APPLIED INTERVAL ANALYSIS

Luc Jaulin,
Michel Kieffer
Olivier Didrit
and Éric Walter

Springer

Applied Interval Analysis

Springer-Verlag London Ltd.

Luc Jaulin, Michel Kieffer, Olivier Didrit
and Éric Walter

Applied Interval Analysis

With Examples in Parameter and State Estimation, Robust Control and Robotics

With 125 Figures

Springer

Luc Jaulin, PhD
Michel Kieffer, PhD
Olivier Didrit, PhD
Éric Walter, PhD
Laboratoire des Signaux et Systèmes, CNRS-SUPÉLEC-Université Paris-Sud, France

British Library Cataloguing in Publication Data
Applied interval analysis
1. Interval analysis (Mathematics)
I. Jaulin, Luc
519.4
ISBN 978-1-4471-1067-5

Library of Congress Cataloging-in-Publication Data
Applied interval analysis / Luc Jaulin ... [et al.].
p. cm.
ISBN 978-1-4471-1067-5 (alk. paper)
1. Interval analysis (Mathematics) I. Jaulin, Luc, 1967-
QA297.75 .A664 2001-04-26 511'.42—dc21
2001020164

ISBN 978-1-4471-1067-5 ISBN 978-1-4471-0249-6 (eBook)
DOI 10.1007/978-1-4471-0249-6

http://www.springer.co.uk

Originally published by Springer-Verlag London Berlin Heidelberg in 2001
Softcover reprint of the hardcover 1st edition 2001

Typesetting: Electronic text files prepared by authors
69/3830-543210 Printed on acid-free paper SPIN 10741551

Preface

At the core of many engineering problems is the solution of sets of equations and inequalities, and the optimization of cost functions. Unfortunately, except in special cases, such as when a set of equations is linear in its unknowns or when a convex cost function has to be minimized under convex constraints, the results obtained by conventional numerical methods are only local and cannot be guaranteed. This means, for example, that the actual global minimum of a cost function may not be reached, or that some global minimizers of this cost function may escape detection. By contrast, interval analysis makes it possible to obtain guaranteed approximations of the set of *all* the *actual* solutions of the problem being considered. This, together with the lack of books presenting interval techniques in such a way that they could become part of any engineering numerical tool kit, motivated the writing of this book.

The adventure started in 1991 with the preparation by Luc Jaulin of his PhD thesis, under Eric Walter's supervision. It continued with their joint supervision of Olivier Didrit's and Michel Kieffer's PhD theses. More than two years ago, when we presented our book project to Springer, we naively thought that redaction would be a simple matter, given what had already been achieved... Actually, this book is the result of fierce negotiations between its authors about what should be said, and how! At times, we feared that we might never end up with an actual book, but we feel that the result was worth the struggle.

There were at least two ideas on which we easily agreed, though. First, the book should be as simple and understandable as possible, which is why there are so many illustrations and examples. Secondly, readers willing to experiment with interval analysis on their own applications should be given the power to do so.

Many people contributed to our conversion to interval analysis, and it is impossible to quote all of them, but we would like at least to thank Vladik Kreinovich for all the energy that he puts into the Interval Computations WEB site and for all that we learned there.

Special thanks are due to Michel Petitot for his help in exploring the mysteries of ADA and the Stewart-Gough platform, to Dominique Meizel for introducing us to robot localization and tracking, to Olaf Knüppel and Siegfried

M. Rump for making PROFIL/BIAS and INTLAB available, to Isabelle Braems, Martine Ceberio, Ramon Moore, Stefan Ratschan and Nathalie Revol for their constructive remarks when reading earlier versions of the manuscript, and to our editorial assistant Oliver Jackson, whose friendly enquiries were instrumental in the release of this book this millennium.

We would also like to express our gratitude to Guy Demoment, head of the *Laboratoire des Signaux et Systèmes* and to Jean-Louis Ferrier, head of the *Laboratoire d'Ingénierie des Systèmes Automatisés* for their support and the way they managed to shield us from the perturbations of the outside world.

