorithm and its application to price management problem. In: PARA 2006 Proceedings. Lecture Notes in Computer Science, vol. 4699, pp. 1055–1064 (2007)
18. Kubica, B.J., Woźniak, A.: Interval componentwise Newton operator in computing the Pareto-front of constrained multicriterial problems. In: Proceedings of KKA 2008 Conference (2008)
19. Kubica, B.J., Woźniak, A.: Interval methods for computing the Pareto-front of a multicriterial problem. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, pp. 1382–1391 (2009)
20. Kubica, B.J., Woźniak, A.: An interval method for seeking the Nash equilibria of non-cooperative games. In: PPAM 2009 Proceedings. Lecture Notes in Computer Science, vol. 6068, pp. 446–455 (2010)
21. Kubica, B.J., Woźniak, A.: A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment. In: PARA 2008 Proceedings. Lecture Notes in Computer Science, vol. 6126/6127. Accepted for Publication (2010)
22. Kubica, B.J., Woźniak, A.: Optimization of the multi-threaded interval algorithm for the Pareto-set computation. J. Telecommun. Inf. Technol. **1**, 70–75 (2010)
23. Kubica, B.J., Woźniak, A.: Applying an interval method for a four agent economy analysis. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 477–483 (2012)
24. Kubica, B.J., Woźniak, A.: Using the second-order information in Pareto-set computations of a multi-criteria problem. In: PARA 2010 Proceedings. Lecture Notes in Computer Science, vol. 7134, pp. 137–147 (2012)
25. Kubica, B.J., Woźniak, A.: Tuning the interval algorithm for seeking Pareto sets of multi-criteria problems. In: PARA 2012 Proceedings. Lecture Notes in Computer Science, vol. 7782, pp. 504–517 (2013)
26. Kubica, B.J., Woźniak, A.: Interval methods for computing strong Nash equilibria of continuous games. Decis. Mak. Manuf. Serv. **9**(1), 63–78 (2015). SING10 Proceedings
27. Lera, D., Sergeyev, Y.D.: Deterministic global optimization using space-filling curves and multiple estimates of Lipschitz and Hölder constants. Commun. Nonlinear Sci. Numer. Simul. **23**(1), 328–342 (2015)
28. Lyudvin, D.Y., Shary, S.P.: Testing implementations of PPS-methods for interval linear systems. Reliab. Comput. **19**(2), 176–196 (2013). SCAN 2012 Proceedings

29. Martin, B.: Rigorous algorithms for nonlinear biobjective optimization. Ph.D. thesis, Université de Nantes (2014)
30. Martin, B., Goldsztejn, A., Granvilliers, L., Jermann, C.: Constraint propagation using dominance in interval branch & bound for nonlinear biobjective optimization. Eur. J. Oper. Res. **260**(3), 934–948 (2017)
31. Martínez, J., Casado, L.G., García, I., Sergeyev, Y.D., Toth, B.: On an efficient use of gradient information for accelerating interval global optimization algorithms. Numer. Algorithms **37**(1–4), 61–69 (2004)
32. Miettinen, K.: Nonlinear Multiobjective Optimization, vol. 12. Kluwer Academic Publishers, Dordrecht (1999)
33. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
34. Nash, J.F.: Equilibrium points in n-person games. Proc. Natl. Assoc. Sci. **36**, 48–49 (1950)
35. Pal, L.: Global optimization algorithms for bound constrained problems. Ph.D. thesis, University of Szeged (2010)
36. Ratz, D., Csendes, T.: On the selection of subdivision directions in interval branch-and-bound methods for global optimization. J. Glob. Optim. **7**, 183–207 (1995)
37. Ruetsch, G.R.: An interval algorithm for multi-objective optimization. Struct. Multidiscip. Optim. **30**(1), 27–37 (2005)
38. Shary, S.P.: A surprising approach in interval global optimization. Reliab. Comput. **7**(6), 497–505 (2001)
39. Shary, S.P.: Randomized algorithms in interval global optimization. Numer. Anal. Appl. **1**(4), 376–389 (2008)
40. Shary, S.P.: Finite-dimensional Interval Analysis. Institute of Computational Technologies, SB RAS, Novosibirsk (2013)
41. Toth, B., Casado, L.G.: Multi-dimensional pruning from the Baumann point in an interval global optimization algorithm. J. Glob. Optim. **38**(2), 215–236 (2007)

Chapter 7
Parallelization of B&BT Algorithms

7.1 Introduction

In the Chap. 4, the author has presented the class of interval branch-and-bound-type (B&BT) algorithms for solving problems of the form (1.1):

$$\text{Find } all\ x \in X \text{ such that } P(x) \text{ is fulfilled,}$$

where $P(x)$ is a formula with a free variable x and $X \subseteq \mathbb{R}^n$. This is a pretty general class, containing, i.a., equations systems, constraint satisfaction problems (CSPs), global optimization, seeking Pareto-optimal points of a multicriteria problem, seeking solutions of a game and many others.

The structure and various details and issues of such algorithms have been described in Chap. 4. In the present chapter, we are going to discuss implementation of B&BT algorithms. This chapter is mostly a survey, but it contains some original considerations, also.

Before we get to the merit, let us remind the general structure of such algorithms, for the sake of completeness.

7.2 Generic Algorithm

The generic B&BT algorithm can be expressed by the pseudocode, presented in Algorithm 1.

This algorithm consists of two phases: the actual B&BT method (Algorithm 2) and the second phase, when the results are checked (if it is necessary; Algorithm 3).

All details can be found in Chap. 4; also notation from Chap. 4 is preserved:

- L—the list of initial boxes;
- $P(x)$—the predicate formula, defining the problem under consideration;

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_7

Algorithm 14 The overall algorithm

Require: L, P
1: perform the initial exclusion phase on L (if sufficient)
2: perform the essential branch-and-bound-type method (i.e., Algorithm 2) for (L, P), storing the results in L_{ver}, L_{pos}, L_{check}
3: {The second phase}
4: perform the verification (i.e., Algorithm 3) for L_{ver}, L_{check}, P
5: perform the verification (i.e., Algorithm 3) for L_{pos}, L_{check}, P

Algorithm 15 The essential generalized branch-and-bound method

Require: L, P
$L_{ver} = L_{pos} = L_{check} = \emptyset$
x = pop (L)
loop
  process the box **x**, using the rejection/reduction tests
  update the *shared quantities* (if any)
  if (**x** does not contain solutions) **then**
    if CHECK(**x**) **then**
      push (L_{check}, **x**)
    discard **x**
  else if VERIF(**x**) **then**
    push (L_{ver}, **x**)
  else if (the tests resulted in two subboxes of **x**: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) **then**
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push (L, $\mathbf{x}^{(2)}$)
    cycle loop
  else if (**x** is small enough) **then**
    push (L_{pos}, **x**)
  if (**x** was discarded **or** **x** was stored) **then**
    x = pop (L)
    if (L was empty) **then**
      return L_{ver}, L_{pos}, L_{check}
  else
    bisect (**x**), obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
    $\mathbf{x} = \mathbf{x}^{(1)}$
    push (L, $\mathbf{x}^{(2)}$)

Algorithm 16 Verification of solutions

Require: L_{sol}, L_{check}, P
for all ($\mathbf{x} \in L_{sol}$) **do**
  discard **x** if it does not contain any point $x \in \mathbf{x}$, satisfying $P(x)$
  {details of the verification depend on P, but the *shared quantities* and, possibly, the boxes from L_{check} are useful there}

- the lists/sets of solutions: L_{ver}—verified solution boxes, L_{pos}—possible solution boxes;
- L_{check}—boxes (possibly, with some additional information) that can be used in the second phase to verify boxes from L_{ver} and L_{pos}, if needed;

- VERIF(**x**)—the box **x** has been verified to contain a solution/a point satisfying some necessary conditions to be a solution;
- CHECK(**x**)—the box **x** does not contains a solution, yet it can be useful to verify P for some other box in the second phase.

7.3 Basic Implementation Details

Let us start with discussing two issues of each implementation: used data structures and memory management. As B&BT algorithms tend to be memory-demanding, these issues are pretty important even for a serial implementation; for a parallel one, their importance is even higher.

7.3.1 Data Structures

The most basic issue of the implementation is to choose a proper data structure to store sets of boxes: L, L_{ver}, L_{pos} and L_{check}. Several data structures might be appropriate: a vector, a linked list, various realizations of a stack, a queue, a priority queue or many others. It depends on the order in which the boxes should be processed—if this order is relevant—and on various other implementation details.

An extremely important factor is the efficiency under the conditions of parallelization. This topic is going to be considered later in the paper (consult Sect. 7.5).

Another important factor is efficient memory management, which will be discussed in the next subsection.

7.3.2 Memory Management

Typically, B&BT algorithms process a large number of boxes. How to store these boxes? How to organize the memory for storing them? As pointed in [32], the implementation that keeps boxes on the heap (which means that, in C++, we use `new/delete` operators) is suboptimal at least for two reasons:

- the default C++ allocator is not tuned for allocating relatively small objects; it is a simple wrapper over the C `malloc/free` operations with some additional overhead; cf. [4];
- memory management in a multithreaded environment happens to be particularly inefficient.

There are a few possibilities for improvement. It is possible to use a specialized memory allocator, e.g., the Small Object Allocator from the Loki library [37] or another one (for details, consult [32]).

Another possibility (and probably the best one) is to use the *move semantics*, introduced in the C++11 standard. This allows to avoid using dynamic variables, but cannot be applied if we decide to store boxes on linked lists. Yet, it is compatible with containers, like, e.g., `std::vector` or `tbb::concurrent_vector` (from the TBB library [25]). Results from [32] seem to confirm the superiority of the move semantics over dynamic allocation of memory, but further investigations seem required to give an unambiguous answer.

7.4 Parallelization of the B&BT Algorithm

Now, let us get to the main topic: parallelization of the considered class of algorithms. Both phases of a B&BT algorithm parallelize well, as different boxes can be processed independently. Yet, they are (in general) *not* "embarrassingly parallel"—for both basic kinds of parallel implementation (shared or distributed memory) some problems have to be addressed.

The next two sections discuss shared- and distributed-memory parallelization possibilities of a B&BT algorithm.

7.5 Shared Memory Implementations

The main difficulties of multithreaded implementation of B&BT algorithms are:

- MT-safe (i.e., multithreaded-safe) implementation of the procedure processing a single box,
- efficient and MT-safe storage of the set of boxes to be considered (L),
- efficient and MT-safe storage of lists/sets L_{ver} and L_{pos},
- efficient and MT-safe storage of L_{check} and other *shared quantities* (if any).

Even the basic "building block" of the algorithm—the procedure dealing with a single box—does not have to be MT-safe, in general. The C-XSC library [2], the package that the author is using, guarantees safe arithmetic operations, midpoint computation (but only since version 2.3.0!), etc., but the automatic differentiation code has been made MT-safe far later. See [34] for several details.

Also, e.g., some linear programming solvers (like GLPK [1]) are not MT-safe, so using LP-preconditioners (see, e.g., [20]) or LP-narrowing (see, e.g., [19, 21]) is not straightforward in multithreaded solvers. The author does not use them in his current codes.

7.5.1 Storage of L

As mentioned in Sect. 7.3.1, several data structures can be used to store boxes: depending on the problem under consideration, it can be a stack, a queue, a priority queue, etc. Whatever data structure is used, operations on it have to be synchronized (unless each thread has a private list, which is an approach characteristic for distributed memory systems, described in next subsection).

A single lock, guarding the whole data structure L is a valid approach, but it becomes inefficient for a higher number of cooperating threads. There are several efficient implementations of MT-safe structures (see, e.g., [27] and the references therein, for a discussion of this topic) and a good collection of such implementations can be found, e.g., in the TBB (Threading Building Blocks) library [3].

Still, there is yet another possibility and—in several situations—it can be recommended. For several programming frameworks, including TBB, L does not have to be represented explicitly; we can rely on the internal mechanism of task servicing. For example, the author's solver of nonlinear equations systems uses the TBB task-based parallelism, by utilizing programming concepts `tbb::parallel_do` and `tbb::parallel_do_feeder<T>` (see TBB documentation at [3]) to implement the branch-and-prune method (see [25, 26, 28, 29, 31]).

However, for global optimization and other problems, where the order of processing boxes is relevant, we may have to represent the list L, explicitly. A priority queue (`tbb::priority_queue<T>`) seems like a proper solution, but not the only one. Lyudvin and Shary [38], among with other authors, claim, it is not beneficial to choose the box with the smallest lower bound, always. This sounds, like a very good news to parallel programmers and the approach of [38]—the two lists: "basic" and "cache"— might allow utilizing all advantages of the task-based parallelism. The author used it in his solver for MPC problems [30].

7.5.2 Storage of L_{ver} and L_{pos}

If the sets of verified and possible solutions are to be shared between all working threads, the access to them has to be synchronized, too. As for L, discussed in the previous paragraph, using a single lock for the whole list does not seem the optimal approach, as we have several concurrent implementations of various data structures, that are much more scalable.

What should we use? A concurrent vector (implemented as a list of arrays) seems a proper choice; TBB [3] provides us a sufficient class for this purpose: `tbb::concurrent_vector<T>` (and so do other libraries, certainly, e.g., Microsoft Parallel Patterns Library or other TBB's competitors).

Yet, if we wish to stick to using a linked list, there is another very good option, also. Instead of using any kind of mutex, we can insert elements in a lock-free manner,

using an *atomic exchange* instruction. Such an instruction can be defined, by the following pseudocode:

Algorithm 17 XCHG

Require: $var, value$
1: $tmp = var$
2: $var = value$
3: **return** tmp

Using this atomic operation, we can perform a very simple, efficient and wait-free [17] operation of element insertion (provided, no-one removes the element from the list, concurrently with the insertion operation, but this is the case):

Algorithm 18 insert

Require: $head, elem$
1: $tmp = \text{XCHG}(head, elem)$
2: $elem->next = tmp$

The procedure is wait-free, as it does not require any retrial or iteration, as methods based on the compare-and-swap operation [17]. A similar procedure is used in Vyukov's algorithms for a non-blocking queue (presented on his web page [49]).

The problem is that the XCHG operation might not be available on assembler languages, using other instruction sets, than x86. Also, relatively few software packages provide this operation for high-level languages, like C/C++, but Intel TBB and the C++11 standard—do. So do CUDA, but GPU programming is yet quite another story (see, e.g., an interesting discussion in [6]).

7.5.3 Shared Quantities

As already indicated, solving equations (systems), CSPs and other problems, not requiring quantifier elimination (or the second phase—Algorithm 3) do not require any *shared quantities* or other shared objects, except—at most—L, L_{ver} and L_{pos}.

But B&BT algorithms for many more sophisticated problems need some *shared quantities*. The simplest case is global optimization, where we need a single floating-point number, representing the upper bound on the global minimum. Such a quantity can be secured by a single lock; an active waiting lock (a spin-lock) is appropriate here, usually, as the operations of updating (or reading) of the single number are relatively quick.

For the problem of seeking the Pareto sets, we need to keep the set of several points (or boxes!), representing an approximation of the Pareto frontier. In the author's

paper [35], it is proposed to protect the Pareto frontier approximation by a readers-writer lock. The results were promising, but—probably—we can do better, by using efficient MT-safe data structures. Such experiments are planned in the near future.

In [27, 33], the author stated that an interval tree might be of great use in representing the shared data in interval B&BT methods. A careful investigation of this idea did not verify it. When we need data related to specific regions of the search domain—as in seeking (strong) Nash equilibria [36]—it is better to rely on the list L_{check} of boxes, as in Algorithm 2. Otherwise, other data structures are pretty sufficient, as discussed in the previous paragraph. The applicability of interval trees seems very limited.

7.6 Distributed Memory Implementations

Environments with the local memory, like MPI [41], seem less appropriate for parallelization of B&BT algorithms, unless we consider the simplest versions with no *shared quantities*. But, even for the case of equations system or CSP solving, we have to address at least two non trivial issues:

- box migration between the nodes for load balancing,
- termination detection.

There are a few approaches to solve both above issues.

7.6.1 Load Balancing

This problem has been studied by several authors; see, e.g., [7, 16, 48] and references therein. To some extent, it is analogous to load balancing between threads, e.g., in TBB (see [25]; also the documentation in [3]). All load-balancing policies are purely heuristical, obviously.

The author does not have his own experiences, yet, but it seems the policy should take the network topology into account. Please note, the modern standard MPI-3 brings several topology-aware features [18].

7.6.2 Termination Detection

Also, detecting that all of the cooperating nodes have become idle, is all but trivial. The paper [39] surveys several techniques for termination detection; this is an abstract study, not related to B&BT algorithms, specifically.

A specific policy to detect termination, usually must depend on the network topology. Ring topology seems particularly convenient. Also, weight-throwing methods of detection might be useful for interval B&BT algorithms.

7.6.3 Advanced Issues

Real difficulties start when implementing algorithm versions for more sophisticated problems—the ones that have to *share* more information. In environments with local memory, nothing is shared, naturally.

For global optimization, when we share a single floating-point number, we can have its local "photographs" on each node and update them according to some protocol.

Yet, synchronizing the approximation of the whole Pareto frontier, like that, does not seem, to be a good idea. There are a few possibilities:

- devote one of the nodes to maintain a shared resource and respond on queries from other nodes,
- distribute the resource, e.g., each of the nodes keeps some parts of the approximated Pareto frontier,
- re-design the algorithm to use the "shared" data as late as possible.

All of these approaches seem to have some advantages, but which of them is acceptable in practice, remains to be determined. The last one seems the most compatible with the idea of communication-avoiding algorithms (see, e.g., [5]), which is the recent trend in distributed algorithms design. But is it possible to re-design B&BT algorithms that way?

Analogous issues are related to storing the L_{check} list, e.g., in algorithms computing (strong) Nash equilibria.

In particular, in [32], the author presents an MPI-based version of the B&BT algorithm for seeking solution of continuous games. In this implementation, each node stores its own lists of solutions and its own sub-list of L_{check}. In the verification phase, sub-lists of L_{check} are not moved, but the solutions peregrinate through all nodes in order, to be verified using all boxes from all sub-list of L_{check}.

After they have gone around all nodes, the root node gathers all results to assemble "global" lists L_{ver} and L_{pos}. For details, the reader is referred to [32].

It is worth noting that the library CXSC-MPI [9] has not been updated for a few years and has several drawbacks. In particular, it lacks:

- functions `MPI_Send_recv()` or `MPI_Send_recv_replace()`;
- non-blocking operations other than sending, in particular `MPI_Irecv()`;
- collective operations more sophisticated than `MPI_Bcast()`, in particular `MPI_Scatter()`, `MPI_Gather()`, `MPI_Scatterv()` or `MPI_Gatherv()`.

These limitations can be worked around in a few ways; for instance, an interval could be replaced with a pair of floating-point numbers. Unfortunately, this would

require wasting time on transforming the data structures. Another possibility is to use pack/unpack functions for sending noncontiguous data. Details and the example solution, developed by the author, have been described in [32].

7.7 Parallelization of Rejection/Reduction Tests

In previous sections, we assumed the following schema of parallelization: various boxes, produced by the B&BT process, are processed concurrently, but processing a single box is carried out by a serial procedure.

Rarely have been rejection/reduction tests on boxes parallelized so far, but there are exceptions to this rule. For instance, in [31] the procedure to enforce bound consistency is parallelized.

Also, much effort has been put to parallelizing operations on interval matrices. Several early papers can be found in references of [24]. Newer investigations usually try to utilize BLAS libraries for interval matrices; see, e.g., [42]; cf. also [10, 11].

It might seem that parallel matrix operations can be applied to parallelizing rejection/reduction tests directly. The author's opinion is different. Techniques used in parallelizing BLAS operations are tuned for matrices of dimension above thousands, which is much larger than the Hesse matrices, or other matrices, encountered in typical nonlinear problems tractable for B&BT algorithms (usually, far below one hundred). BLAS libraries might help, because of efficient cache utilization and vectorization (cf. [44]), but not parallelization; at least in the predictable future.