The French *Centre National de la Recherche Scientifique* provided us with ideal working conditions, and partial support by *INTAS* is also gratefully acknowledged.

Contents

Notation

The following tables describe the main typographic conventions and symbols to be used.

Punctual quantities

x	:	punctual scalar
x^*	:	actual value of an uncertain variable x
$\check{x}$	:	prior value of an uncertain variable x
$\widehat{x}$	:	posterior value of an uncertain variable x
$\mathbf{x}$	:	punctual column vector
$\mathbf{x}^{\mathrm{T}}$	:	punctual row vector
$\mathbf{0}$	:	vector of zeros
$\mathbf{1}$	:	vector of ones
$\mathbf{X}$	:	punctual matrix
$\mathbf{O}, \mathbf{O}_{n\times m}$	:	matrix of zeros, $(n \times m)$ matrix of zeros
$\mathbf{I}, \mathbf{I}_n$	:	identity matrix, $(n \times n)$ identity matrix
$\mathrm{Im}(s)$	:	imaginary part of s
$\mathrm{Re}(s)$	:	real part of s

Sets

$\varnothing$	:	empty set
$\mathbb{S}$	:	set
$\mathbb{N}$	:	set of all positive integers
$\mathbb{Z}$	:	set of all integers
$\mathbb{R}$	:	set of all real numbers
$\mathbb{IR}$	:	set of all interval real numbers
$\mathbb{C}$	:	set of all complex numbers
$\mathbb{C}^-$	:	set of all complex numbers with a strictly negative real part
$\mathbb{B}$	:	set of all Boolean numbers
$\mathbb{IB}$	:	set of all interval Boolean numbers
$\partial\mathbb{S}$	:	boundary of $\mathbb{S}$
$[\mathbb{S}]$	:	interval hull of $\mathbb{S}$
$\overline{\mathbb{S}}$	:	outer approximation of $\mathbb{S}$
$\underline{\mathbb{S}}$	:	inner approximation of $\mathbb{S}$
$\mathcal{L}$	:	list, stack, queue, tree or graph

Intervals

$[x] = [\underline{x}, \overline{x}]$	:	interval scalar
$[\mathbf{x}] = [\underline{\mathbf{x}}, \overline{\mathbf{x}}]$	:	interval vector (or box)
$[\mathbf{X}] = \left[\underline{\mathbf{X}}, \overline{\mathbf{X}}\right]$	:	interval matrix
$[x_i] = ([\mathbf{x}])_i$	:	ith entry of $[\mathbf{x}]$
$[x_{ij}] = ([\mathbf{X}])_{ij}$	:	entry of $[\mathbf{X}]$ at ith row and jth column
$\mathrm{lb}([x])$	:	lower bound of $[x]$
$\mathrm{ub}([x])$	:	upper bound of $[x]$
$w\,([x])$	:	width of $[x]$
$\mathrm{mid}([x])$	:	centre of $[x]$

Other symbols

$\triangleq$	:	equal by definition
$:=$	:	assignment operator
$\forall$	:	universal quantifier (*for all*)
$\exists$	:	existential quantifier (*there exists*)
$\neg$	:	logical complementation
$\wedge$	:	logical `AND`
$\vee$	:	logical `OR`
$\mathbb{A} \times \mathbb{B}$	:	Cartesian product of $\mathbb{A}$ and $\mathbb{B}$
$\mathbb{A} \backslash \mathbb{B}$	:	$\{x \mid (x \in \mathbb{A}) \wedge (x \notin \mathbb{B})\}$
$\mathbb{A} \sqcup \mathbb{B}$	:	interval union of $\mathbb{A}$ and $\mathbb{B}$, equal to $[\mathbb{A} \cup \mathbb{B}]$

Functions

Functions are denoted with the same typographical convention as the elements of their image spaces, thus $[f](\cdot)$ is a scalar interval function and $[\mathbf{f}](\cdot)$ a vector interval function.