The master's thesis [43] considers parallelization of algorithms for computing eigenvalues of interval matrices. And checking the sign of eigenvalues is an important rejection test, e.g., for optimization or game solution seeking (see [32]).

7.7.1 Parallelization of Existence Tests

A separate couple of words should be devoted to parallelization of existence tests of nonlinear systems. The simplest tests, the classical ones based on the interval Newton operator (see, e.g., [20, 40]) or Miranda's theorem [20], are probably too simple to be parallelized efficiently, but this is not true for more complicated ones.

In particular, the new tests, using the toolset of algebraic topology: the Borsuk theorem [8], topological degree theory [14, 15] or, especially, (co)homotopy and (co)homology theory [12, 13] are much more complicated and computationally intensive. Operations on simplicial sets or cell complexes, used in the latter approaches, seem pretty well suited for parallelization.

On the other hand, the importance of existence tests does not seem crucial, as they can be used only for verification of solutions that have already been found.

Nevertheless, the role of parallelized (or vectorized) rejection/reduction tests is likely to be increasing. With the advent of many-core architectures, parallelization of

the algorithm should be multi-level and scalable. And a parallelized test is more likely to utilize hyper-threads, working on the same core (they share the same cache!), than concurrent processing of unrelated boxes (resulting in cache misses, inevitably).

7.7.2 Modern Architectures

Finally, it is worth noting, that tuning such implementations for modern architectures is an issue itself. In [31], the author described his efforts to tune the bound-consistency enforcing procedure for the Intel Xeon Phi coprocessor; he has only partially successful.

Tuning the algorithms may require in-depth knowledge about the parameters of a specific device; auto-tuning techniques should probably be applied here, also.

These remarks apply to GPUs [22, 23], Xeon Phi and similar coprocessors [46, 47] and, in particular, to hybrid architectures, combining a few different devices [45].

7.8 Summary

This chapter discussed several issues of parallel implementations of interval B&BT algorithms: both shared- and distributed-memory ones. Synchronization, sharing data structures, memory management, and other issues have been discussed.

In the author's opinion, discussion on parallelization of rejection/reduction tests (in Sect. 7.7) was particularly innovative.

Also other remarks form this chapter are crucial for a successful implementation of a B&BT algorithm, a universal and well-suited for current hardware architectures approach to solve various decision problems; namely, problems of seeking points that satisfy a certain condition, specified in a first-order logic.

References

1. GNU Linear Programming Kit (2014). http://www.gnu.org/software/glpk/
2. C++ eXtended Scientific Computing library (2015). http://www.xsc.de
3. Intel Threading Building Blocks (2017). http://www.threadingbuildingblocks.org
4. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
5. Baboulin, M., Donfack, S., Dongarra, J., Grigori, L., Rémy, A., Tomov, S.: A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. Procedia Comput. Sci. **9**, 17–26 (2012)
6. Beck, P.D., Nehmeier, M.: Parallel interval Newton method on CUDA. In: PARA 2012 Proceedings. Lecture Notes in Computer Science, vol. 7782, pp. 454–464 (2013)
7. Berner, S.: Parallel methods for verified global optimization practice and theory. J. Glob. Optim. **9**(1), 1–22 (1996)

8. Borkowski, T.: Comparison of existence tests of zeros of equations systems in a given region: tests of Miranda, Borsuk and Newton. Master's thesis, ICCE WUT (2013). (under supervision of Bartłomiej J. Kubica). (in Polish)
9. CXSC-MPI: MPI extension for the use of C-XSC in parallel environments (2015). http://www2.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html#cxsc_mpi
10. Dąbrowski, R., Kubica, B.J.: Comparison of interval C/C++ libraries in global optimization. Prace Naukowe Politechniki Warszawskiej. Elektronika **169**, 51–56 (2009)
11. Dąbrowski, R., Kubica, B.J.: Cache-oblivious algorithms and matrix formats for computations on interval matrices. Lecture Notes in Computer Science, vol. 7134, pp. 269–279 (2012)
12. Franek, P., Krčál, M.: Robust satisfiability of systems of equations. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 193–203. SIAM (2014)
13. Franek, P., Krčál, M.: Cohomotopy groups capture robust properties of zero sets (2015). arXiv preprint arXiv:1507.04310
14. Franek, P., Ratschan, S.: Effective topological degree computation based on interval arithmetic. Math. Comput. **84**(293), 1265–1290 (2015)
15. Frommer, A., Hoxha, F., Lang, B.: Proving the existence of zeros using the topological degree and interval arithmetic. J. Comput. Appl. Math. **199**(2), 397–402 (2007)
16. Gau, C.Y., Stadtherr, M.A.: Dynamic load balancing for parallel interval-Newton using message passing. Comput. Chem. Eng. **26**(6), 811–825 (2002)
17. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Elsevier (2012)
18. Hoffler, T.: Advanced MPI: new features of MPI-3 (2016). http://htor.inf.ethz.ch/teaching/mpi_tutorials/speedup15/hoefler-advanced-mpi-speedup15.pdf
19. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
20. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
21. Kolev, L.V.: Some ideas towards global optimization of improved efficiency. In: GICOLAG Workshop, Wien, Austria, pp. 4–8 (2006)
22. Kozikowski, G.: Implementation of an OpenCL library for automatic differentiation. Bachelor's thesis, ICCE WUT (2011). (under supervision of Bartłomiej J. Kubica). (in Polish)
23. Kozikowski, G., Papamanousakis, G., Yang, J.: Potential future exposure, modelling and accelerating on GPU and FPGA. In: Proceedings of the 8th Workshop on High Performance Computational Finance, WHPCF 2015, pp. 4:1–4:8. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2830556.2830560
24. Kreinovich, V., Bernat, A.: Parallel algorithms for interval computations: an introduction. Interval Comput. **3**, 6–62 (1994)
25. Kubica, B.J.: Intel TBB as a tool for parallelization of an interval solver of nonlinear equations systems. Technical Report 09-02, ICCE WUT (2009)
26. Kubica, B.J.: Shared-memory parallelization of an interval equations systems solver—comparison of toos. Prace Naukowe Politechniki Warszawskiej. Elektronika **169**, 121–128 (2009). KAEiOG 2009 (Konferencja Algorytmy Ewolucyjne i Optymalizacja Globalna) Proceedings
27. Kubica, B.J.: A class of problems that can be solved using interval algorithms. Computing **94**, 271–280 (2012). SCAN 2010 (14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics) Proceedings
28. Kubica, B.J.: Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 467–476 (2012)
29. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. Numer. Algorithms **70**(4), 929–963 (2015). https://doi.org/10.1007/s11075-015-9980-y

30. Kubica, B.J.: Preliminary experiments with an interval Model-Predictive-Control solver. In: PPAM 2015 Proceedings. Lecture Notes in Computer Science, vol. 9574, pp. 464–473 (2016)
31. Kubica, B.J.: Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. J. Parallel Distrib. Comput. **107**, 57–66 (2017). https://doi.org/10.1016/j.jpdc.2017.03.009
32. Kubica, B.J.: Advanced interval tools for computing solutions of continuous games. Vychislennyie Tiehnologii (Computational Technologies) **23**(1), 3–18 (2018)
33. Kubica, B.J., Woźniak, A.: An interval method for seeking the Nash equilibria of non-cooperative games. In: PPAM 2009 Proceedings. Lecture Notes in Computer Science, vol. 6068, pp. 446–455 (2010)
34. Kubica, B.J., Woźniak, A.: A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment. In: PARA 2008 Proceedings. Lecture Notes in Computer Science, vol. 6126/6127. Accepted for Publication (2010)
35. Kubica, B.J., Woźniak, A.: Optimization of the multi-threaded interval algorithm for the Pareto-set computation. J. Telecommun. Inf. Technol. **1**, 70–75 (2010)
36. Kubica, B.J., Woźniak, A.: Interval methods for computing strong Nash equilibria of continuous games. Decis. Mak. Manuf. Serv. **9**(1), 63–78 (2015). SING10 Proceedings
37. Loki: Loki C++ template library (2015). http://loki-lib.sourceforge.net
38. Lyudvin, D.Y., Shary, S.P.: Testing implementations of pps-methods for interval linear systems. Reliab. Comput. **19**(2), 176–196 (2013). SCAN 2012 Proceedings
39. Matocha, J., Camp, T.: A taxonomy of distributed termination detection algorithms. J. Syst. Softw. **43**(3), 207–221 (1998)
40. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
41. MPI: Message Passing Interface (2017). http://www.mpi-forum.org
42. Nguyen, H.D.: Efficient implementation of interval matrix multiplication. Lecture Notes in Computer Science, vol. 7134, pp. 179–188 (2012)
43. Owczarek, B.: Parallel algorithms for computing eigenvalues of interval matrices. Master's thesis, ICCE WUT (2015). (under supervision of Bartłomiej J. Kubica). (in Polish)
44. Skalna, I., Duda, J.: A study on vectorisation and paralellisation of the monotonicity approach. Lecture Notes in Computer Science (2016). Submitted
45. Szustak, Ł., Halbiniak, K., Kuczyński, Ł., Wróbel, J., Kulawik, A.: Porting and optimization of solidification application for CPU-MIC hybrid platforms. Int. J. High Perform. Comput. Appl. (2016). https://doi.org/10.1177/1094342016677740
46. Szustak, Ł., Rojek, K., Olas, T., Kuczynski, Ł., Halbiniak, K., Gepner, P.: Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. Sci. Program. **2015** (2015). https://doi.org/10.1155/2015/642705
47. Szustak, Ł., Rojek, K., Wyrzykowski, R., Gepner, P.: Toward efficient distribution of MPDATA stencil computation on Intel MIC architecture. In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations, pp. 51–56 (2014)
48. Ueberholz, P., Willems, P., Bull, M., Lang, B.: Non-blocking load balancing for branch-and-bound-type algorithms. In: PARA 2008 Proceedings. Lecture Notes in Computer Science. Accepted for Publication (2013)
49. Vyukov, D.: Non-intrusive MPSC node-based queue. http://www.1024cores.net/home/lock-free-algorithms/queues/non-intrusive-mpsc-node-based-queue. (web page; Accessed 2017)

Chapter 8
Interval Software, Libraries and Standards

Throughout the years, several packages and programs have been developed for interval computations. One of the classical interval solvers is GlobSol, written by Ralph B. Kearfott in Fortran [4]. Several papers describe the use of this solver (see, e.g., [28] and the references therein).

There are many other packages for Matlab [10], Python, ADA and for more niche languages such, as OCaml [14] or Julia [5]. The web page [9] lists several of them. Yet, in our survey, the focus is on C++ packages. But before we present them, let us discuss what they *should* contain.

8.1 Main Issues in Implementing Interval Libraries

It might seem, implementing an interval data type and providing basic arithmetic operations on it (as defined in Sect. 2.3), should be relatively easy in any modern object-oriented language. Notwithstanding, there are some issues; most of them are related to correctly performing of the outward rounding, but other representation-related problems can be encountered, also, like:

- Can an interval have infinite bounds or not?
- What will be the result of operations on such infinite bounds, e.g. $[0, 1] \cdot [1, +\infty]$?
- How to treat the value NaN (Not a Number)? Should we have a NaI (Not an Interval), also?

These and other similar questions have found some answers in the IEEE 754 standard for the floating-point operations.

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_8

8.1.1 IEEE 754 Standard

This standard has been published in 1985; the last revision is from 2008 [23]. The standard defines the representation of floating-point numbers and properties of some basic operations on them.

It is worth noting that this standard may not be supported by some devices, e.g., by some embedded devices or older (as well as niche) GPUs. Nevertheless, it is supported by x86 assembler, most modern GPUs (both using CUDA and OpenCL) and most other machines (cf. Sect. 8.3.4).

There is a wide literature on the standard (cf., e.g., Sect. 10 of [26]). Also, in one of the appendices (Appendix B) to this monograph, this standard is described. Many features are discussed there: positive and negative zero, normal and subnormal (denormal) numbers, various kinds of NaN, etc. Here, let us emphasize only the properties relevant for operations on intervals (cf. [20]):

- There are four rounding modes: towards the nearest number, towards zero, upwards and downwards.
- There are infinities: $+\infty$ and $-\infty$; they are different from NaNs.

The rounding mode can be switched, using the proper instruction (in C/C++ we have a `feset()` function; in the x86 assembly language, `fldcw` should be used). Unfortunately, changing the mode is usually a relatively time-consuming operation. And a typical interval operation has to change the rounding mode at least three times: downwards, to compute the lower bound; upwards, to compute the upper bound; then restore the initial rounding mode. An important part of many interval libraries is to minimize then umber of rounding mode changes; some tricks allow us to use only one kind of directed rounding [21].

As for considering intervals with infinite bounds, there are a few possibilities, also; see, e.g., [21, 37].

8.2 C-XSC

This library has been used by the author in his computational experiments, so it will be described in more details than other packages.

The C-XSC library [2], as other XSC languages (the name stands for eXtended Scientific Computing), has been developed in the University of Wuppertal.

Some time ago, the package had been relatively slow (cf., e.g., comparison in [19]), but much has changed since then. The version 2.3.0 has been highly improved and optimized. Support for BLAS operations (see below, Sect. 8.2.2), novel types for sparse vectors and matrices and many other innovative tools have been integrated to the library (cf. [33]).

8.2.1 Basic Types

Intervals are represented by the `cxsc::interval` class. Its endpoints are double-precision floating point numbers; they are not represented by `double` variables, but using a custom `cxsc::real` wrapper class. There are also classes for complex numbers and intervals: `cxsc::complex`, `cxsc::cinterval`.

All these classes have their multiple-precision counterparts (also known as *staggered representations*): `cxsc::l_real`, `cxsc::l_complex`, `cxsc::l_interval`, and `cxsc::l_cinterval`. Afterwards, we have so-called *extended staggered representations* of these entities [15]: `cxsc::lx_real`, `cxsc::lx_complex`, `cxsc::lx_interval`, `cxsc::lx_cinterval`, etc.

It is worth noting that vectors and matrices in C-XSC are not represented using STL templates, but with custom classes: `cxsc::rvector`, `cxsc::rmatrix`, `cxsc::ivector`, `cxsc::imatrix`, `cxsc::l_rvector`, etc.

8.2.2 The Use of BLAS

As for traditional floating-point algorithms, for many interval algorithms, their most intensive steps are operations on matrices and vectors. For traditional computations, several versions of BLAS (Basic Linear Algebra Subroutines) have been developed to provide tuned (or auto-tuning) procedures, optimized for a specific architecture and environment. OpenBLAS, GotoBLAS, MKL or ATLAS are good examples of various BLAS instances.

In what manner are they optimized? It depends on the architecture. Usually, their main focus is efficient use of the cache hierarchy, but multithreading, vectorization and reducing computational complexity are of interest, also.

Interval algorithms can benefit from using BLAS, as well, as pointwise ones. Several papers, including [42, 43] describe algorithms for computing enclosures of matrix products (a real and an interval matrix, two interval matrices, etc.), using a few computations of products of real-valued matrices.

C-XSC can optionally use a given BLAS package for matrix operations. As the author's experiments indicate, this can be very worthwhile [34], but not for all interval algorithms [35].

8.2.3 The Toolbox and Additional Software

The library of C-XSC is pretty rich. We get some automatic differentiation codes, solvers for linear systems with interval coefficients, etc.

On the web page [2], we can find also some additional packages, e.g., for MPI computations with C-XSC [18], computing slopes of functions or some sophisticated solvers, e.g., for differential and integral equations. Some of these tools have been mentioned in other chapters of this monograph.

Unfortunately, the library has not been updated since the February 2014, when version 2.5.4 has been released.

8.2.4 Author's Solvers and Libraries

All interval-related programs and packages developed by the author, at least so-far, have been based on C-XSC. This is true for the ADHC library, described in Chap. 3, HIBA_USNE solver for equations systems, described in Chap. 5, and solvers used in Chap. 6: not made public, yet.

Whether in the future these pieces of software should migrate to another interval library, is an open question.

8.3 Other Libraries

8.3.1 PROFIL/BIAS

This is another widely used C++ interval library; cf., e.g., [26]. The name PROFIL stands for Programmer's Runtime Optimized Fast Interval Library [12]. It is a C++ library with several features, including transcendental functions, automatic differentiation, etc. Actually, it is a C++ interface to lower-level procedures of BIAS.

BIAS.

BIAS is an acronym for Basic Interval Arithmetic Subroutines. It is a set of C functions. As the name suggests, the focus is made on operations on interval matrices. Yet, unlike C-XSC (cf. Sect. 8.2.2), specific BIAS procedures are developed, not basing on non-interval BLAS packages. Their focus is not on efficiently using the cache memory, but on reducing the number of rounding mode switching.

In the author's opinion, the idea is inferior to using BLAS: floating-point BLAS packages are changing constantly, adapting to evolving hardware architectures and utilizing other improvements. BIAS procedures are not able to cope with the development of computer technology. Moreover, they do not seem to be updated since 2009.

8.3.2 Boost::Interval

The Boost libraries [6] are well-known to virtually all C++ programmers. They provide a variety of packages for data structures, algorithms, meta-programming and mathematical notions. In particular, we have the Boost::Interval library [7], providing the interval arithmetic and other basic interval operations.

This package has several interesting features. Firstly, the interval data type it provides is a template class, parameterized by two arguments: type and a policy class; so it looks as follows: `boost::interval<T, policies>`.

The first template argument, is obviously, the underlying type: over which space do we define intervals. What types can be used here? The answer is relatively complicated. Three data types are recommended: `float`, `double` and `long double`, but the set of possible options is not limited to this triple. Ordinal types, like `int` or `boolean` can be used here, also, although some functionality might not be available for them.

On the other hand, some substituting for `T` some types is explicitly forbidden: these are all types that are not totally ordered, like `std::complex<T>`.

The second argument, the policy class describes several features of the actual instantiation of the interval data type:

- What rounding should be used for the endpoints?
- What conditions should be checked during the operations, e.g., do we allow empty or improper intervals, should we throw exceptions when they occur, etc.
- What comparison operation should be used for intervals (cf. the discussion in Sect. 2.6)?

The library is universal, but less optimized than C-XSC or other competitors. This may change in future versions, obviously.

8.3.3 Other Packages

GAOL

Another package worth mentioning is GAOL [8]. The name is a pun: "gaol" means a jail and, according to the opinion of its providers,"GAOL is not JAIL", i.e., "Just Another Interval Library". One of its distinct features is implementing the so-called relational interval arithmetic.

FILIB++

This is another package of C-XSC providers; a C++ version of the older C FI_LIB library. The name stands for Fast Interval LIBrary [1]. The library is minimalistic (no vectors, no matrices, etc.), yet highly optimized for speed, e.g., transcendental functions are computed using *table lookup*. Unfortunately, no new version of the library has been released since 2011.

Moore library

Paper [39] describes yet another modern approach to providing an interval arithmetic library. The paper has been released in 2018 and it claims to use the C++20 standard of the C++ language.

Similarly to Boost::Interval, discussed above, the package makes intensive use of template metaprogramming and other modern C++ tools (including so-called

concepts). According to [39], intervals, interval vectors and matrices are represented in the library; there are also some codes to perform automatic differentiation.

One of its interesting (but controversial; cf. Sect. 2.8) features is the possibility of representing not only closed, but also open and half-open intervals.

To the best knowledge of the author, the library is not publicly available: at least currently.

Yet another libraries

There are many more libraries, obviously. It is worth to mention some multiprecision libraries: MPFR and, based on it, MPFI [22]. To the best knowledge of the author, a pretty good implementation of the interval arithmetic was present in Sun's Fortran compilers. In [25], yet another library, based on long double-precision numbers, has been presented. See papers [19, 44] and references therein, for more information.

IBEX

IBEX [13] is one of the interval solvers that are actively developed. It allows solving CSPs (including equations systems) and optimization problems. The core library defines C++ classes representing intervals, interval vectors, matrices, etc.

8.3.4 GPU Libraries

These packages deserve a separate subsection, as they differ from CPU libraries to the high extent. Firstly, GPUs and their programming interfaces are not as standardized as high-level programming languages for traditional architectures. Not only we have the "dualism" between CUDA and OpenCL, but even various versions ("compute capabilities") of CUDA have a very differentiated API. What is more, some GPUs (especially older ones) are not compatible with the IEEE 754 Standard; in particular, they do not allow directed roundings. Consequently, first interval programs for GPU, had to implement outward rounding in a more sophisticated (and less efficient) manner; see [17].

Cards using OpenCL, as well as modern NViDIA cards using CUDA allow directed roundings. Packages with this feature have been implemented and used; cf. [30–32].

Unfortunately, to the best knowledge of the author, none of this packages is publicly available, as for now.

8.3.5 IEEE Standard 1788–2015: Standard for Interval Arithmetic

This standard [24] has been developed throughout years 2008–2015 [41] by the IEEE P-1788 Working Group for Interval Arithmetic, coordinated by John Pryce [40].

It was an attempt to unify various models, approaches, "philosophies" and definitions of the interval calculus: set-theoretical interval arithmetic, containment set arithmetic, modal interval arithmetic, etc. They are called various *flavors* by authors of the standard; some of them have been mentioned in Chap. 2. As for now, only the "*set-based* flavor" (intervals are sets of *numbers*; if a bound is equal to $\pm\infty$, it is *not* a member of the interval) is included in the standard, but other ones are planned to be included in the revisions of the Standard. This includes the Kaucher flavor [27], allowing improper intervals, like [2, 1].

All of them need to be consistent with the original Moore's arithmetic (called *common intervals*, by the Standard's authors), i.e., the arithmetic of bounded, nonempty, compact intervals [40].

Decorators

Several interesting (and even more not-so-interesting, yet highly important) details are defined by the Interval Standard. We cannot discuss all of them here (please consult papers [40, 41] or the standard itself [24] for details).

Yet, we shall describe one of the features, which turns out to be very innovative with respect to earlier implementation concepts of the interval system: *decorators* of intervals. What does that mean? A decorated interval is a pair $(\mathbf{x}, d)$, where $\mathbf{x}$ is an interval $[\underline{x}, \overline{x}]$ and d is its *decorator*, i.e., a description of definedness, continuity, etc.

The idea of decorators is similar to global flags, defined by the IEEE 754 Standard for floating-point computations. These flags indicate underflow, overflow, or other possible features affecting the quality of the obtained result. In IEEE 1788 Standard, global flags have been replaced by "local" decorators, more adequate for today platforms, that support parallelizm pretty often.

To describe the decorator values, defined by the Standard, let us consider the interval $\mathbf{y} = \mathsf{f}(\mathbf{x})$. Now, the decorator of $\mathbf{y}$ can have the following values:

- `com` (common)—f was defined, continuous and bounded on all points from $\mathbf{x}$,
- `dac` (defined and continuous)—f was defined and continuous on all points from $\mathbf{x}$, although possibly unbounded at some of them,
- `def` (defined)—f was defined (but not necessarily continuous) on all points from $\mathbf{x}$,
- `trv` (trivial)—"no information",
- `ill` (ill-formed)—"Not an Interval", f was not defined on $\mathbf{x}$.

Obviously, the higher the value from the above list, the better the result is considered to be.

Software for IEEE Std 1788–2015

Few libraries are consistent with this standard, up to now. The author is aware of two, only, described in [41]: `libieeep1788` [3], written by Marco Nehmeier, and Octave Interval [11], by Olivier Heimlich.

Developing this standard may not only help to unify interval libraries, but also provide better hardware support for interval analysis [29]. As, in particular, Kulisch

indicates in his papers, present x86 processors already contain all components necessary for the hardware support of interval arithmetic and the exact dot product of interval vectors [16, 36, 38].

References

1. FILIB++ library (2011). http://www2.math.uni-wuppertal.de/wrswt/software/filib.html
2. C++ eXtended Scientific Computing library (2015). http://www.xsc.de
3. The C++ IEEE 1788 library (2015). https://github.com/nehmeier/libieeep1788
4. GlobSol solver (2015). https://interval.louisiana.edu/GlobSol/
5. ValidatedNumerics package (2016). https://github.com/JuliaIntervals/ValidatedNumerics.jl
6. Boost C++ libraries (2017). http://www.boost.org/
7. Boost Interval library (2017). http://www.boost.org/doc/libs/1_66_0/libs/numeric/interval/doc/interval.htm
8. Gaol: NOT Just Another Interval Library (2017). https://sourceforge.net/projects/gaol/
9. Interval and related software (2017). http://www.cs.utep.edu/interval-comp/intsoft.html
10. IntLab. the Matlab/Octave toolbox for reliable computing (2017). http://www.ti3.tu-harburg.de/rump/intlab/
11. Octave Interval library (2017). https://octave.sourceforge.io/interval/
12. PROFIL/BIAS (2017). http://www.ti3.tuhh.de/keil/profil/index_e.html
13. IBEX library (2018). http://www.ibex-lib.org/
14. Alliot, J.M., Gotteland, J.B., Vanaret, C., Durand, N., Gianazza, D.: Implementing an interval computation library for OCaml on x86/amd64 architectures. In: OUD 2012, OCaml Users and Developers Workshop (2012)
15. Blomquist, F.: Staggered correction computations with enhanced accuracy and extremely wide exponent range. Reliab. Comput. **15**(1), 26–35 (2011)
16. Bohlender, G., Kulisch, U.W.: Comments on fast and exact accumulation of products. In: PARA 2010 Proceedings. Lecture Notes in Computer Science, vol. 7134, pp. 148–156 (2012)
17. Collange, S., Flórez, J., Defour, D.: A GPU interval library based on Boost.Interval. In: 8th Conference on Real Numbers and Computers, pp. 61–71 (2008)
18. CXSC-MPI: MPI extension for the use of C-XSC in parallel environments (2015). http://www2.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html#cxsc_mpi
19. Dąbrowski, R., Kubica, B.J.: Comparison of interval C/C++ libraries in global optimization. Prace Naukowe Politechniki Warszawskiej. Elektronika **169**, 51–56 (2009)
20. Goualard, F.: Towards good C++ interval libraries: Tricks and traits. In: the 4th Asian Symposium on Computer Mathematics. Chiang Mai. Citeseer (2000)
21. Goualard, F.: Fast and correct SIMD algorithms for interval arithmetic. In: PARA 2008 Proceedings. Lecture Notes in Computer Science, vol. 6126/6127 (2010). (Accepted for publication)
22. Grimmer, M., Petras, K., Revol, N.: Multiple precision interval packages: comparing different approaches. Lecture Notes in Computer Science, pp. 64–90 (2004)
23. IEEE: 754-2008–IEEE standard for floating-point arithmetic (2008). http://ieeexplore.ieee.org/document/4610935/
24. IEEE: 1788-2015–IEEE standard for interval arithmetic (2015). http://standards.ieee.org/findstds/standard/1788-2015.html
25. Jankowska, M.A.: Remarks on algorithms implemented in some C++ libraries for floating-point conversions and interval arithmetic. In: PPAM 2009 Proceedings. Lecture Notes in Computer Science, vol. 6068, pp. 436–445 (2010)
26. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
27. Kaucher, E.: Interval analysis in the extended interval space $\mathbb{IR}$. In: Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis), pp. 33–49. Springer (1980)

28. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
29. Kirchner, R., Kulisch, U.W.: Hardware support for interval arithmetic. Reliab. Comput. **12**(3), 225–237 (2006)
30. Kozikowski, G.: Implementation of an OpenCL library for automatic differentiation (in Polish). Bachelor's thesis, ICCE WUT (2011). (under supervision of Bartłomiej J. Kubica)
31. Kozikowski, G., Kubica, B.J.: Interval arithmetic and automatic differentiation on GPU using OpenCL. In: PARA 2012 Proceedings. Lecture Notes in Computer Science, vol. 7782, pp. 483–503 (2013)
32. Kozikowski, G., Kubica, B.J.: Parallel approach to Monte Carlo simulation for option price sensitivities using the adjoint and interval analysis. In: PPAM 2013 Proceedings. Lecture Notes in Computer Science, vol. 8385, pp. 600–612 (2014)
33. Krämer, W., Zimmer, M., Hofschuster, W.: Using C-XSC for high performance verified computing. In: PARA 2010 Proceedings. Lecture Notes in Computer Science, vol. 7134, pp. 168–178 (2012)
34. Kubica, B.J.: Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 467–476 (2012)
35. Kubica, B.J., Woźniak, A.: Tuning the interval algorithm for seeking Pareto sets of multicriteria problems. In: PARA 2012 Proceedings. Lecture Notes in Computer Science, vol. 7782, pp. 504–517 (2013)
36. Kulisch, U.: An axiomatic approach to rounded computations. Numerische Mathematik **18**(1), 1–17 (1971)
37. Kulisch, U.: Computer Arithmetic and Validity-Theory, Implementation and Applications. De Gruyter, Berlin, New York (2008)
38. Kulisch, U.: An axiomatic approach to computer arithmetic with an appendix on interval hardware. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 484–495 (2012)
39. Mascarenhas, W.F.: Moore: Interval arithmetic in C++20 (2018). https://arxiv.org/abs/1802.08558
40. Pryce, J.: The forthcoming IEEE standard 1788 for interval arithmetic. In: SCAN 2014 Proceedings. Lecture Notes in Computer Science, vol. 9553, pp. 23–39 (2015)
41. Revol, N.: Introduction to the IEEE 1788-2015 standard for interval arithmetic. In: International Workshop on Numerical Software Verification, pp. 14–21. Springer (2017)
42. Revol, N., Théveny, P.: Parallel implementation of interval matrix multiplication. Reliab. Comput. **19**(1), 91–106 (2013)
43. Rump, S.M.: Fast interval matrix multiplication. Numer. Algorithms **61**(1), 1–34 (2012)
44. Žilinskas, J.: Comparison of packages for interval arithmetic. Informatica **16**(1), 145–154 (2005)

Chapter 9
Applications of Interval B&BT Methods

9.1 Introduction

Interval methods have found applications in several areas; yet, they are still not the "mainstream". This is caused as by the relative hardness of of mastering this tool, as by its relatively high computational demands. Both obstacles should decrease in the future bit by bit: the interval calculus becomes more widely known and accepted constantly and modern, highly-parallel computer architectures allow successful implementations of these demanding algorithms.

As for now, the most prominent areas where interval analysis has been successfully applied are: robotics, chemical engineering and control theory.

This chapter surveys several selected applications; also, it discusses some less known or only potential applications, e.g., in queueing theory.

9.2 Robotics

Robotics is one of the fields where the interval calculus has found pretty many applications. This is not surprising when considering at least the following facts:

- In the description of both kinematics and dynamics of robots, we encounter strongly nonlinear functions.
- While the models are nonlinear, their structure can be described by relatively precise formulae; this is in contrast to, e.g., economic modeling, where most models are rather crude approximations.
- We have to estimate several quantities from measurements that are (by their nature) imprecise; the measurement error can usually be bound by an interval.

Interval methods find applications in designing robots [31], kinematics computations, path planning [45] and solving other problems.

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3_9

9.2.1 Manipulator Kinematics

The most common problems solved for manipulator kinematics are: the *forward kinematic problem* (when we know the configuration of the manipulator and we need to compute the position of the effector) and the *inverse kinematics problem* (when we know the effector's position and we need to determine the manipulator's configuration).

Variables used to describe the manipulator's configuration vary, depending on the type of manipulator (serial or parallel one) and the type of used joints: rotational, transitional, spherical, etc.

As an example, let us consider solving the inverse kinematic problem of a serial planar *nR*-manipulator, i.e., a manipulator working in the *XOY* space and consisting of *n* rotational joints. Assume, the kinematic chain starts in the point (0, 0) and the effector is supposed to be placed in the point (1, 1) and oriented orthogonally (under the right angle) to the *OY* axis. This problem can be formulated as the following system of equations:

$$\begin{aligned}
&\sum_{i=1}^{n} l_i \cdot \cos\Big(\sum_{j=1}^{i} x_j\Big) - 1 = 0\,,\\
&\sum_{i=1}^{n} l_i \cdot \sin\Big(\sum_{j=1}^{i} x_j\Big) - 1 = 0\,,\\
&\sum_{i=1}^{n} x_i - \frac{\pi}{2} = 0\,,\\
&x_i \in \Big[-\frac{\pi}{2}, \frac{\pi}{2}\Big], \quad i = 1, \ldots, n\,.
\end{aligned} \tag{9.1}$$

Obviously, x_i's represent angles between subsequent links and l_i's—lengths of these links. We assume $l_i = 1.0$ for $i = 1, \ldots, n$.

For $n = 3$ the problem is well determined—there are exactly two manipulator configurations satisfying the constraints (see Fig. 9.1, on the left). But for $n = 5$, the set of possible manipulator configurations is a manifold—it is of the measure continuum. A few example configurations are presented on the right part of Fig. 9.1

This problem has been considered, in particular, in the author's papers [58, 60].

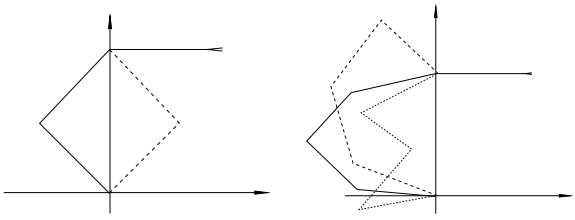

Fig. 9.1 Left: both feasible 3R manipulator configurations, right: three examples of uncountably many feasible 5R manipulator configurations

To avoid costly computations of trigonometric functions, angles are often represented by pairs (s, c), consisting of their sines and cosines. Obviously, because of the Pythagorean identity, we have $s^2 + c^2 = 1$, for each pair.

Problem (9.1), expressed in this manner, takes the following form:

$$\begin{aligned}
&\sum_{i=1}^{n} l_i \cdot c_i - 1 = 0\,,\\
&\sum_{i=1}^{n} l_i \cdot s_i - 1 = 0\,,\\
&s_i^2 + c_i^2 - 1 = 0, \quad i = 1, \ldots, n\,,\\
&c_n = 0\,,\\
&s_n = 1\,,\\
&s_i, c_i \in [-1, 1], \quad i = 1, \ldots, n\,.
\end{aligned}$$

Obviously, $s_i = \sin \sum_{j=1}^{i} x_j$, $c_i = \cos \sum_{j=1}^{i} x_j$, where x_j's come from (9.1). As c_n and s_n are, in fact, precisely known, we can reduce the above system to the following form:

$$\begin{aligned}
&\sum_{i=1}^{n-1} l_i \cdot c_i - 1 = 0\,,\\
&\sum_{i=1}^{n-1} l_i \cdot s_i + l_n - 1 = 0\,,\\
&s_i^2 + c_i^2 - 1 = 0, \quad i = 1, \ldots, n-1\,,\\
&s_i, c_i \in [-1, 1], \quad i = 1, \ldots, n-1\,.
\end{aligned} \tag{9.2}$$

Some well-known benchmarks, e.g., "Puma" or "Robot kinematics" [2] have such origin, also.

There are several papers describing other similar problems and their interval-based solutions. In particular, solving the forward kinematics problem for the Steward-Gough platform has been extensively studied; cf. [80–83] and Chap. 8 of [45].

9.2.2 Mobile Robots

9.2.2.1 Robot Localization

The problem of mobile robot localization, in its traditional formulation, is founded on the assumption that we know the environment, but do not know the current position of the robot. We need to estimate this position from the measurements

of the robot's sensors: the odometer (which is usually the least accurate), cameras, sonars, radars, etc. This problem is sometimes facetiously called "the kidnapped robot problem" [16].

There can be several formulations of this problem and several assumptions we can make. Initially, let us assume that the robot is immobile and the map is described, e.g., as a set of segments, representing the walls. Measurements, provided by sonars, cameras and other sensors, give the robot the information about its distance from walls and other objects. Now, knowing the map, the robot has to answer the question: in which location is it possible to see, such an environment? This reduces to a CSP of the form: find x such that $f(x) \in [\underline{y}, \overline{y}]$, where x represents the robot's pose, f represents the distances from the obstacles, seen in a specific pose and $[\underline{y}, \overline{y}]$ is the range of these distances, provided by the sensors.

Obviously, many questions should be answered to even describe the function f, not to mention solving the problem. Section 8.4.2 of [45] describes the model of sensors, allowing to bound their error. Other parts of Chap. 8 of the book [45] discusses several other details.

Also, specific tests have been developed for solving the underlying CSP [52]; according to the claims of the authors, they reduce the search region pretty efficiently.

What about the situation, when the robot is not immobile? Then providing the good initial estimate is still quite important. When the initial location is known with enough precision, the robot can be tracked using the Kalman filter or its bounded-error versions [45].

Another problem is that the map may be inaccurate or outdated. Then we face the presence of so-called *outliers*, inevitable in robotics, but present also in other estimation problems. Estimation in the presence of outliers will be described in Sect. 9.3.

Finally, the map may be completely unknown. Such a situation is described in the next paragraph.

9.2.2.2 SLAM

The problem of SLAM (Simultaneous Localization and Mapping) [24] is of great importance in mobile robotics. In such problems we have no (or very limited) knowledge not only of the initial position of the robot, but also on its surroundings, the landmarks it may observe, etc. So, not only does the robot have to estimate its position, but also to identify and localize the landmarks and describe the map of the surrounding terrain.

Several (often incompatible) approaches have been proposed for solving the SLAM problem; see, e.g., [24]. Interval methods have extensively been applied there, in various manners; in particular, the group of Jaulin has developed various SLAM methods for underwater robots: [5, 43, 44, 106]. It is worth noting that the interval approach succeeded in some cases where other approaches failed [79].

In general, we can distinguish two main approaches to SLAM:

- feature-based SLAM—we treat coordinates of landmarks, obstacles, etc. as variables (together with the robot pose) and solve the resulting CSP problem,
- location-based SLAM—we define on the (2D or 3D) space of possible robot locations a function (called *occupancy map*), returning zero if the point is free and 1 if the point is occupied by an obstacle; then we perform a SIVIA-like procedure to enclose the "map"—the set of occupied points.

The first approach is usually cheaper for situations, when we are interested only in point landmarks or when obstacles to localize have regular shapes (as on Fig. 9.2).

Hence in a natural environment, with quite irregular obstacles, it would be far less useful (Fig. 9.3).

In general, the occupancy map does not have to be binary; it can return the probability that a point is occupied. Also, we could consider a fuzzy map, where points can be occupied to some degree. Nevertheless, we are interested in points where the robot can or cannot move, so such fuzzy values would need to be defuzzified for path planning (cf. Sect. 9.2.3) or other processing.

Feature-based SLAM is adopted, e.g., in the *CuikSLAM* method of Porta [89]. In this approach, we use a kinematic solver, analogous as for manipulators, to solve distance constraints between the robot and various landmarks. This approach has

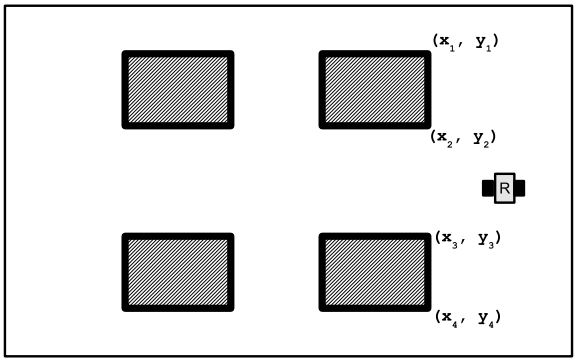

Fig. 9.2 The robot in an environment with regular obstacles

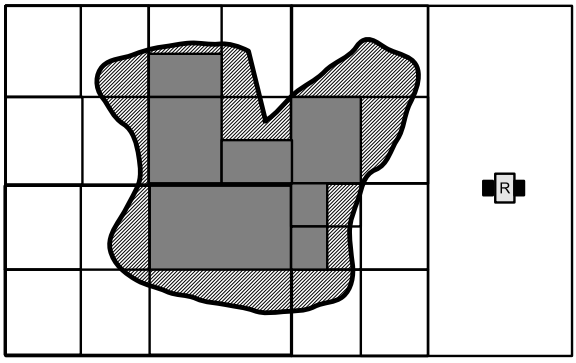

Fig. 9.3 The robot in an environment with irregular obstacles

been generalized by the author to the case when landmarks cannot be distinguished faultlessly [67], but only preliminary experimental results are available, as for now.