If $\mathbf{f}(.)$ is a once-differentiable function from $\mathbb{R}^{n_\mathrm{x}}$ to $\mathbb{R}^{n_\mathrm{y}}$, then its *Jacobian matrix* at $\mathbf{x}$ is

$$\mathbf{J}_\mathbf{f}(\mathbf{x}) \triangleq \begin{pmatrix} \dfrac{\partial f_1}{\partial x_1}(\mathbf{x}) & \cdots & \dfrac{\partial f_1}{\partial x_{n_\mathrm{x}}}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_{n_\mathrm{y}}}{\partial x_1}(\mathbf{x}) & \cdots & \dfrac{\partial f_{n_\mathrm{y}}}{\partial x_{n_\mathrm{x}}}(\mathbf{x}) \end{pmatrix}.$$

If $f(.)$ is a once-differentiable function from $\mathbb{R}^{n_\mathrm{x}}$ to $\mathbb{R}$, then its *gradient* at $\mathbf{x}$ is

$$\mathbf{g}_f(\mathbf{x}) \triangleq \begin{pmatrix} \dfrac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \dfrac{\partial f}{\partial x_{n_\mathrm{x}}}(\mathbf{x}) \end{pmatrix}.$$

If $f(.)$ is twice differentiable, then its *Hessian matrix* at $\mathbf{x}$ is the (symmetric) Jacobian matrix associated with its gradient, *i.e.*,

$$\mathbf{H}_f(\mathbf{x}) \triangleq \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \cdots & \dfrac{\partial^2 f}{\partial x_{n_\mathrm{x}} \partial x_1}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_1 \partial x_{n_\mathrm{x}}}(\mathbf{x}) & \cdots & \dfrac{\partial^2 f}{\partial x_{n_\mathrm{x}}^2}(\mathbf{x}) \end{pmatrix}.$$

Algorithms

Algorithms are described in a pseudo-code allowing the usual mathematical notation. The most important arguments are listed after the NAME of the algorithm as input arguments (in:), output arguments (out:) or input-output arguments (inout:). To facilitate reading, we take the liberty to omit some of them, such as inclusion functions, gradients, Hessian matrices... Blocks of statements are indicated by indentation. Any return statement causes an immediate return from the current algorithm. Return statements at the end of the algorithms are implicit.

For details about the implementation of these algorithms, see Chapter 11, where C++ code is set in `Typewriter`.

Part I

Introduction

1. Introduction

This book is about guaranteed numerical methods for approximating sets, and their application to engineering. Guaranteed means here that outer (and sometimes inner) approximations of the sets of interest are obtained, which can, at least in principle, be made as precise as desired. It thus becomes possible to achieve tasks often thought to be out of the reach of numerical methods, such as finding all solutions of sets of non-linear equations and inequalities or all global optimizers of possibly multi-modal criteria.

The figure on the cover illustrates this idea. It describes a question mark defined by a series of inequalities connected by logical operators. The prior space of interest has been partitioned into three sets of rectangles (two-dimensional versions of what we shall call boxes). The first one, in red, consists of rectangles proved to belong to the question mark. The second one, in blue, is made of rectangles proved to have an empty intersection with it. The last one, in yellow, contains rectangles for which nothing could be proved. The surface of the yellow region could easily be decreased, at the cost of more intensive computation.

The main tool to be used is interval analysis, based upon the very simple idea of enclosing real numbers in intervals and real vectors in boxes. This first made it possible to obtain guaranteed results on computers by direct transposition to interval variables of classical numerical algorithms usually operating on floating-point numbers. More recently, interval analysis also allowed the derivation of algorithms specifically dedicated to dealing with sets, with no real counterpart. This made it possible to use *numerical* methods to *prove* mathematical statements about sets. Algorithms based on interval analysis thus compete with those based on computer algebra, with the advantage that they can deal with more general classes of problems and that even steps that can only be solved numerically (such as finding the roots of a high-degree polynomial equation) can nevertheless be solved in a guaranteed way. Note that the usual numerical methods based on Monte-Carlo sampling or on systematic gridding could not be used to prove even such simple properties as the emptiness or disconnected nature of a set.