And how to identify the landmarks? Image processing and object detection methods have to be used there. Several such methods have been developed; artificial intelligence techniques are a good example. As we discuss in Sect. 9.4, interval methods can find applications at this level, as well.

9.2.3 Path Planning

When we have the map and know the approximate robot position, we may want to compute a feasible path to follow—both, for manipulators and mobile robots. The path should be guaranteed not to collide with any obstacles; also, it should probably be verified to intersect with some areas, the robot has to visit.

Interval methods are useful there, also [42].

Assume, there are two robot positions: a and b and we wish to find the path from a to b. Obviously, a and b should be feasible. The simplest approach is to enclose the set of all feasible robot positions (using some version of SIVIA algorithm) and then use the Dijkstra's algorithm (or another graph algorithm) to find the path from a to b. More sophisticated algorithms make use of the fact that the Dijkstra's algorithm is much quicker than the branch-and-bound type procedure, so we can try to seek the path repeatedly, even before finishing the exhaustive search for feasible positions [45].

9.3 Measurements and Estimation

All measurements are inaccurate. The measurement error is usually modeled using probabilistic techniques; often it is assumed to have a normal distribution.

In his book [27] and several papers (i.a., [26, 28–30]) Gutowski thoroughly criticizes traditional statistical techniques. Instead, an approach based on interval analysis is proposed. Similar ideas have been introduced by other authors—in particular, [9, 10, 55, 101] and many others.

9.3.1 Parameter Estimation

In the simplest case, basing on the measurements, we intend to determine some parameters that are fixed—they have identical values for all measurements.. Other words, we need to find p such that:

$$y = f(x, p) \; . \tag{9.3}$$

Assume we know n pairs $(\mathbf{x}_i, \mathbf{y}_i)$ to satisfy the above relation, but we do not know p—at most we have some crude bounds on it. Problem (9.3) can be linear (e.g., [26]) or nonlinear (e.g., [28, 102]).

In both cases, we can formulate the problem in two manners: either as an optimization problem:

$$\min_p ||y - f(x, p)|| \,, \tag{9.3'}$$

or as a CSP:

$$\text{Find the set } \{p \mid \mathbf{f}(\mathbf{x}_i, p) \subseteq \mathbf{y}_i \; i = 1, \ldots, n\} \,. \tag{9.3''}$$

Both approaches have their advantages and drawbacks. Using (9.3'), we can verify if the model is correct, i.e., if it is consistent with all the measurements. On the other hand, formulation (9.3") allows us to find a solution, even if the model is not strictly correct—or if there are some outliers (see below, Sect. 9.3.3).

Anyway, both problems can be solved using interval algorithms, described in Chaps. 5 and 6.

9.3.2 State Estimation

State estimation is a specific kind of parameter estimation. If one of the parameters is not constant, we treat its values in different time moments as different parameters: $p(0)$, $p(1)$, $p(2)$, etc.

We shall meet this kind of estimation problems, in particular, in Model-Predictive-Control, that is described in Sect. 9.5.

9.3.3 Outliers

The outliers are measurements for which our model does not hold: usually, the error is out of range that we assumed. There are several reasons for outliers' occurrences: malfunctioning of the hardware, unexpected events, etc.

As an example, we can assume the mobile robot, estimating its position, using sensors' data (cf. Sect. 9.2.2). The sensors provide us the information about distances from the obstacles. But in many situations, another, unknown obstacle may occur: a person enters the room, a column in a drowned city gets moved by the water, etc.

In such situations, an object that is not present on the map (or lack of an object present on the map) influences the measurements, resulting in outliers. There can be two results of such a situation:

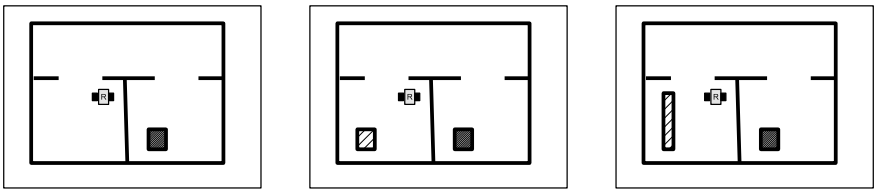

Fig. 9.4 Left: the original map—the robot can determine, in which room it is, thanks to the presence of a pillar, center: an additional object may confuse the robot and make it think, it is in the other room, right: now, as the additional object has different shape than the pillar, the robot will hopefully realize that something is wrong—it should nowhere see what it does

- the set of values of the estimated quantity (e.g., the robot's position) is empty,
- the set of values is nonempty, but erroneous.

The first situation is actually less dangerous, as the error gets detected, while in the second case it escapes detection. Such situations are illustrated by Fig. 9.4.

How to solve problems of type Eq. (9.3")? There are two basic approaches:

- seek solutions that satisfy at least k out of n inequalities,
- seek solutions that approximately satisfy all solutions, i.e., they satisfy $\mathsf{f}(\mathbf{x}_i, p) \subseteq [\underline{y}_i - \varepsilon, \overline{y}_i + \varepsilon]$ for some $\varepsilon > 0$.

9.3.4 *Processing Statistical Samples Under Interval Uncertainty*

Finally, a few words have to be devoted to processing statistical data under interval and/or fuzzy uncertainty. Computing the expected value of an interval-valued sample is straightforward:

$$\mathbb{E}\,\mathbf{x} = \frac{1}{N} \cdot \sum_{i=1}^{N} \mathbf{x}_i \,, \tag{9.4}$$

but computing the sample's variance or covariance is much more complicated. Actually, it has turned out [22] that while computing the lower bound of the variance is tractable, computing (with a prescribed accuracy) the upper bound is NP-hard (it is a bound of a general quadratic function).

Hence, computing the median or other quantiles is, again, simple. The domain is interesting and worth further studies; developing interval-based tests for various statistical hypotheses is going to be very useful, while far from simple.

9.4 Artificial Intelligence Systems

The topic of AI tools is extremely broad (and pretty hot these days). Even a brief survey is completely beyond the scope of this monograph. Let us just mention two of the AI techniques, where interval methods have found some applications: artificial neural networks and so-called support vector machines.

9.4.1 Neural Networks

Artificial neural networks (ANN) are commonly used for classification and for many other tasks. The idea is to create a structure that works in a manner that mimics the way human (or animal) brain works—obviously, in a severe simplification.

A single neuron is represented by the structure visible on Fig. 9.5. Values x_i for $i = 1, \ldots, n$ are its inputs, w_i's are their weights and $\sigma(\cdot)$ is the so-called sigmoid function, that returns values close to zero when the neuron is "not triggered" and close to one if it is "triggered". As the sigmoid function, we can use, in particular:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}\,, \tag{9.5}$$

hyperbolic tangent or arctan can also be used. It is worth noting that in the original works of McCulloch and Pitts [78], the Heaviside step has been used, instead of the sigmoid function:

$$\mathscr{H}(z) = \begin{cases} 1 \text{ for } z \geq 0, \\ 0 \text{ for } z < 0. \end{cases}\,, \tag{9.6}$$

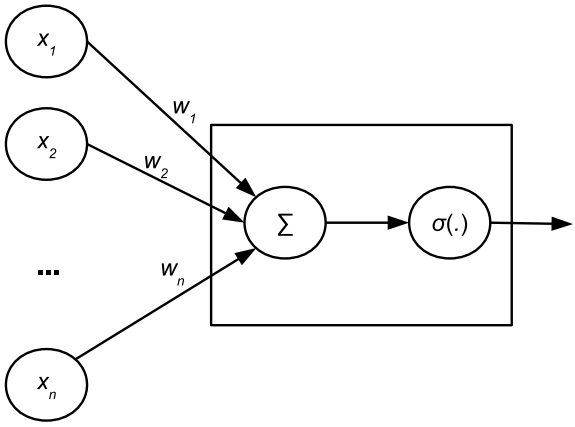

Fig. 9.5 A single neuron model

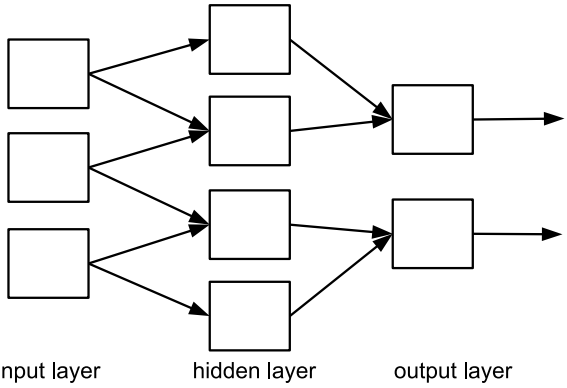

Fig. 9.6 An artificial neural network

Yet, as the Heaviside step is non-differentiable (at least in the space of "proper" functions), the smooth alternatives described above are more commonly used, today.

So, the artificial neuron works as the function, transferring its inputs as follows:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i\right). \tag{9.7}$$

Such neurons are grouped in a few *layers*; each of them processes the data generated by earlier layers (see Fig. 9.6). Traditionally, just two layers have been used: the input layer and the output layer; recently *deep believe networks*, with several hidden layers, become more and more popular. The idea of DBN is that the network creates some auxiliary "notions" to produce the final decision; a good overview of deep learning techniques (not restricted to ANNs) can be found in [96].

Training a neural network (teaching it to respond with desired outputs to proper inputs) is, in its essence, a specific kind of parameter estimation: we have a set of input-output pairs—precise (x_i, y_i) or interval-valued ones $(\mathbf{x}_i, \mathbf{y}_i)$—and we want to determine the weights. It can be done, as in Sect. 9.3.1, either by solving a CSP or an optimization problem. The former approach can be found, e.g., in [3], while [95] focuses on the latter one. Actually, using global optimization to train a neural network seems more popular, as non-interval researchers also tend to use optimization for parameter estimation. Some use other global optimization techniques, as simulated annealing or genetic algorithms. Non-global techniques, like the steepest descent iteration or quasi-Newton methods have also been applied in ANNs training.

The interval approach seems a promising alternative and their acceptance increases. Both interval approaches—CSP and global optimization—are considered in [7], for a three-layer network.

The network presented on Fig. 9.6 does not contain feedback connections: all information is transmitted "in one way": from the input layer, through the hidden one(s) to the output; there are no "backward" connections to earlier layers. Such non-recurrent ANNs are commonly (and successfully) used in practice, e.g., in clas-

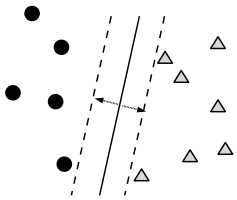

Fig. 9.7 The "margin" between linearly separate sets—maximized by SVMs

sification [73, 108], image recognition [109] or simply approximating multivariate nonlinear functions.

Nevertheless, in some applications (like prediction of a time series or other issues related to dynamical systems), we need the neural network to remember its previous states—and this can be achieved by using the feedback connections. It seems reasonable to assume, interval algorithms can be used for training ANNs with feedback as well, as for these without it. Nevertheless, the author is not aware of any actual research performed in this area.

9.4.2 Support Vector Machines

Neural networks are a powerful, but complicated AI tool. For simple classification problems, they are often replaced by tools less sophisticated, but simpler and—which is the most important—easier to train. Support Vector Machines (SVM) an example of such tools. As example applications, let us present [11, 70–72, 116].

In its principle, an SVM is roughly equivalent to a single neuron of an ANN. We have two sets of points: belonging and non-belonging to the approximated set, and we seek the hyperplane separating these sets of points. The "margin" (Fig. 9.7) between the sets should be maximized.

This decision problem can be formulated in terms of optimization:

$$\begin{aligned} &\min_{w,b} ||w|| \\ &\text{s.t.} \\ &y_k \cdot (w^T x_k - b) \geq 1 \;\; \forall k = 1, \ldots, n. \end{aligned} \tag{9.8}$$

Does (9.8) always have a solution? Not necessarily, of course. The feasible set may be empty—this is when the sets of elements belonging and non-belonging to a notion, are not linearly separable.

In such cases, the so-called *kernel trick* can be applied: we project the points (x_k, y_k) to another, higher-dimensional, space, to make them more likely to be linearly separable.

This results in the following optimization problem:

$$\begin{aligned} &\min_{w,b} ||w|| \\ &\text{s.t.} \\ &y_k \cdot (w^T \phi(x_k) - b) \geq 1 \quad \forall k = 1, \ldots, n \,. \end{aligned} \tag{9.9}$$

Most common kernel functions can be found, i.a., in the "Practical guide to SVM classification" [12].

In some situations, we are aware that our data contain outliers (cf. Sect. 9.3.3) and the separating hyperplane does not have to precisely separate all of the points. In such cases, some of the constraints may not be satisfied at the solution point. To obtain such a solution, we introduce additional *slack variables* ζ_k for each constraint $k = 1, \ldots, n$. The importance of minimizing the slack variables will be described by an additional parameter C.

Thus, we obtain a modified optimization problem for training the SVM:

$$\begin{aligned} &\min_{w,b} \left(w^T w + C \cdot \sum_{k=1}^{n} \zeta_k\right) \\ &\text{s.t.} \\ &y_k \cdot (w^T \phi(x_k) - b) \geq 1 - \zeta_k, \quad \text{for } k = 1, \ldots, n \,. \end{aligned} \tag{9.10}$$

The optimization problems we need to solve for training an SVM, specifically problems (9.8)–(9.10), presented above, are usually convex (but not for all kernels!). This is actually the greatest advantage of SVMs with respect to ANNs (where the optimization problem is certainly nonconvex) and also the reason why interval methods are relatively rarely used to parameterize this tool.

However, they become virtually irreplaceable in the case training pairs are interval-valued: $(\mathbf{x}_i, \mathbf{y}_i)$, instead of precise ones: (x_i, y_i). Such a situation has been considered, i.a., in [93] or [111].

9.5 Control Theory

Robust control is another field where interval methods find a natural application. Control systems should be designed so that some properties (which usually means stability, but also observability, etc.) of the closed-loop system are fulfilled, even if the parameters deviate from their nominal values. Checking many of these conditions reduces to solving a CSP or an equations system that either has to be fulfilled or cannot be fulfilled.

9.5.1 Stability Checking

For instance, let us consider a continuous-time linear system:

$$\dot{x}(t) = A \cdot x(t) + B \cdot u(t) ,$$
$$y(t) = C \cdot x(t) .$$

Assume we do not know exact values of the matrix parameters A, B, C, but only their bounds: $A \in \mathbf{A}$, $B \in \mathbf{B}$, $C \in \mathbf{C}$.

Such a system has the property of *asymptotic stability* when all eigenvalues of A have negative real parts (see, e.g., Chap. 7 of [45]). Such a condition would be relatively easy to verify using interval methods, if we had a good algorithm to bound the eigenvalues of an interval matrix $\mathbf{A}$. Unfortunately, the solution of such problem is not straightforward; see the discussion in Sect. 3.1 of [61].

A better approach is to use the *characteristic polynomial* of the matrix $A \in \mathbf{A}$. The Routh-Hurwitz criterion allows us to check the stability of this polynomial, by solving a system of inequalities, hence a CSP. Details can be found, in particular, in [85] or in Sect. 7.2 of [45].

An analog for a discrete-time system:

$$x(k+1) = A \cdot x(k) + B \cdot u(k) ,$$
$$y(k) = C \cdot x(k) ,$$

is the Schur criterion, verifying that all roots of the characteristic polynomial of $A \in \mathbf{A}$ (and hence all eigenvalues of the matrix) lie in the unit circle. As in the continuous case, interval methods can be applied for verification of this condition [112].

9.5.1.1 Robust Control

Robust control problems are often formulated not in terms of uncertain matrices, as in the previous subsection, but in terms of other uncertain parameters:

$$\dot{x}(t) = A(p) \cdot x(t) + B(p) \cdot u(t) ,$$
$$y(t) = C(p) \cdot x(t) ,$$

where $p \in [\underline{p}, \overline{p}]$.

In this case, the characteristic polynomial might have a more complicated structure (its coefficients will not be independent). Section 7.3 of [45] considers several techniques and theorems to deal with such problems; a good survey can also be found in [112].

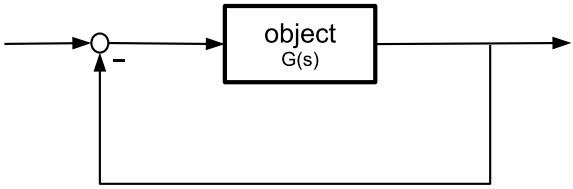

Fig. 9.8 A system with negative feedback—closed loop control

In general, affine arithmetic and other techniques for parametric linear systems can be applied there: [34, 103, 104]. Modal interval arithmetic methods are also very useful for robust control problems [113]; yet, the use of these kind of methods is beyond the scope of this monograph.

9.5.1.2 Frequency Methods

Analyzing dynamical systems in the time domain can be cumbersome, especially when the responses are slowly decaying. In such cases, it is better to analyze the system in its frequency domain. There are several such methods, typically applying the Fourier transform. These methods include spectral analysis, Bode plots and Nyquist plots.

This last technique is devoted to analyzing stability of systems with the negative feedback, presented on Fig. 9.8.

Such systems tend to have better characteristics than open-loop ones; in particular, they are less sensitive to parameter variations.

How to analyze the stability of closed-loop systems? Thanks to the Nyquist criterion, the problem reduces to analyzing the open-loop system. If the open-loop system has the transfer function $G(s)$, then this function for the closed-loop system would be as follows:

$$G_c(s) = \frac{G(s)}{1 + G(s)} \,. \tag{9.11}$$

Assume, the open-loop system is stable, i.e., $G(s)$ does not have poles in the real half-plane. Then it suffices to check, if $\big(1 + G(s)\big)$ does not have zeros, i.e., $G(j\omega \neq -1$ for $\omega \in [0, +\infty]$. This boils down, again, to solving a system of equations; precisely: verify that such system does *not* have solutions.

9.5.2 Designing a Controller

When designing a controller (Fig. 9.9)—either of type PID, or another one (H_∞, fuzzy, MPC, neural…)—we need to choose its parameters, so that given requirements are met. Such design problems are often formulated as optimization problems: we tend to maximize the stability degree or other performance measure; see, e.g., Section 7.5 of [45].

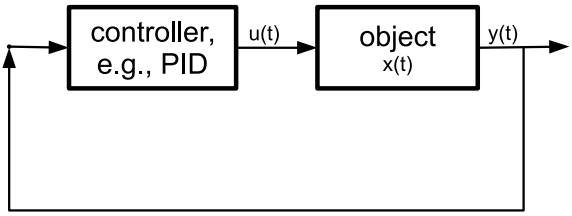

Fig. 9.9 Control system

Global optimization (cf. Sect. 6.1) can be applied here or linear programming with interval coefficients, as in [84].

Hence in [69], we consider bicriteria design of a PI controller, with the transfer function:

$$R(s) = k \cdot \left(1 + \frac{1}{Ts}\right). \tag{9.12}$$

The criteria are: the integral square error and the total change of the unit response.

Multicriteria analysis methods can be applied to choose the best parameters of the controller. In [69], the whole Pareto frontier (and Pareto set) has been computed. Other approaches, like goal programming or TOPSIS [98] are applicable, as well.

9.5.3 H_∞-Control

The H_∞-control is one of modern approaches to designing controllers. The idea is to choose such a controller that minimizes some quality measure, specifically the H_∞ norm of some matrix.

Thus, the synthesis problem reduces to seeking the global optimum of a function over the feasible set. Such an optimization problem can, obviously, be solved using interval methods (cf. Sect. 6.1). A specialized algorithm has been proposed in [85]. In [115] it is noted that such algorithm is more stable and less sensitive to numerical errors than its alternatives.

A similar optimization can be performed over Hardy spaces other than H_∞, e.g., over H_2. There are also mixed H_2/H_∞ approaches [112].

9.5.4 Model-Predictive-Control

Model-Predictive Control (MPC) is another popular advanced control technique, often preferred to PID, because of its robustness and simplicity in taking control constraints into account. In recent years, the interest grows in applying interval methods to compute MPC.

There are several variants of this approach. We can use it in continuous or discrete-time systems, we can use a single prediction or a bunch of predictions, we can associate probabilities with various predictions or use the minimax approach. In any case, we tend to optimize the predicted behavior of the controlled system, in a given time interval. Hence, finding the control for the MPC problem boils down to solving a nonlinear optimization problem.

Because of that, interval methods are a natural tool to be used. Indeed, several researchers have contributed to develop proper algorithms. In particular, papers of Rauh and his collaborators are remarkable in this area; see, e.g., [91, 92] and the references therein.

Also the author contributed to the field, proposing to use total derivatives [59] in MPC problem description.

As for nearly-linear systems, simpler control methods (like PID) suffice, the most important purpose of MPC is often to drive the system to the vicinity of some stable region. In this region, we can linearize the system and apply some simple control method. Such use of MPC is called the *dual-mode MPC* and interval methods are useful there, as well—see [17].

9.6 Nonlinear Dynamics, Chaos and Differential Equations

Also in other problems related to dynamical systems, interval analysis can become handy. Solving ordinary and partial differential equations, integral equations and other variations have been extensively studied by several authors. Particularly, let us note several contributions of Berz and Makino, e.g., [35, 77].

Proving the chaotic behavior of some dynamical systems is worth noting as one of the significant achievements of the interval community. Several papers can be quoted here, in particular many papers of Tibor Csendes and his collaborators: [14, 15]. In 2014 a Moore's prize has been given to Balázs Bánhelyi, Tibor Csendes, Tibor Krisztin and Arnold Neumaier, for verifying the Wright's conjecture on a delay differential equation [4].

Among other notable papers, let us list, i.a., the following ones: [25, 39–41, 86, 114].

The above list of papers on using interval methods for differential equations is far from being complete (or even fair), but a more complete description of contributions to ODE and PDE is absolutely beyond the scope of this Chapter; an interested reader can easily find several other papers and books on these topics.

9.7 Economical Modeling and Multiagent Systems

Economical modeling differs to the high extent from robot modeling, described in Sect. 9.2. Although, the models are nonlinear, also, they are far less precise than for distance constraints in kinematic problems. Various models of price sensitivity [65] are a good example.

Consequently, in solving economical problems, we are often not interested in precise localization, e.g., of the global optimum (or another precisely-defined solution), but in finding an acceptable solution, which can be obtained by less computationally-demanding techniques. On the other hand, problems under solutions are often of very high dimensionalities (thousands and more of variables), which is far beyond the possibilities of today interval B&BT algorithms (cf., e.g., [65]).

Does it mean interval algorithms will not be useful for economical applications? No, because of the following reasons:

- Although often we can settle with crude approximations of the solution (obtained, basing on a very crude and inaccurate model), there are some applications, where finding, e.g., the global maximum (or minimum) is crucial; such an application is described in Sect. 2 of [13]: risk management requires robust approximation of VaR (Value-at-Risk) or another risk measure.
- Imprecision of economical models can often be described as interval uncertainty. This approach is applied, e.g., in [19].

9.7.1 Economy Modeling

In this subsection, let us consider three economical modeling issues, where interval methods have found useful applications.

9.7.1.1 Game-Theoretic Models

In [68], the author has described a game-theoretical model of the economy, basing on a simple Keynesian theory. Using the algorithms described in Sect. 6.3, we localize the Nash equilibrium of the game between the following players: the monopolistic trade union, the government and the central bank. Their decision variables are: the nominal wage W, budget deficit B and the supply of money M, respectively. Companies are not considered as a player in this model, as they are assumed to be perfectly rational and choose decisions that maximize their profits; hence they can be eliminated from the model (cf. [68] and the references therein).

Decision problems for all three players can be formulated as follows.
For the trade union:

$$\begin{aligned}\text{find } w^o = \arg\max_{0\le w\le e} TU = (1-\gamma)\cdot\Big(w - m - \frac{B}{\exp(m)} + y_c\Big) + \\ + \alpha_1 \cdot \min\Big(\gamma\cdot\big(m + \frac{B}{\exp(m)} - w\big) + (1-\gamma)\cdot y_c - y_{TU}, 0\Big) + \\ - \alpha_2 \cdot \max\Big(\exp\big((1-\gamma)\cdot\big(m + \frac{B}{\exp(m)}\big) + \gamma w - (1-\gamma)\cdot y_c\big) + \\ - 1 - \Pi_{TU}, 0\Big)\,.\end{aligned} \tag{9.13}$$

For the government:

$$\begin{aligned}\text{find } B^o = \arg\max_{0\le B\le B_m} G = \beta_1 \cdot \min(B - B_G, 0) + \\ + \beta_2 \cdot \min\Big(\gamma\cdot\big(m + \frac{B}{\exp(m)} - w\big) + (1-\gamma)\cdot y_c - y_G, 0\Big) + \\ - \beta_3 \cdot \max\Big(\exp\big((1-\gamma)\cdot(m + \frac{B}{\exp(m)}) + \gamma w - (1-\gamma)\cdot y_c\big) + \\ - 1 - \Pi_G, 0\Big)\,.\end{aligned} \tag{9.14}$$

And for the central bank:

$$\begin{aligned}\text{find } m = \arg\max_{m^-\le m\le m^+} CBS(m) = \\ - \max\Big(\exp\big((1-\gamma)\cdot(m + \frac{B^N(m)}{\exp(m)}) + \\ + \gamma w^N(m) - (1-\gamma)\cdot y_c\big) - 1 - \Pi_{CB}, 0\Big) + \\ + \delta \cdot \min\Big(\gamma\cdot\big(m + \frac{B^N(m)}{\exp(m)} - w^N(m)\big) + (1-\gamma)\cdot y_c - y_{CB}, 0\Big)\,.\end{aligned} \tag{9.15}$$

Values w, b and m are logarithms of W, B and M. Again, details have been explained in [68].

Interval B&BT methods allowed to enclose the solution of this game—at least for this simple model.

9.7.1.2 Input-Output Models

Another important economical model is the *Leontief input-output model*, representing interdependencies between various branches of an economy. The model of Wassily Leontief has earned a Nobel prize (in 1973) and is currently used by several countries for planning.

Assume we have N goods and the demand for each of them by the end consumers (households) is d_i for $i = 1, \ldots, N$. But to produce a unit of the j-th good, we need a_{ij} amounts of i-ths goods for $i = 1, \ldots, N$ and $j \neq i$.

This means, we need to produce the following amount of the i-ths good in our economy:

$$x_i = \sum_{j=1}^{N} a_{ij} x_j + d_i \, . \tag{9.16}$$

This results in a system of equations, that can be presented in the matrix form:

$$x = A \cdot x + d \, . \tag{9.17}$$

The obvious problem is that the parameters a_{ij} and d_i are hard to determine and prone to various kinds of uncertainty. Hence, it is useful to represent them not as precise numbers, but either intervals (see, e.g., [94], Chap. 6 of [51]) or fuzzy numbers (Sect. 7.2 of [20]).

For interval linear systems, various methods (and various solution concepts!) have been proposed; we have already discussed some of them—cf., e.g., [36, 87, 97, 99, 100, 104] and many others.

What about fuzzy linear systems? Actually, a fuzzy number can be represented as a collection of nested intervals; see Fig. 9.10 and Chap. 11 of [51].

So, the interval calculus can be applied to fuzzy numbers, as well.

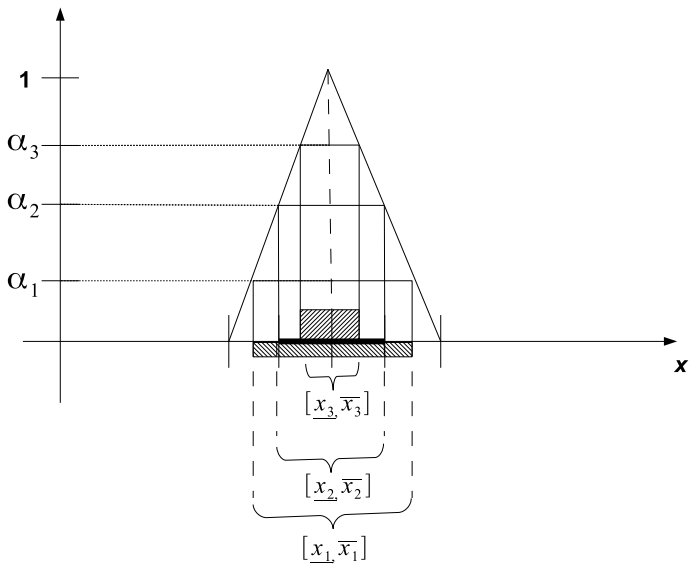

Fig. 9.10 Fuzzy set as a collection of nested intervals

9.7.1.3 Net Present Value and Internal Rate of Return

The Net Present Value (NPV) is a common quality measure of financial projects [20]. It is usually computed as:

$$NPV = \sum_{t=1}^{T} \frac{P_t}{(1+d)^t} - KV \ , \tag{9.18}$$

where KV is the initial capital to be invested and P_t is the total income in year $t = 1, \ldots, T$. Obviously, d is the discount rate.

The value of d for which KV is at least returned by the income is called the Internal Rate of Return (IRR); so, it is the solution of the following nonlinear equation:

$$\sum_{t=1}^{T} \frac{P_t}{(1+d)^t} = KV \ .$$

As for the Leontief equations system, the parameters are severely uncertain in all real-life situations. While KV may be considered known (although, in [20] it is considered fuzzy, also), P_t's are definitely vague.

All representations of uncertainty can be applied to capture this imprecision: probabilistic, set-theoretic and fuzzy models. As in previous paragraphs, the interval calculus is useful to perform the computations. The concept of *interval extended zero* (cf. [18, 19, 97]) seems particularly useful, in such applications.

However, it is worth noting that only relatively simple cases can be solved in practice. Providing useful estimates for real-world data still remains an open problem [20].

9.7.1.4 Value at Risk

The Value at Risk (VaR) is another widely accepted tool in financial decision making. It is a measure of the risk of a project, defined as the possible loss of the project (under certain assumptions).

Computing this value, even provided all the assumptions are proper, may be a hard task. In [13], Kearfott describes this issue for the currency trading problem by a bank. It turns out, an optimum of a nonconvex function has to be localized to properly estimate the VaR. This has been one of the applications of the GlobSol solver [1].

9.7.2 Queueing Systems

This is another natural source of nonlinear problems. Various queueing systems and networks have features (mean sojourn time, mean busy period length, etc.) that are nonlinear functions of some parameters.

For instance, in the case of an $M/M/1$ queueing system, i.e., a system with exponential (M = "memoryless") interarrival and service time and a single service channel, we have the following formulae for the average sojourn time (by which we understand the time the task spends in the system: waiting in the queue and then being serviced).

In the case of an infinite buffer for the queue, we have:

$$\mathbb{E}\,S = \frac{1}{\mu - \lambda}\,, \tag{9.19}$$

while for a limited buffer of size $n-1$:

$$\mathbb{E}\,S = \frac{1}{\mu - \lambda} - \frac{(n+1)\cdot\left(\frac{\lambda}{\mu}\right)^{n+1}}{\mu\cdot\left(1-\left(\frac{\lambda}{\mu}\right)^{n+1}\right)}\,. \tag{9.20}$$

In both cases, λ is the arrival rate (parameter of the exponential distribution of the interarrival time; $\frac{1}{\lambda}$ is the mean of this time) and μ—the service rate.

In the case of an $M/G/1$ queue, i.e., the one with non-exponential service time, the formula is more sophisticated:

$$\mathbb{E}\,S = \mathbb{E}\,B + \frac{\lambda\cdot\mathbb{E}(B^2)}{2\cdot(1-\lambda\cdot\mathbb{E}\,B)}\,, \tag{9.21}$$

where B is the service time. As we can see from Formula (9.21), the mean value of the sojourn time depends not only on the mean value of the service time, but also on its variance (the second moment). For details and explanation of these interesting phenomena of probability theory, the reader can consult several textbooks, e.g., [32, 107].

Attempting to optimize such measures results in global optimization problems [62–64]. It is worth noting that, for some service time distributions, particularly long-tailed ones (like the Pareto distribution), the expected value (9.21) may be infinite. In such cases, another performance measure is necessary. It can, in particular, be based on the Laplace transform of the PDF of the service time [63].

Interval methods are well suited to bound the values of performance measures—presented by above formulae as well, as other ones. They get even more useful, if some parameters of these distributions are not precisely known, which is a common situation. Paper [76] surveys several such situations. In [117], an application to modeling the performance of SunRPC systems is discussed.

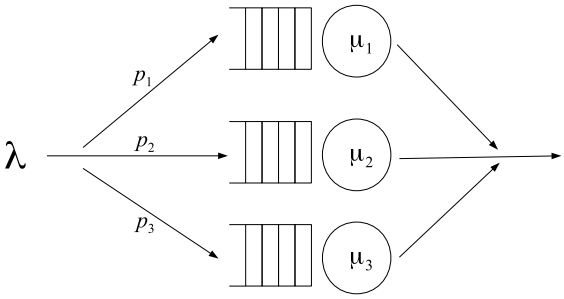

Fig. 9.11 Open queueing network example

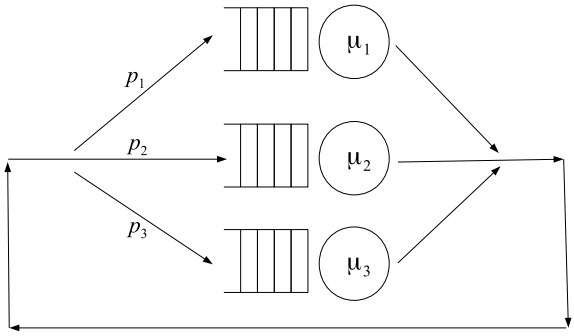

Fig. 9.12 Closed queueing network example

In the latter paper (and many other ones), a computer system is modeled as a *queueing network*: open or closed one. In the first case, there is an external source of tasks entering the system (customers, packets, programs or whatever the system is modeling).

Closed queueing networks are harder to analyze as the arrival rate to the system is not a parameter, but has to be deduced from the structure of the system. In closed networks, the number of tasks that circulate in the system is *fixed*. Clients requesting some services can be represented by additional queueing systems with a single task that is send to the server with some probability distribution, representing the likelihood of the service being requested by this client (Figs. 9.11 and 9.12).

An approach to analyze closed queueing network is the meanvalue analysis (MVA; see, e.g., [32]).

In [74], MVA is performed for a closed queueing network with uncertain parameters. The interval algorithm described there is well-tuned, making use of monotonicity of some functions and other specific features. Another approach is described in [75].

In a more general setting, the parameters may not be interval-valued, but fuzzy or described by yet another uncertainty representation (cf. Sect. 9.7.3.1). In [90], it is claimed that while the uncertainty related to arrival times is usually of probabilistic nature, vagueness of service times is better described by fuzzy or possibilistic quantities. This is the case when we have only superficial knowledge about scheduling policies and the servicing process on the server machine. In the author's opinion, the interarrival time's uncertainty may also be of non-probabilistic nature. This would

be the case when the users choose the server they want to use basing on some criteria that are only broadly known to us. Such a model would be an obvious extension of the one described in [57].

Queueing systems are usually modeled via Markov chains [107]. Other model is diffusive approximation, used in the case of so-called heavy-traffic: we assume there a continuum of tasks processed at service stations, instead of a discrete set of tasks. Such a model is considered, i.a., in [53]. There are also other approaches to solve queueing systems. Using interval methods, we can attempt to directly solve the Lindley's equation, using the Wiener-Hopf factorization [33]; details are beyond the scope of this survey.

Markov chains are typically considered to have a precisely-known transition matrix; yet, in real-life situations this assumption should often be relaxed. In particular, several authors have considered Markov chains with interval-valued transition probabilities. Two versions of this problem should be distinguished:

- the Markov process is stationary: its transition matrix is invariant, just not precisely known; such a situation is considered, i.a., in [54];
- the Markov process may be non-stationary and its transition matrix may vary; yet, we know bounds on its changes [105].

Up to now, relatively few studies on these topics have been published. A possible reason is that few researchers are skilled in both areas: queueing theory and interval methods. Hopefully, the situation will change in the future, as modeling computer systems (and many other systems well-described by queueing models) has several important applications, and interval methods are being used more and more frequently.

9.7.3 Decision Making

Most problems considered in this monograph can be classified as "decision-making problems", namely almost all problems of type (1.1). Yet, in this subsection, we shall consider more practical applications.

When facing the need of choosing an option out of a few alternatives or from a continuum of possible values, we need to deal with two problems: uncertainty and multicriteriality.

9.7.3.1 Uncertainty Representation

The uncertainty we face for most real-world problems, has several sources. Numerical computations are inaccurate, our measurements are imprecise, our models are only some approximations of actual phenomenons, we lack the knowledge of several things (and of decisions of other decision-makers); even our own goals (and sometimes also our limitations) are not quite strict.

Uncertainty modeling is a broad research field. Traditional tools and theories used to describe uncertainty are:

- probability theory,
- set theory,
- fuzzy set theory and fuzzy logic.

Some more modern theories, include:

- various kinds of imprecise probability theory (see, e.g., [6, 23, 57, 63], Chap. 10 of [51]),
- intuitionistic fuzzy sets—these of Atanassov and other versions (cf. Sect. 3.1 of [20]),
- so-called type-2 and type-n fuzzy sets (cf. Chap. 3 of [20]),
- possibility theory of Dubois and Prade,
- Dempster-Shafer theory (cf. [21], Sect. 3.3 of [20]),
- info-gap theory of Ben-Haim [8]; see also [88, 110].

and many other approaches. In an interesting paper [56], a few of the above uncertainty models are applied to decide whether to use cloud storage or not.

The rationale behind all these models is far beyond the scope of this monograph. What is important is that interval methods are pretty useful in processing several of them.

It is important not to identify the interval calculus with any of the uncertainty representations. Even for the set-theoretic uncertainty, it might not be necessary to apply the interval analysis; the Hurwicz approach, combining the best and worst case with some coefficients [37, 66] or other approaches (like in [118] or [38]) can be sufficient.

Nevertheless, in many situations, intervals may become very handy:

- set-theoretic uncertainty can be bound using (a collection of) intervals to investigate the worst-case scenario, using optimization methods (possibly of Sect. 6.1),
- also for the info-gap theory, the interval calculus can be applied to analyze the possible values of the unknown parameters and find the *robustness* and *oportunness* of the uncertainty model, in the range of the horizon of uncertainty,
- fuzzy sets (and their various extensions) can be represented as a collection of α-cuts, which again may be bound using intervals (cf. Chaps. 11–13 of [51],
- also for the Dempster-Shafer theory and various imprecise probability theories, the application of interval methods is straightforward.

Particular interest should be devoted solving multistage decision problems in presence of various kinds of uncertainty. A well-established approach (or, more precisely: class of approaches) to solve such problems is *dynamic programming* (DP; see, e.g., [49] and the references therein), efficient for some, but definitely not all cases [48]. Alternatives to DP are some randomized algorithms [48] or methods similar to B&BT [46, 47].

In [50] an application to controlling regional development is described. Both objective and subjective goals and constraints are considered in this paper and they are modeled using fuzzy quantities.