L. Jaulin et al., *Applied Interval Analysis*

1.1 What Are the Key Concepts?

Some of the problems treated are still deemed unsolvable by many, so why would interval analysis allow one to solve them? A short answer, to be substantiated by this book, is that interval analysis allows guaranteed conclusions to be reached about the properties of boxes in search space after a finite number of operations, although the vectors in each of these boxes are not even denumerable. This is achieved by *wrapping* the sets of interest into boxes or unions of boxes upon which computations can be conducted more easily than upon the original sets.

Consider a box $[\mathbf{x}]$ of $\mathbb{R}^n$, a function f from $\mathbb{R}^n$ to $\mathbb{R}$ and a subset $\mathbb{S}$ of $\mathbb{R}^n$ defined by a series of constraints connected by logical operators such as `AND` or `OR`. The question mark on the cover is a two-dimensional example of such a set $\mathbb{S}$. Interval analysis makes it possible to implement three essential operations. The first one is computing an interval that contains the image of $[\mathbf{x}]$ by f. The key to this operation, which is a direct consequence of the properties of interval calculus, is the notion of *inclusion function*. The second operation is *testing whether* $[\mathbf{x}]$ *belongs to* $\mathbb{S}$, or more precisely whether $[\mathbf{x}] \subset \mathbb{S}$ or whether $[\mathbf{x}] \cap \mathbb{S} = \varnothing$. For this purpose, the notion of *inclusion test* will be introduced. The third operation is the *contraction of* $[\mathbf{x}]$ *with respect to* $\mathbb{S}$, *i.e.*, the replacement of $[\mathbf{x}]$ by a smaller box $[\mathbf{z}]$, such that $[\mathbf{x}] \cap \mathbb{S} = [\mathbf{z}] \cap \mathbb{S}$. If $[\mathbf{z}]$ turns out to be empty and $\mathbb{S}$ defines the feasibility set for the solution of some problem, then $[\mathbf{x}]$ can be eliminated from the list of boxes that may contain this solution.

When no conclusion can be reached about a given box, this box can be bisected into subboxes, and each of these can be studied in turn. This corresponds to *branch-and-bound* (or *branch-and-prune*) algorithms, whose main drawback is their exponential complexity in the number of interval variables. *Contractors* can be employed to decrease (sometimes eliminate) the need for splitting boxes into subboxes, thereby playing an essential role in the struggle against the curse of dimensionality.

These concepts and operations are at the core of the main algorithms to be considered, and any other mathematical theory that would allow their implementation could be substituted for interval analysis. Ellipsoids, for instance, have also been used for guaranteed computation on sets. Interval analysis, however, can be employed over a much wider class of functions f and constraints defining $\mathbb{S}$. Moreover, this can be done in such a way that rounding errors due to the inaccurate representation of real numbers on computers are taken into account to guarantee the results provided.

1.2 How Did the Story Start?

Moore completed his doctorate on the use of intervals to analyze and control numerical errors in computers in 1962. In 1966, he published his first

book *Interval Analysis* (Moore, 1966), which remains a reference to this day. During the same period, Hansen studied interval manipulation in linear algebra (Hansen, 1965), and a group of German researchers including Alefeld, Krawczyk and Nickel developed many aspects of computer implementation (Nickel, 1966). Interval analysis is thus a child of many fathers.

During the first twenty years, the spreading of the interval methodology remained relatively confined to the periphery of the initial seeds, notably in Germany within Karlsruhe University (Kulisch and Miranker, 1981). Among the new adepts who brought important advances, one may quote Neumaier (1985) on the solution of sets of linear and non-linear equations, and Ratschek and Rokne (1984) and Kearfott (1989a) on optimization.

During the 1990s, interval analysis has recruited a larger community. It now has its own journal *Interval Computations,* created in 1991 and renamed *Reliable Computing* in 1995, and several regular international conferences. References to thousands of papers can be found at the WEB site

```
http://liinwww.ira.uka.de/bibliography/
                    ?query=interval&case=off&partial=on
```

The reader is also advised to visit the very active site

```
http://www.cs.utep.edu/interval-comp/main.html
```

entirely dedicated to interval analysis.