As it has already been stated, interval methods can help in solving such problems on various levels: at least, in the branch-and-bound process, in the implementation of fuzzy arithmetic itself, and, finally, in aggregating various criteria. This last topic is going to be described in the next paragraph.

9.7.3.2 Multicriteria Decision Making (MCDM)

In Sect. 6.2, we considered the problem of approximating the Pareto frontier (and Pareto set) of a multicriteria decision problem. But even enclosing the whole efficient set does not solve the issue of choosing a single decision, that is going to get selected—the one "subjectively" optimal for the decision-maker. To obtain this goal, we need to "aggregate" the criteria somehow; we may or may not approximate the whole frontier for that.

A few approaches have been proposed for MCDM: utility functions—linear or nonlinear ones (aggregating all criteria with some weights), goal programming, TOPSIS, AHP…

For several of these approaches, the preferences of the decision-maker are not completely precise. Fuzzy (and intuitionistic fuzzy) representation of goals or reference points seems quite natural. Interval methods can be applied in such situations, in a manner analogous to the ones described previous paragraphs.

For details, the interested reader should consult [20] and the references therein. Heuristics for comparison of intervals, fuzzy intervals, etc. (cf. Sect. 2.6) are particularly useful, in this application.

Interval-valued reference points have also been considered (cf., e.g., [98] and the references therein).

9.8 Summary

As we have seen, while the interval approach is still underused, the area of its applications continuously increases. Potential usage in various branches of science and engineering are, as it has been shown, very broad. We hope for increasing adoption of the interval calculus in these areas—and we look forward to it. The advent of multicore and manycore architectures, as well as other massively parallel ones, should promote this approach, as the interval B&BT algorithms are well suited for parallel implementations (cf. Chap. 7).

References

1. GlobSol solver (2015). https://interval.louisiana.edu/GlobSol/
2. Polynomial nonlinear system benchmarks (2017). https://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/node1.html
3. Adam, S.P., Karras, D.A., Magoulas, G.D., Vrahatis, M.N.: Solving the linear interval tolerance problem for weight initialization of neural networks. Neural Netw. **54**, 17–37 (2014)
4. Bánhelyi, B., Csendes, T., Krisztin, T., Neumaier, A.: Global attractivity of the zero solution for Wright's equation. SIAM J. Appl. Dyn. Syst. **13**(1), 537–563 (2014)
5. Bars, F.L., Bertholom, A., Sliwka, J., Jaulin, L.: Interval SLAM for underwater robots; a new experiment. In: NOLCOS 2010 (2010)
6. Beer, M., Ferson, S., Kreinovich, V.: Imprecise probabilities in engineering analyses. Mech. Syst. Signal Process. **37**(1–2), 4–29 (2013)
7. Beheshti, M., Berrached, A., de Korvin, A., Hu, C., Sirisaengtaksin, O.: On interval weighted three-layer neural networks. In: Proceedings of the 31st Annual on Simulation Symposium, 1998, pp. 188–194. IEEE (1998)
8. Ben-Haim, Y.: Info-gap Decision Theory: Decisions Under Severe Uncertainty, 2nd edn. Academic Press (2006)
9. Berleant, D., Xie, L., Zhang, J.: Statool: a tool for distribution envelope determination (DEnv), an interval-based algorithm for arithmetic on random variables. Reliab. Comput. **9**(2), 91–108 (2003)
10. Berleant, D., Zhang, J.: Using Pearson correlation to improve envelopes around the distributions of functions. Reliab. Comput. **10**(2), 139–161 (2004)
11. Bilski, A.: A review of artificial intelligence algorithms in document classification. Int. J. Electron. Telecommun. **57**(3), 263–270 (2011)
12. Chang, C.C., Lin, C.J.: LIBSVM–A Library for Support Vector Machines (2016). https://www.csie.ntu.edu.tw/~cjlin/libsvm/
13. Corliss, G.F., Kearfott, R.B.: Rigorous global search: Industrial applications. In: Developments in Reliable Computing, pp. 1–16. Springer (1999)
14. Csendes, T., Bánhelyi, B., Garay, B.: A global optimization model for locating chaos. In: International Workshop on Global Optimization, pp. 81–84 (2005)
15. Csendes, T., Bánhelyi, B., Hatvani, L.: Towards a computer-assisted proof for chaos in a forced damped pendulum equation. J. Comput. Appl. Math. **199**(2), 378–383 (2007)
16. Desrochers, B., Lacroix, S., Jaulin, L.: Set-membership approach to the kidnapped robot problem. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3715–3720. IEEE (2015)
17. Dombrovskii, V.V., Chausova, E.V.: Model predictive control for linear systems with interval and stochastic uncertainties. Reliab. Comput. **19**(4), 351–360 (2014)
18. Dymova, L., Pilarek, M., Wyrzykowski, R.: Solving systems of interval linear equations with use of modified interval division procedure. In: PPAM 2009 Proceedings (2010)
19. Dymova, L., Sevastjanov, P., Pilarek, M.: A method for solving systems of linear interval equations applied to the Leontief input-output model of economics. Expert Syst. Appl. **40**(1), 222–230 (2013)
20. Dymowa, L.: Soft Computing in Economics and Finance. Springer (2011)
21. Fedrizzi, M., Kacprzyk, J., Yager, R.R. (eds.): Advances in the Dempster-Shafer Theory of Evidence (1994)
22. Ferson, S., Kreinovich, V., Aviles, M.: Exact bounds on sample variance of interval data. In: Extended Abstracts of the 2002 SIAM Workshop on Validated Computing, Toronto, pp. 67–69 (2002)
23. Ferson, S., Kreinovich, V., Ginzburg, L., Myers, D.S., Sentz, K.: Constructing probability boxes and Dempster-Shafer structures. Technical Report (2003)
24. Frese, U.: A discussion of simultaneous localization and mapping. Auton. Robot. **20**(1), 25–42 (2006)

25. Gajda, K., Jankowska, M., Marciniak, A., Szyszka, B.: A survey of interval Runge–Kutta and multistep methods for solving the initial value problem. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, 1361–1371 (2009)
26. Gutowski, M.W.: Interval straight line fitting (2001). arXiv:math/0108163
27. Gutowski, M.W.: Introduction to Interval Calculi and Methods (in Polish). BEL Studio, Warszawa (2004)
28. Gutowski, M.W.: Interval experimental data fitting. In: Liu, J. (ed.) Focus on Numerical Analysis, pp. 27–70. Nova Science Publishers, New York (2006)
29. Gutowski, M.W.: Breakthrough in interval data fitting I. The role of Hausdorff distance (2009). arXiv:0903.0188
30. Gutowski, M.W.: Breakthrough in interval data fitting II. From ranges to means and standard deviations (2009). arXiv:0903.0365
31. Hao, F., Merlet, J.P.: Multi-criteria optimal design of parallel manipulators based on interval analysis. Mech. Mach. Theory **40**(2), 157–171 (2005)
32. Harchol-Balter, M.: Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press (2013)
33. Haßlinger, G., Fausten, D.: Analysis of the workload in communication systems including data transfers over arbitrary time scales. Int. J. Simul. **3**(3–4), 25–37 (2002)
34. Hladík, M.: Enclosures for the solution set of parametric interval linear systems. Int. J. Appl. Math. Comput. Sci. **22**(3), 561–574 (2012)
35. Hoefkens, J., Berz, M., Makino, K.: Efficient high-order methods for ODEs and DAEs. In: Automatic Differentiation of Algorithms, pp. 343–348 (2002)
36. Horacek, J., Hladík, M.: Subsquares approach–a simple scheme for solving overdetermined interval linear systems. In: PPAM 2013 (10th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 8385, pp. 613–622 (2014)
37. Hurwicz, L.: Optimality criteria for decision making under ignorance. In: Cowles Commission Discussion Paper, Statistics, 370 (1951)
38. Huynh, V., Kreinovich, V., Nakamori, Y., Nguyen, H.T.: Towards efficient prediction of decisions under interval uncertainty. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, pp. 1372–1381 (2009)
39. Jankowska, M.A.: Remarks on algorithms implemented in some C++ libraries for floating-point conversions and interval arithmetic. In: PPAM 2009 Proceedings. Lecture Notes in Computer Science, vol. 6068, pp. 436–445 (2010)
40. Jankowska, M.A.: An interval backward finite difference method for solving the diffusion equation with the position dependent diffusion coefficient. In: PPAM 2011 Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 447–456 (2013)
41. Jankowska, M.A.: Interval finite difference method for solving the problem of bioheat transfer between blood vessel and tissue. In: PPAM 2013 Proceedings. Lecture Notes in Computer Science, vol. 8385, pp. 644–655 (2014)
42. Jaulin, L.: Path planning using intervals and graphs. Reliab. Comput. **7**(1), 1–15 (2001)
43. Jaulin, L.: Range-only SLAM with occupancy maps: a set-membership approach. IEEE Trans. Robot. **27**(5), 1004–1010 (2011)
44. Jaulin, L., Dabe, F., Bertholom, A., Legris, M.: A set approach to the simultaneous localization and map building-application to underwater robots. ICINCO-RA **2**, 65–69 (2007)
45. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
46. Kacprzyk, J.: A branch-and-bound algorithmm for the multistage control of a nonfuzzy system in a fuzzy environment. Control Cybern. **7**, 51–64 (1978)
47. Kacprzyk, J.: A branch-and-bound algorithmm for the multistage control of a fuzzy system in a fuzzy environment. Kybernetes **8**(2), 139–147 (1979)
48. Kacprzyk, J.: A genetic algorithm for the multistage control of a fuzzy system in a fuzzy environment. Mathw. Soft Comput. **4**(3), 219–232 (1997)

49. Kacprzyk, J., Esogbue, A.O.: Fuzzy dynamic programming: main developments and applications. Fuzzy Sets Syst. **81**(1), 31–45 (1996)
50. Kacprzyk, J., Straszak, A.: Application of fuzzy decision-making models for determining optimal policies in "stable" integrated regional development. In: Fuzzy Sets, pp. 321–328 (1980)
51. Kearfott, R.B., Kreinovich, V.: Applications of Interval Computations, vol. 3. Springer Science & Business Media (2013)
52. Kieffer, M., Jaulin, L., Walter, É., Meizel, D.: Robust autonomous robot localization using interval analysis. Reliab. Comput. **6**(3), 337–362 (2000)
53. Kingman, J.F.C.: The single server queue in heavy traffic. In: Mathematical Proceedings of the Cambridge Philosophical Society, vol. 57, pp. 902–904. Cambridge University Press (1961)
54. Kozine, I.O., Utkin, L.V.: Interval-valued finite Markov chains. Reliab. Comput. **8**(2), 97–113 (2002)
55. Kreinovich, V.: Statistical data processing under interval uncertainty: Algorithms and computational complexity. In: Soft Methods for Integrated Uncertainty Modelling, pp. 11–26. Springer (2006)
56. Kreinovich, V., Gallardo, E.: Optimizing cloud use under interval uncertainty. In: PPAM 2015 Proceedings. Lecture Notes in Computer Science, vol. 9574, pp. 435–444 (2016)
57. Kubica, B.J.: Estimating utility functions of network users-an algorithm using interval computations. Ann. Univ. Timisoara **40**, 121–134 (2002)
58. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. Numer. Algorithms **70**(4), 929–963 (2015). http://dx.doi.org/10.1007/s11075-015-9980-y
59. Kubica, B.J.: Preliminary experiments with an interval Model-Predictive-Control solver. In: PPAM 2015 Proceedings. Lecture Notes in Computer Science, vol. 9574, pp. 464–473 (2016)
60. Kubica, B.J.: Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. J. Parallel Distrib. Comput. **107**, 57–66 (2017). https://doi.org/10.1016/j.jpdc.2017.03.009
61. Kubica, B.J.: Advanced interval tools for computing solutions of continuous games. Vychislennyie Tiehnologii (Computational Technologies) **23**(1), 3–18 (2018)
62. Kubica, B.J., Malinowski, K.: An interval global optimization algorithm combining symbolic rewriting and componentwise Newton method applied to control a class of queueing systems. Reliab. Comput. **11**(5), 393–411 (2005)
63. Kubica, B.J., Malinowski, K.: Interval random variables and their application in queueing systems with long–tailed service times. In: Soft Methods for Integrated Uncertainty Modelling, pp. 393–403. Springer (2006)
64. Kubica, B.J., Malinowski, K.: Optimization of performance of queuing systems with long-tailed service times. Prace Naukowe Politechniki Warszawskiej. Elektronika **156**, 237–245 (2006)
65. Kubica, B.J., Niewiadomska-Szynkiewicz, E.: An improved interval global optimization algorithm and its application to price management problem. In: PARA 2006 Proceedings. Lecture Notes in Computer Science, vol. 4699, pp. 1055–1064 (2007)
66. Kubica, B.J., Pownuk, A., Kreinovich, V.: What decision to make in a conflict situation under interval uncertainty: efficient algorithms for the Hurwicz approach. In: PPAM 2017 Proceedings. Lecture Notes in Computer Science, vol. 10778, pp. 402–411 (2018)
67. Kubica, B.J., Szynkiewicz, W.: CuikSLAM with unknown correspondence-preliminary results. Prace Naukowe Politechniki Warszawskiej. Elektronika **160**, 143–151 (2007)
68. Kubica, B.J., Woźniak, A.: Applying an interval method for a four agent economy analysis. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 477–483 (2012)
69. Kubica, B.J., Woźniak, A.: Tuning the interval algorithm for seeking Pareto sets of multicriteria problems. In: PARA 2012 Proceedings. Lecture Notes in Computer Science, vol. 7782, pp. 504–517 (2013)

70. Kurek, J., Kruk, M., Osowski, S., Hoser, P., Wieczorek, G., Jegorowa, A., Górski, J., Wilkowski, J., Śmietańsk, K., Kossakowska, J.: Developing automatic recognition system of drill wear in standard laminated chipboard drilling process. Bull. Pol. Acad. Sci. Techn. Sci. **64**(3), 633–640 (2016)
71. Kurek, J., Osowski, S.: Support Vector Machine for diagnosis of the bars of cage inductance motor. In: 15th IEEE International Conference on Electronics, Circuits and Systems, 2008. ICECS 2008, pp. 1022–1025 (2008)
72. Kurek, J., Osowski, S.: Support vector machine for fault diagnosis of the broken rotor bars of squirrel-cage induction motor. Neural Comput. Appl. **19**(4), 557–564 (2010)
73. Kurek, J., Swiderski, B., Jegorova, A., Kruk, M., Osowski, S.: Deep learning in assessment of drill condition on the basis of images of drilled holes. In: Proceedings ICGIP 2016 (Eighth International Conference on Graphic and Image Processing), p. 10225 (2017)
74. Lüthi, J., Haring, G.: Mean value analysis for queueing network models with intervals as input parameters. Perform. Eval. **32**(3), 185–215 (1998)
75. Lüthi, J., Lladó, C.M.: Splitting techniques for interval parameters and their application to performance models. Perform. Eval. **51**(1), 47–74 (2003)
76. Majumdar, S.: Application of relational interval arithmetic to computer performance analysis: a survey. Constraints **2**(2), 215–235 (1997)
77. Makino, K., Berz, M.: Efficient control of the dependency problem based on Taylor model methods. Reliab. Comput. **5**(1), 3–12 (1999)
78. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1943)
79. Meizel, D., Preciado-Ruiz, A., Halbwachs, E.: Estimation of mobile robot localization: Geometric approaches. In: Milanese, M., Norton, J., Piet-Lahanier, H., Walter, É. (eds.) Bounding Approaches to System Identification, pp. 463–489. Springer (1996)
80. Merlet, J.P.: Solving the forward kinematics of a Gough-type parallel manipulator with interval analysis. Int. J. Robot. Res. **23**(3), 221–235 (2004)
81. Merlet, J.P.: Kinematics of the wire-driven parallel robot MARIONET using linear actuators. In: IEEE International Conference on Robotics and Automation, 2008. ICRA 2008, pp. 3857–3862. IEEE (2008)
82. Merlet, J.P.: Interval analysis for certified numerical solution of problems in robotics. Int. J. Appl. Math. Comput. Sci. **19**(3), 399–412 (2009)
83. Merlet, J.P., Gosselin, C.: Parallel mechanisms and robots. In: Springer Handbook of Robotics, pp. 269–285. Springer (2008)
84. Mitra, S., Keel, L., Bhattacharyya, S.: Data-robust design of PID controllers via interval linear programming. IFAC Proc. **40**(20), 632–636 (2007)
85. Monnet, D., Ninin, J., Clement, B.: Global optimization of H_∞ problems: Application to robust control synthesis under structural constraints. In: International Conference on Mathematical Aspects of Computer and Information Sciences, pp. 550–554. Springer (2015)
86. Nagatou, K.: A numerical method to verify the elliptic eigenvalue problems including a uniqueness property. Computing **63**(2), 109–130 (1999)
87. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press (1990)
88. Piegat, A., Tomaszewska, K.: Decision-making under uncertainty using info-gap theory and a new multidimensional RDM interval-arithmetic. Electr. Rev. **88**(8), 71–76 (2013)
89. Porta, J.M.: CuikSlam: A kinematics-based approach to SLAM. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation, 2005. ICRA 2005, pp. 2425–2431. IEEE (2005)
90. Prade, H.M.: An outline of fuzzy or possibilistic models for queuing systems. In: Fuzzy sets, pp. 147–153. Springer (1980)
91. Rauh, A., Hofer, E.P.: Interval methods for optimal control. In: Variational Analysis and Aerospace Engineering, pp. 397–418. Springer (2009)
92. Rauh, A., Senkel, L., Kersten, J., Aschemann, H.: Interval methods for sensitivity-based model-predictive control of solid oxide fuel cell systems. Reliab. Comput. **19**(4), 361–384 (2014)

93. Rodríguez, J.J., Alonso, C.J., Maestro, J.A.: Support vector machines of interval-based features for time series classification. Knowl. Based Syst. **18**(4–5), 171–178 (2005)
94. Rohn, J.: Input-output model with interval data. Econom. J. Econom. Soc. 767–769 (1980)
95. Saraev, P.V.: Numerical methods of interval analysis in learning neural network. Autom. Remote Control **73**(11), 1865–1876 (2012)
96. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
97. Sevastjanov, P., Dymova, L.: Fuzzy solution of interval linear equations. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, pp. 1392–1399 (2009)
98. Sevastjanov, P., Tikhonenko, A.: Direct interval extension of TOPSIS method. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 504–512 (2012)
99. Sharaya, I.A.: On maximal inner estimation of the solution sets of linear systems with interval parameters. Reliab. Comput. **7**(5), 409–424 (2001)
100. Shary, S.P.: Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of kaucher arithmetic. Reliab. Comput. **2**(1), 3–33 (1996)
101. Shary, S.P.: Finite-Dimensional Interval Analysis. Institute of Computational Technologies, SB RAS, Novosibirsk (2013)
102. Shary, S.P.: Strong compatibility in data fitting problem under interval data uncertainty. Comput. Technol. **22**(2), 150–172 (2017)
103. Skalna, I.: On checking the monotonicity of parametric interval solution of linear structural systems. In: PPAM 2007 Proceedings. Lecture Notes in Computer Science, vol. 4967, pp. 1400–1409 (2009)
104. Skalna, I., Hladík, M.: A new method for computing a p-solution to parametric interval linear systems with affine-linear and nonlinear dependencies. BIT Numer. Math. **57**(4), 1109–1136 (2017)
105. Škulj, D.: Finite discrete time Markov chains with interval probabilities. In: Soft Methods for Integrated Uncertainty Modelling, pp. 299–306. Springer (2006)
106. Sliwka, J., Bar, F.L., Reynet, O., Jaulin, L.: Using interval methods in the context of robust localization of underwater robots. In: 2011 Annual Meeting of the North American on Fuzzy Information Processing Society (NAFIPS), pp. 1–6. IEEE (2011)
107. Stewart, W.J.: Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling. Princeton University Press (2009)
108. Swiderski, B., Kurek, J., Osowski, S.: Multistage classification by using logistic regression and neural networks for assessment of financial condition of company. Decis. Support Syst. **52**(2), 539–547 (2012)
109. Swiderski, B., Osowski, S., Kurek, J., Kruk, M., Lugowska, I., Rutkowski, P., Barhoumi, W.: Novel methods of image description and ensemble of classifiers in application to mammogram analysis. Expert Syst. Appl. **81**, 67–78 (2017)
110. Tomaszewska, K., Piegat, A.: Uncertainty analysis for efficient fuel allocation using info-gap theory. Inf. Syst. Manag. **4**, (2015)
111. Utkin, L.V., Chekh, A.I.: A new robust model of one-class classification by interval-valued training data using the triangular kernel. Neural Netw. **69**, 99–110 (2015)
112. Vehí, J., Ferrer, I., Sainz, M.Á.: A survey of applications of interval analysis to robust control. IFAC Proc. **35**(1), 389–400 (2002)
113. Vehí, J., Rodellar, J., Sainz, M., Armengol, J.: Analysis of the robustness of predictive controllers via modal intervals. Reliab. Comput. **6**(3), 281–301 (2000)
114. Watanabe, Y., Nagatou, K., Plum, M., Nakao, M.T.: Verified computations of eigenvalue exclosures for eigenvalue problems in Hilbert spaces. SIAM J. Numer. Anal. **52**(2), 975–992 (2014)
115. Weinhofer, J.K., Haas, W.C.: H_∞-control using polynomial matrices and interval arithmetic. Reliab. Comput. **3**(3), 229–237 (1997)
116. Wilinski, A., Osowski, S., Siwek, K.: Gene selection for cancer classification through ensemble of methods. In: International Conference on Adaptive and Natural Computing Algorithms, pp. 507–516. Springer (2009)

117. Woodside, C.M., Majumdar, S., Neilson, J.E.: Interval arithmetic for computing performance guarantees in client-server software. In: International Conference on Computing and Information, pp. 535–546. Springer (1991)
118. Yager, R.R., Kreinovich, V.: Fair division under interval uncertainty. Int. J. Uncertain. Fuzziness Knowl. Based Syst. **8**(5), 611–618 (2000)

Appendix A
Notation

In general, the notation follows conventions from [1].

A.1 General Notions

Sets are (as usually) denoted by capital letters. The difference from other notions denoted by capital letters (like matrices) should be clear from the context.
$\#A$—number of elements of the (finite) set A;
$\frac{\partial f(x_0,y)}{\partial x}$—a short notation for $\left.\frac{\partial f(x,y)}{\partial x}\right|_{x=x_0}$;

$\mathbb{R}$—the set of real numbers.

The interval notation from [1] discourages using the "$\subset$" sign for "$\subseteq$". The author follows it, but making some exceptions: when it is absolutely impossible that $A = B$, using $A \subseteq B$, instead of $A \subset B$ would be misleading. A good example is Proposition 5.3. In such cases, the author uses the "$\subset$" sign, not following [1].

The Imaginary Unit

Following the engineering tradition, the imaginary unit is denoted as j and not i. Obviously, this is such a number that $j^2 = -1$.

A.2 Interval Analysis Notions

A closed interval is denoted using brackets: $[\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$. Open or half-open intervals are denoted using obverse brackets: $]\underline{x}, \overline{x}[$, $[\underline{x}, \overline{x}[$, $]\underline{x}, \overline{x}]$. Parenteses are popular to denote such intervals, but they can also have several other meanings (pairs, vectors, etc.), so we avoid using them for intervals.

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3

The following notation is used for interval-related quantities:
x, y, z, …(italic small letters)—real-valued variables and real-valued vector variables;
$\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, …(boldface small letters)—interval variables and interval vector variables (boxes);
$\underline{x}$—the lower bound of the interval $\mathbf{x} = [\underline{x}, \overline{x}]$;
$\overline{x}$—the upper bound of the interval $\mathbf{x} = [\underline{x}, \overline{x}]$;

$\mathrm{wid}\,\mathbf{x}$—width (diameter) of the interval $\mathbf{x}$;
$\mathrm{mid}\,\mathbf{x}$—midpoint of the interval $\mathbf{x}$;

A, B, C, …(italic capital letters)—real-valued matrices;
$\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, …(boldface capital letters)—interval-valued matrices;
$A_{i:}$—i-th row of a matrix;
$A_{:j}$—j-th column of a matrix;

$f(x)$, $g(x)$—real-valued functions;
$\mathsf{f}(\mathbf{x})$, $\mathsf{g}(\mathbf{x})$—(interval) inclusion functions;
$f'(x)$—the derivative of a function $f : \mathbb{R} \to \mathbb{R}$;
$\nabla f(x)$—the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$;
$\nabla\mathsf{f}(\mathbf{x})$—inclusion function of the gradient;

$\square S$—interval hull of the set $S \subseteq \mathbb{R}^n$; smallest box $\mathbf{x} \in \mathbb{IR}^n$ such that $\mathbf{x} \supseteq S$;

$\mathbb{IR}$—the set of intervals of real numbers;
$\mathbb{KR}$—the set of extended (Kaucher) intervals.

Reference

1. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Vychislennyie Tiehnologii (Computational Technologies) **15**(1), 7–13 (2010)

Appendix B
Standards for Numerical Computation

To perform computations on intervals, we need some representation of real numbers. Currently, the most commonly used representation of real numbers are *floating-point numbers*. They have a commonly accepted standard: IEEE 754 [7]; also the interval standard IEEE 1788-2015 assumes its use.

However, this is not the only possibility of real numbers representation. As we can possibly neglect other standards for floating-point computations (although there are certainly devices—for instance older GPUs—that are not compatible with the IEEE 754 guideline), there are at least two other important approximations of real numbers: *fixed-point numbers* and a new format of so-called *unums*.

In this appendix, all three formats are going to be briefly reviewed.

B.1 IEEE 754 Standard for Floating-Point Arithmetic

As it has already been stated elsewhere, this standard has been published in 1985 and the last revision has been done in 2008.

The standard defines the representation of floating-point numbers [7] (Fig. B.1):

This encodes the number:

$$(-1)^{(\text{sign})} \cdot (\text{significand}) \cdot (\text{base})^{(\text{exponent})} .$$

The size of the sign is always one bit: 0 for positive and 1 for negative numbers. Sizes of the exponent and significand depend on a specific format. For instance, for

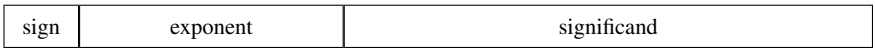

Fig. B.1 The IEEE 754 format

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3

single-precision numbers, they have values 8 and 24 respectively, while for double-precision numbers: 11 and 53.

What is the meaning of these values? Let us describe them subsequently.

The base is usually equal to 2; these are binary formats. Decimal formats, with base 10 are also defined by the standard; yet, they are less frequently used. The base is the part of the datatype and does not have to be stored.

The exponent is an integer number. It is stored in the form of a *biased exponent*, that is always positive. The *unbiased exponent* is the difference of the stored value and some number typical for a specific representation, e.g., 127 for the single-precision number (binary32) or 1023 for the double-precision one (binary64).

The significand, also called "coefficient" or, less precisely, "mantissa" is a non-integer of the form: $b_{n+1}.b_n b_{n-1} \ldots b_1 b_0$. The integer-part bit b_{n+1} is usually equal to one and, as we shall see, it is not stored explicitly. The other bits fit in the remaining part of the data field.

Let us consider an example, to make the things more clear.

The number 1 has sign zero, significand $1.0000 \ldots 0$ and unbiased exponent one. So for the double-precision number, we shall represent it as follows:

- the sign bit: 0,
- the biased exponent: 1024 (stored on 11 bits), so that the unbiased exponent was: $1024 - 1023 = 1$,
- the mantissa of the significand: 52 zero bits, the leading 53rd bit equal to one is not explicitly stored.

What are the feasible ranges of the exponent and significand? This is related to the size of these fields. For double-precision numbers, the (unbiased) exponent can have values from the interval $[-1023..1024]$, but the extreme values have special meaning:

- the maximal possible value denotes infinities and NaNs,
- the minimal possible value denotes zeros and so-called subnormal numbers.

Infinities ($+\infty$ and $-\infty$) have the significand filled with zeros, and NaNs—with ones; these are the values that have the maximal possible exponent (1024 for the double numbers).

Also zeros have a special representation, as the assumption of the leading bit of the significand being equal to one, does not allow explicitly representing zeros. But when the exponent is minimal (-1023 for the double-precision), this leading bit is assumed to be equal to zero. This is true for zeros and so-called *subnormal numbers*, also called *denormal* or *denormalized* numbers. Unlike zeros, they have a non-zero significand: $0.b_n b_{n-1} \ldots b_1 b_0$. Operations on subnormals are less precise than on "normal" numbers, yet still possible.

The above considerations show yet another peculiar feature of IEEE 754 floating-point number, called the *signed zero* concept: we have distinctly represented numbers $+0.0$ and -0.0! To make the things even more complicated, the standard requires them to be considered equal by comparison operators. Consequently, the following C++ program:

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    double x{0.0};
    double y{-0.0};
    cout << boolalpha;
    cout << (x == y) << "\n"; //true
    cout << (signbit(x) == signbit(y)) << "\n"; //false
}
```

will print "true" in the first output operation, but false in the second one, which is far from being intuitive, but consistent and justified.

There are also several NaN values, as all non-zero significands (and both signs) for the maximal exponent value represents a NaN. This is also consistent, as—according to the standard—the comparison `(NaN == NaN)` should return false.

These and many other interesting features of IEEE 754 arithmetic have fortunately little influence on the implementation of interval arithmetic. More important are rounding modes, but they have been described earlier in the monograph (Chap. 8). Also, Chap. 10 of [9] discusses several interesting issues of using the IEEE 754 format in interval arithmetic.

B.2 Fixed-Point Formats

Current processors typically have a built-in floating-point unit (FPU); this makes operations on such data relatively efficient. Nevertheless, there are still several devices—like microcontrollers or other CPUs devoted to embedded systems—that lack such a component. Using floating-point computations on this sort of hardware may be pretty expensive: not only with respect to time, but also energy consumption. As we have already stated in Chap. 9, interval algorithms have found several—actual and potential—applications in robotics, control, and measurement systems; hence, the possibility of their efficient execution in embedded systems is quite important. To the best knowledge of the author, such implementation of interval arithmetic has not been explicitly considered, yet.

There are a few formats for fixed-point computations; one of the most popular is the Q-format [15]. The idea is simple: numbers are stored as signed integers. We specify the number of bits that represent the fractional part (Qm) and (optionally) also the number of bits in the integer part (Q$n.m$). For instance: "Q16.16" means that the number will consist of 32 bits: 16 bits to store the integer part and 16—the fractional part. Please note that what is stored are just integer numbers; parameters n and m are part of the type, not of a specific fixed-point number.

Other words: we explicitly store integer nominators and denominators of type 2^m are implicit.

Thanks to the representation, three of the arithmetic operations: addition, subtraction and multiplication, can be performed by the direct use of their integer counterparts. Specifically:

$$\begin{aligned}\frac{K_1}{2^m} + \frac{K_2}{2^m} &= \frac{K_1 + K_2}{2^m}\,,\\ \frac{K_1}{2^m} - \frac{K_2}{2^m} &= \frac{K_1 - K_2}{2^m}\,,\\ \frac{K_1}{2^m} \cdot \frac{K_2}{2^m} &= \frac{\mathrm{int}(K_1 \cdot K_2 \cdot 2^{-m})}{2^m}\,.\end{aligned}$$

By $\mathrm{int}(\cdot)$ we mean casting to integers: either $\lfloor\cdot\rfloor$ or $\lceil\cdot\rceil$, depending on if we want to round downwards or upwards. Obviously, multiplication by powers of two can be performed using a bit shift of the number.

Only the division has to be implemented in a more sophisticated manner, as we need to deal with division of nominators:

$$\frac{K_1}{2^m} \div \frac{K_2}{2^m} = \frac{\mathrm{int}(\frac{K_1}{K_2} \cdot 2^m)}{2^m}\,. \tag{B.1}$$

If we are to preform interval computations, outward rounding for addition and subtraction should be simple; for multiplication and division, the key is proper casting to integers. Obviously, proper rounding has to be performed also for the operators casting the Q*n*.*m* type to/from floating-point or integer numbers.

Neither the Q format nor any other fixed-point format is the part of current C++ standards, but there are third-party implementations, like the libraries [8, 10], to name just modern C++11 packages. Several other ones can easily be found on the web—for instance the Java package [2].

B.3 A New Format—Unums

The term *unum* stands for "universal number"; this format has been developed by Gustafson in his 2015 book "The End of Error" [3]. The book sorely criticizes the existing representation of floating-point numbers, proposing a quite different format.

It is compatible with the IEEE floating-point standard, described earlier, but—as explained in [13]—some concepts form IEEE 754 are not necessary for unums.

The basic ideas can be formulated as follows:

1. The IEEE floating-point numbers have fixed sizes of the exponent and significand fields; by giving these fields variable sizes, we can highly increase the precision, using the same or smaller number of bits.
2. The IEEE floating-point numbers are a discrete subset of $\mathbb{R}$; unums instead contain either a precise number from a similar discrete set or an open interval between such numbers.

Thanks to the second "trick", unums are able (at least, in a certain sense) to represent all number from $\mathbb{R}$—not only from its discrete subset.

The format proposed in the book [3] has six fields, as in Fig. B.2.

The first three are similar to the IEEE 754 floating-point format, but—in contrast to the former—the exponent and significand may have various size. Then, we have the bit u, that indicates whether the unum represents an exact number ($u = 0$) or an open interval between the representable numbers ($u = 1$). Finally, the sizes of significand and exponent are stored. As we can observe, the format does not have a fixed size, which is its clear drawback. Nevertheless, the number of bits required to achieve a given precision is typically smaller, than for IEEE 74 floats.

Signed infinities can be represented for unums, as for floating-point numbers; there is also a NaN (not for unum 2.0 and 3.0 representations!), but it has a single representation only (and, according to [13], it is not necessary at all, only introduced for compatibility reasons).

The Gustafson's web page [6] contains several more papers and presentations, elaborating many more details of unums, including their new versions.

The unum approach has been criticized by William Kahan [11]. In response, Gustafson proposed a new version: unums 2.0, breaking the compatibility with IEEE 754 format, but fixing several of the drawbacks.

The latest version, unum 3.0 [5], is supposed to be hardware-friendly, as it has a fixed size of the overall data record and it allows implementing several operations using hardware-encoded integer procedures (and hence, very efficiently). The format for unum 3.0 has been presented in Fig. B.3.

Only the (single-bit) sign and (variable-sized) regime are mandatory. The regime either occupies the rest of the record or stops at some place, leaving some place for the other two fields.

The regime contains an integer number, coded in unary: it is a bit-string containing of identical bits. The regime field ends when either a bit with different value is

Fig. B.2 The unum 1.0 format

Fig. B.3 The unum 3.0 format

encountered or the whole record ends. Thanks to this representation, the regime field can have a variable length, not coded elsewhere. The regime bit string can consist either of zeros followed by a one or by ones followed by a zero. These two possibilities allow to code the sign of the encoded number: $k = -m$, for the first case, and $k = m - 1$, for the second one. This value defines the scaling factor.

If there are free bits after the regime field, there is an exponent; it has a fixed size *es* in this format. The exponent is not biased, neither has it a sign: it is always a positive number, encoded in binary.

The overall encoded number is:

$$(-1)^{(\text{sign})} \cdot 2^{k \cdot 2^{es}} \cdot 2^{(\text{exponent})} \cdot (1.\text{fraction}) \, .$$

There is no NaN for this representation, as special values can be represented as sets of numbers (so-called *valids*, in contrast with precise unums—*posits*), e.g., $\sqrt{-1} = \emptyset$, $0/0 = [-\infty, +\infty]$, $1^{\infty} = [0, +\infty]$, etc.; cf. [4].

Ubounds

Intervals having unums as their endpoints are called ubounds [6]. There is an interesting subtlety with this representation: when the bounds are, e.g., type 1.0 unums with the *u*-bit set to 1—open sets of "non-representable" numbers—the whole interval becomes open, also. A discussed in Sect. 1.11B of [18] (cf. also Sect. 2.8 of this monograph), using non-closed intervals in the interval calculus may lead to various mathematical difficulties.

On the other hand, we can restrict the bounds of such intervals to "precise" unums (posits), only. The problems seems to require further studies and achieving some concensus in the community.

Use of Unums

As stated above, the unum approach has been criticized, but also it has its advocates. Kulisch in [13] encourages their use, claiming that (as IEEE floating-points) they are compatible with the theory of rounded computations [12].

Currently, to the best knowledge of the author, experimental implementations for the Julia [1] and Matlab [14] languages have been introduced. Also, Ruffaldi implemented Python 3.5 [17] and C++11 [16] packages. Yet it remains to be seen, if the idea will get universally accepted or not.

References

1. Unums.jl Julia library (2016). https://github.com/JuliaComputing/Unums.jl
2. Garcia, M.: Q-Number-Format for Java (2018). https://github.com/mgarcia01752/Q-Number-Format
3. Gustafson, J.L.: The End of Error: Unum Computing. Chapman and Hall/CRC (2015)

4. Gustafson, J.L.: A radical approach to computation with real numbers (2016). http://www.johngustafson.net/presentations/Multicore2016-JLG.pdf
5. Gustafson, J.L.: Beating floating point at its won game: Posit arithmetic (2017). http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf
6. Gustafson, J.L.: Web page (2018). http://www.johngustafson.net/unums.html
7. IEEE: 754-2008–IEEE standard for floating-point arithmetic (2008). http://ieeexplore.ieee.org/document/4610935/
8. Ikkala, J.: Fixed-point numbers C++11 library (2018). https://github.com/juliusikkala/Fixed-point
9. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
10. Johnston, P.: C++11 Fixed Point Arithmetic Library (2018). https://embeddedartistry.com/blog/2017/8/25/c11-fixed-point-arithemetic-library
11. Kahan, W.: A critique of John L. Gustafson's The End of Error–Unum Computation and his a radical approach to computation with real numbers (2016)
12. Kulisch, U.: An axiomatic approach to computer arithmetic with an appendix on interval hardware. In: PPAM 2011 (9th International Conference on Parallel Processing and Applied Mathematics) Proceedings. Lecture Notes in Computer Science, vol. 7204, pp. 484–495 (2012)
13. Kulisch, U.: Up-to-date interval arithmetic: from closed intervals to connected sets of real numbers. In: PPAM 2015 Proceedings. Lecture Notes in Computer Science, vol. 9574, pp. 413–434 (2016)
14. Kvasnica, M.: MUNUM Matlab package (2018). https://bitbucket.org/kvasnica/munum
15. Oberstar, E.L.: Fixed-point representation & fractional math. In: Oberstar Consulting, vol. 9 (2007)
16. Ruffaldi, E.: The cppunum2 C++ unum 2.0 library (2018). https://github.com/eruffaldi/cppunum2
17. Ruffaldi, E.: The pypnunum Python 3.5 unum 2.0 library (2018). https://github.com/eruffaldi/pyunum2
18. Shary, S.P.: Finite-dimensional Interval Analysis. Institute of Computational Technologies, SB RAS, Novosibirsk (2013)

Appendix C
Implementations of the Interval Class in Various Languages

Various implementations, in several programming languages, can be provided. As indicated elsewhere, we can find several C and C++ libraries (cf. Chap. 8) and, e.g., some Julia [2] and OCaml codes [3].

In this Appendix, let us present a few toy implementations for some other popular languages. They will not be complete: to spare space, we shall only define the format for printing intervals and two arithmetic operations: addition and multiplication. Subtraction and division can easily be added, for all presented implementations.

Despite their simplicity, the presented interval data types are not completely useless. Not only can they be used for educational purposes, but also they may find applications in distributed systems, where various components are written in different languages. Such heterogeneous, multi-language systems are usually based on some message-oriented middleware (the simple ZeroMQ [13] or more advanced RabbitMQ [11], possibly Kafka [7] or yet another message broker) or remote procedure call (SOAP, XML-RPC, JSON-RPC, etc.). In such distributed applications, the core computations would probably be performed by C, C++, Fortran or another high-performance language, but the user interface is likely to be manufactured in some web technology, like JavaScript. Also, gathering and reduction of the data will possibly require less computational efficiency.

Obviously, just for the data transfer and user experience, we need only the representation of an interval, using a pair of numbers. But other operations will be handy not only for the debugging purposes, but also, as already indicated, for data reduction.

Python, which we present first, is particularly often used as a "glue" for several technologies in Web services and similar frameworks. Hence, the Python-based implementation of the interval data type might turn out to be particularly useful.

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3

C.1 Python 3

The language created by Guido van Rossum is pretty popular in the scientific computing community. There exist (or at least have existed) proficient implementations of the interval calculus in Python, e.g., in the Sage package, currently renamed to SageMath [12].

Here, let us present a simple class in Python 3:

```
class Interval:
    def __init__(self, lb, ub):
        self.lb, self.ub = lb, ub

    def __str__(self):
        return "[" + str(self.lb) + ", " + str(self.ub) + "]"

    def __add__(self, a):
        return Interval(self.lb + a.lb, self.ub + a.ub)

    def __mul__(self, a):
        lb = min(self.lb*a.lb, self.lb*a.ub, self.ub*a.lb, self.ub*a.ub)
        ub = max(self.lb*a.lb, self.lb*a.ub, self.ub*a.lb, self.ub*a.ub)
        return Interval(lb, ub)

    # ... (insert other arithmetic operations here) ...

    def __pow__(self, n):
        if (n % 2 != 0):
            return Interval(self.lb ** n, self.ub ** n)
        else:
            a, b = self.lb ** n, self.ub ** n
            if self.lb > 0 or self.ub < 0:
                if a <= b:
                    return Interval(a, b)
                else:
                    return Interval(b, a)
            else:
                # the interval contains zero
                return Interval(0, max(a, b))

if __name__ == "__main__":
    x = Interval(1.0, 2.0)
    y = Interval(2.0, 4.0)
    z = x + y
    print(z)
```

In addition to the two arithmetic operations, we have defined the power function, also. Python has the operator `**` for computing the power of an argument. In the above implementation, `n` is assumed to be an integer number.

If we wanted to have directed rounding in Python, we could use, in particular, the `mpmath` package, allowing operations, like:

```
lb = fadd(a.lb, b.lb, rounding='d')
ub = fadd(a.ub, b.ub, rounding='u')
```

It is worth noting, that switching the rounding mode may not be supported in some of the Python runtime environments; and example is Jython, as described below. Yet, the most popular ones: CPython and PyPy should handle such operations properly.

Python provides us also several other tools, like the `Decimal` type, various extensions of the `NumPy` package (including the `nextafter()` function) or possibility to run C code [4]; they are beyond the scope of this survey.

C.2 C#

An analogous C# class can be defined in the following manner:

```
using System;

class Interval {
    public double lb;
    public double ub;

    public Interval(double lb_=0.0, double ub_=0.0) {
        lb = lb_;
        ub = ub_;
    }

    public override string ToString() {
        return "[" + lb.ToString() + ", " + ub.ToString() + "]";
    }

    public static Interval operator+ (Interval a, Interval b) {
        Interval result = new Interval();
        result.lb = a.lb + b.lb;
        result.ub = a.ub + b.ub;
        return result;
    }

    public static Interval operator* (Interval a, Interval b) {
        Interval result = new Interval();
        result.lb = Math.Min(a.lb*b.lb, a.lb*b.ub, a.ub*b.lb, a.ub*b.ub);
        result.ub = Math.Max(a.lb*b.lb, a.lb*b.ub, a.ub*b.lb, a.ub*b.ub);
        return result;
    }

    // ... (insert other arithmetic operations here) ...
}


public class Example {
    static void Main(string[] args) {
        Interval x = new Interval(1.0, 2.0);
        Interval y = new Interval(2.0, 4.0);
        Interval z = x + y;
        Console.WriteLine(z);
    }
}
```

Proficient C# programmers would probably implement fields `lb` and `ub` as private, and provide setters and getters for them, but let us stick to the presented version for simplicity.

If we wanted to use directed rounding, it is possible, also—at least since .NET framework 2.0—yet not very convenient. The `Math.Round()` method allows us to choose the rounding mode. Precisely, we have the following syntax:

```
public static double Round (double value,
                            int digits,
                            MidpointRounding mode)
```

where the `mode` can have values: `AwayFromZero` or `ToEven` [10]. Explicit rounding towards $+\infty$ or $-\infty$ is, to the best knowledge of the author, not possible—at least not in a simple manner.

Another solution is to use the `nextafter()` function (from the `math` package), that returns the next representable number towards a give direction. We shall not discuss the details here, as an analogous solution is available for Java, Scala, Golang, etc.; so, we refer to the sections devoted to these languages.

Yet another possibility is to use the `Decimal` type – different from IEEE floating-point types.

C# is not the only language working in the .NET runtime environment. It is worthwhile to mention at least two other languages: F# and Visual Basic. The former is similar to OCaml, so the aforementioned library in this programming language [3] could possibly be translated to F#. As for the latter one, it is rarely used by the scientific computing community, and we shall not discuss it. Other languages, working on the CLR (in Particular IronPython and IronRuby), have not gained much popularity.

Python and C# have allowed us to overload operators, making the implementation of the interval data type quite nice and easy to use. We get the same freedom in Ruby or Scala, but not in all languages.

C.3 Java

Java is similar to C#, but, as we shall see, there are various differences. In particular:

- operators cannot be overloaded,
- there are no default values of function arguments,
- some other operations are less convenient, e.g., computing the minimum or maximum of a sequence.

```
class Interval {
    public double lb;
    public double ub;

    public Interval(double lb_, double ub_) {
        lb = lb_;
        ub = ub_;
    }

    public Interval() {}

    public String toString() {
        return "[" + lb + ", " + ub + "]";
    }

    public static Interval add (Interval a, Interval b) {
        Interval result = new Interval();
        result.lb = a.lb + b.lb;
        result.ub = a.ub + b.ub;
        return result;
    }

    public static Interval mult (Interval a, Interval b) {
        Interval result = new Interval();
        result.lb = Math.min(a.lb*b.lb, Math.min(a.lb*b.ub,
                             Math.min(a.ub*b.lb, a.ub*b.ub)));
        result.ub = Math.max(a.lb*b.lb, Math.max(a.lb*b.ub,
                             Math.max(a.ub*b.lb, a.ub*b.ub)));
        return result;
    }

    // ... (insert other arithmetic operations here) ...
}


public class IntervalExample {
    public static void main(String[] args) {
        Interval x = new Interval(1.0, 2.0);
        Interval y = new Interval(2.0, 4.0);
        Interval z = Interval.add(x, y);
        System.out.println("" + z);
    }
}
```

As for C#, Java programmers would probably prefer to implement fields `lb` and `ub` as private, and provide setters and getters for them.

It has to be noted that for Java, we cannot change the rounding mode. The point 2.8.1 of the Oracle's documentation makes it clear: "*The rounding operations of the Java Virtual Machine always use IEEE 754 round to nearest mode. Inexact results are rounded to the nearest representable value, with ties going to the value with a zero least-significant bit. This is the IEEE 754 default mode. But Java Virtual Machine instructions that convert values of floating-point types to values of integral types*

round toward zero. The Java Virtual Machine does not give any means to change the floating-point rounding mode." [6].

This property of JVM has been severely criticized [5]. Probably, we can still enforce switching the rounding mode, by calling C functions from Java.

Yet another possibility is to use the function `nextAfter()` from the `java.lang.Math` package to extend the lower bound downwards and the upper bound upwards. Such an interval will be more overestimated than in case of using the properly rounded arithmetic operation, but will be guaranteed to contain the actual value. For instance, the method `add()` of the `Interval` class could look as follows:

```
public static Interval add (Interval a, Interval b) {
    Interval result = new Interval();
    result.lb = java.lang.Math.nextAfter(a.lb + b.lb,
        Double.NEGATIVE_INFINITY);
    result.ub = java.lang.Math.nextAfter(a.ub + b.ub,
        Double.POSITIVE_INFINITY);
    return result;
}
```

As the former version of the presented Java application prints:

```
[3.0, 6.0]
```

after the change we obtain:

```
[2.9999999999999996, 6.000000000000001]
```

Also, it is worth noting that, in the past, some effort has been put to incorporate intervals in the Java standard library (the names of William Walster and David Hough have to be mentioned here) [9]. Unfortunately, to the best knowledge of the author, these efforts have not been successful.

C.4 Scala

Java is not the only language working on the JVM (Java Virtual Machine). Another one, that has been continuously becoming more and more popular, is Scala. Not only is it much more succinct than Java, but also it allows operator overloading.

Programs can be written in Scala in various manners; in particular, they can adopt several features of functional programming. Here let us show a simple sample class, representing an interval:

```
class Interval(val lba: Double, val uba: Double) {
    var lb: Double = lba
    var ub: Double = uba

    def this() = this(0.0, 0.0)
```

```
    override def toString = "[" + lb + ", " + ub + "]"

    def +(x: Interval) = new Interval(lb + x.lb, ub + x.ub)

    def *(x: Interval) = new Interval(math.min(lb*x.lb,
            math.min(lb*x.ub, math.min(ub*x.lb, ub*x.ub))),
        math.max(lb*x.lb, math.max(lb*x.ub,
            math.max(ub*x.lb, ub*x.ub))))

    // ... (insert other arithmetic operations here) ...
}

object IntervalExample extends App {
    val a = new Interval(1.0, 2.0)
    val b = new Interval(2.0, 4.0)
    val c = a + b
    println(c)
}
```

As in Scala the "underline" sign has a special meaning, we had to change the convention of naming the constructor's arguments, that we had used for C# and Java.

Despite its elegance, Scala suffers all the limitations of the JVM, described in the previous section. And, obviously, this applies to all other languages working on the JVM: Jython, Groovy, Clojure, Kotlin, etc.

Nevertheless, we can apply the same remedy, as for Java—function `nextAfter`. The modified plus operator is presented by the following code:

```
def +(x: Interval) : Interval = {
    val resultlb: Double = java.lang.Math.nextAfter(lb + x.lb,
        Double.NegativeInfinity)
    val resultub: Double = java.lang.Math.nextAfter(ub + x.ub,
        Double.PositiveInfinity)
    return new Interval(resultlb, resultub)
}
```

C.5 Golang

Another language that has been gaining growing popularity in the recent years is the Google's Golang. The name is sometimes abbreviated simply as "Go"; we shall not use this version of the name, to avoid confusion with Go!, the agent-based language of Francis McCabe and Keith Clark and other tools.

Golang is simple and elegant, yet pretty different from other languages presented in this chapter. It is not "essentially" object-oriented, yet it makes use of several object-like concepts. We do not define "classes" in Golang, yet we can define structures. All functions and "methods" are defined outside the structure, but the structure can implement an *interface*.

In particular, our structure named `Interval` implements the interface `Stringer`. However, please note, we nowhere have any "`implements`" declaration as we would have in Java or C#. The relation between these two entities is *deduced* by the compiler from the fact that the function `String()` is used for the structure `Interval` (actually, it is used implicitly, in our program).

The name "`Stringer`" refers to "the interface that contains the method `String()`". This is a widely accepted convention in Golang that an interface contains only a single method and its name is identical to the method's name with suffix "-er".

All details can be found in the on-line documentation for Golang and several tutorials available [8].

```
package main

import "fmt"
import "math"

type Interval struct {
    lb float64
    ub float64
}

func (x Interval) String() string {
    return fmt.Sprintf("[%v, %v]", x.lb, x.ub)
}

func (a Interval) add (b Interval) Interval {
    result := Interval{0, 0}
    result.lb = a.lb + b.lb
    result.ub = a.ub + b.ub
    return result
}

func (a Interval) mult (b Interval) Interval {
    result := Interval{0, 0}
    result.lb = math.Min(a.lb*b.lb, math.Min(a.lb*b.ub,
        math.Min(a.ub*b.lb, a.ub*b.ub)))
    result.ub = math.Max(a.lb*b.lb, math.Max(a.lb*b.ub,
        math.Max(a.ub*b.lb, a.ub*b.ub)))
    return result
}


func main() {
    a := Interval{1.0, 2.0}
    b := Interval{2.0, 4.0}
    c := a.add(b)
    fmt.Println(c)
}
```

There is no operator overloading (nor even function overloading!) in Golang, so we cannot define arithmetic operators for the interval type.

The inconvenient computation of minimum and maximum of several numbers can be improved by defining, e.g., the following function:

```
func min(args ...float64) float64 {
    min := args[0]
    for _, x := range args[1:] {
        min = math.Min(min, x)
    }
    return min
}
```

Now, we can simply write:

```
result.lb = min(a.lb*b.lb, a.lb*b.ub, a.ub*b.lb, a.ub*b.ub)
```

The author has not been able to track down any information about switching the rounding modes in Golang. However, in the `math` package, we have a convenient function `Nextafter()`, analogous to previously discussed languages. It allows, in particular, the following implementation of addition:

```
func (a Interval) add (b Interval) Interval {
    result := Interval{0, 0}
    result.lb = math.Nextafter(a.lb + b.lb, math.Inf(-1))
    result.ub = math.Nextafter(a.ub + b.ub, math.Inf(1))
    return result
}
```

Obviously, multiplication and other arithmetic operations can be written in a similar manner.

C.6 JavaScript

Let us finish this short survey with a JavaScript implementation sample. It is interesting, as this language (prior to ECMAScript 6 specification) followed a pretty unique variant of object-oriented programming: it had no “classes”, understood as other object-oriented languages have, but only objects.

The below code should illustrate this idea:

```
function Interval (lb, ub) {
    this.lb = lb;
    this.ub = ub;
    this.show = showInterval;
    this.add = addIntervals;
    // ... (insert other interval operations here) ...
}

function showInterval() {
    document.write ("[" + this.lb + ", " + this.ub + "]");
}
```

```
function addIntervals(a, b) {
    return new Interval(a.lb + b.lb, a.ub + b.ub);
}

var a = new Interval (1.0, 2.0); var b = new Interval (2.0, 4.0);
var c = a.add(b); c.show();
```

JavaScript does not support operator overloading; there are some tricks to fake it, but they do not seem applicable to our case.

It is not surprising that JavaScript does not seem to allow control of the rounding mode or functions like `nextAfter()`: numerical computations are not the focus of this language. Yet, the implementation is, in the opinion of the author, significant, as JavaScript is ubiquitous in today web development: both on client and server (node.js) sides; hence it is likely to be used as a web interface to our interval solver: both for data input and results presentation.

C.7 Summary

We have presented some basic implementations of the interval data type in a few popular programming languages. The author had also two other implementations: in PHP and Fortran 95, but decided against presenting them. PHP seems too irrelevant for scientific computing and Fortran is too verbose. It is also worth noting that there exist proficient implementations of the interval arithmetic in Fortran, like GlobSol, mentioned in Chap. 8 (see also [1]).

It would be worthwhile to provide such implementations for many other technologies. Kotlin and F# have already been mentioned. Other tools worth attention are functional languages, like the famous Haskell.

References

1. GlobSol solver (2015). https://interval.louisiana.edu/GlobSol/
2. ValidatedNumerics package (2016). https://github.com/JuliaIntervals/ValidatedNumerics.jl
3. Alliot, J.M., Gotteland, J.B., Vanaret, C., Durand, N., Gianazza, D.: Implementing an interval computation library for OCaml on x86/amd64 architectures. In: OUD 2012, OCaml Users and Developers workshop (2012)
4. Gorelick, M., Ozsvald, I.: High Performance Python: Practical Performant Programming for Humans. O'Reilly Media, Inc. (2014)
5. Kahan, W., Darcy, J.D.: How Java's floating-point hurts everyone everywhere. In: ACM 1998 workshop on Java for high-performance network computing, p. 81. Stanford University (1998)

6. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine specification (2015). https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.8.1
7. Apache kafka: A distributed streaming platform (2018). https://kafka.apache.org
8. Golang documentation (2018). https://golang.org/doc/
9. Let's add intervals to Java (a proposal) (2018). http://www.cs.utep.edu/interval-comp/java.html
10. Math.Round C# documentation (2018). https://msdn.microsoft.com/en-us/library/system.midpointrounding%28v=vs.80%29.aspx
11. RabbitMQ (2018). https://www.rabbitmq.com
12. SageMath (2018). http://www.sagemath.org
13. ZeroMQ (2018). http://zeromq.org

Solutions

Problems of Chap. 2

2.1 $\mathsf{f}\big([-2,-1]\big) = [-7,3] \supset [-4,0] = \text{range}\big(f,[-2,-1]\big)$
$\mathsf{f}\big([1,2]\big) = [2,12] = \text{range}\big(f,[1,2]\big)$
$\mathsf{f}\big([-2,2]\big) = [-8,12] \supset [-4,12] = \text{range}\big(f,[-2,2]\big)$

2.2 Distributivity of interval multiplication
(a) Yes. (b) Yes. (c) Yes. (d) Yes. (e) No. (f) No.

2.3 Consider $\mathbf{x} = [0,5]$ and $\mathbf{y} = [1,3]$. We have three possibilities:

(a) $H_1 = \{X \in [0,1] \text{ and } Y \in [1,3]\}$,
(b) $H_2 = \{X \in [1,3] \text{ and } Y \in [1,3]\}$,
(c) $H_3 = \{X \in [3,5] \text{ and } Y \in [1,3]\}$.

Please note that $X \leq Y$ is *certain* for H_1, *possible* for H_2 and has *probability zero* (only the unlikely case of $X = Y = 3$) for H_3. The *basic probability assignment* to these events are:

$$P(H_1) = \frac{1-0}{5-0} \cdot \frac{3-1}{3-1} = \frac{1}{5} = 0.2\,,$$
$$P(H_2) = \frac{3-1}{5-0} \cdot \frac{3-1}{3-1} = \frac{2}{5} = 0.4\,,$$
$$P(H_3) = \frac{5-3}{5-0} \cdot \frac{3-1}{3-1} = \frac{2}{5} = 0.4\,.$$

And we obtain:

$$Bel\big(\{X \leq Y\}\big) = P(H_1) = 0.2\,,$$
$$Pl\big(\{X \leq Y\}\big) = P(H_1) + P(H_2) = 0.6\,.$$

B. J. Kubica, *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems*, Studies in Computational Intelligence 805,
https://doi.org/10.1007/978-3-030-13795-3

Obviously, for $\mathbf{x} = [0, 3]$ and $\mathbf{y} = [3, 5]$, we have:

$$Bel\big(\{X \leq Y\}\big) = Pl\big(\{X \leq Y\}\big) = 1 \,.$$

2.4 Various implementations, in several programming languages, can be provided. On the web we can find several C and C++ libraries (cf. Chap. 8) and, e.g., some OCaml codes [1]. We present implementations in a few other languages in Appendix C.

Yet, it is worth noting that, to obtain a useful code, one has to implement many more functions. We need not only operations on two intervals, but also, e.g., operations on an interval and a number. Covering all necessary cases (not to mention compatibility with the IEEE Std 1788-2015) may be quite cumbersome and in some languages, the programmer might face some limitations, e.g., in Lua, the operands should be of the same type.

This exercise shows, it is worthwhile to use third-party libraries for the interval calculus; cf. Chap. 8.

Reference

1. Alliot, J.M., Gotteland, J.B., Vanaret, C., Durand, N., Gianazza, D.: Implementing an interval computation library for OCaml on x86/amd64 architectures. In: OUD 2012, OCaml Users and Developers workshop (2012)