1.3 What About Complexity?

Interval algorithms will always take longer than their real counterparts when such counterparts exist. Sometimes, it is necessary to split intervals into subintervals that may require vast quantities of memory to be stored, which soon leads to the curse of dimensionality. The increase factors in time and memory required vary considerably from one application to another. The examples treated in Chapters 6 to 8 should convince the reader that complexity is not prohibitive for quite a number of problems of practical interest. These problems have been treated over a period of about a decade, on a variety of personal computers, and by programs written in PASCAL, ADA and C++. We did not find it worthwhile to process all of them again for the sole benefit of giving unified computing times, and chose instead to indicate times as measured on the computers operating when the examples were treated. The reader may thus rest assured that the times indicated are pessimistic — and sometimes very pessimistic — upper bounds of what can be achieved with present-day personal computers.

1.4 How is the Book Organized?

Part II is devoted to basic tools. Every effort has been made to present them as simply as possible, in a concrete and readily applicable way. Some of the techniques reported appear in book format for the first time. Chapter 2 recalls a few simple notions of set theory that form the foundations of the methodology. It then presents the main concepts of interval analysis, including the very important notions of inclusion functions and inclusion tests. Chapter 3 is about subpavings, *i.e.*, sets of non-overlapping boxes to be used to approximate compact sets. The few notions of topology needed to quantify the distance between subpavings and compact sets are recalled. The representation of a useful class of subpavings by binary trees is explained, and computation on subpavings is applied to two operations of fundamental importance, namely set inversion and direct image evaluation. Chapter 4 presents contractors, *i.e.*, operators used to decrease the sizes of the domains on which variables may be allowed to vary if they are to satisfy a given set of constraints. Contractors have already been mentioned as playing a fundamental role in the struggle against the curse of dimensionality. Chapter 5 describes problem solvers. Contractors alone cannot solve all problems of interest and one must sometimes resort to the bisection of boxes to get better approximations of solution sets by subpavings. The problems considered include solving sets of non-linear equations or inequalities, and optimizing multi-modal and minimax criteria.

The ability of the tools of Part II to solve non-trivial engineering problems is demonstrated in Part III. Sufficient details are provided on each topic to allow readers with other applications in mind to grasp its significance. Chapter 6 is about estimation, *i.e.*, the use of experimental data to derive information on the numerical value of some uncertain variables, which may be assumed constant (parameter identification) or time-varying (state estimation or parameter tracking). Estimation is performed either by optimizing a cost function (this is the case, for instance, of least-square estimation), or by looking for all values of the vector of uncertain quantities of interest that are consistent with the data up to prior bounds on the acceptable errors. In both cases, interval analysis allows guaranteed results to be obtained. Chapter 7 deals with two basic problems of robust control. The first one is the analysis of the robustness of a given control system to uncertainty in the model of the process to be controlled. The second one, more complicated, is the design of a controller achieving a satisfactory level of performance in the presence of uncertainty. Chapter 8 addresses three difficult problems of robotics. The first one is the evaluation of all possible configurations of a parallel robot, known as a Stewart–Gough platform, given the lengths of its limbs, a now classical benchmark in computer algebra. The second problem is the planning of a collision-free path in an environment cluttered with obstacles. The last one is the localization and tracking of a vehicle from on-board distance measurements in a partially known environment.

An in-depth treatment of implementation issues in Part IV facilitates the understanding and use of freely available software that makes interval computation about as simple as computation with floating-point numbers. Chapter 9 presents automatic differentiation, a numerical tool that can be used to obtain guaranteed estimates of the derivatives of functions with respect to their arguments, as needed by some of the algorithms described earlier. Chapter 10 describes the facilities offered by the IEEE-754 standard for binary floating-point arithmetic, adhered to by most present-day computers, and its limitations. Pointers to readily available software dedicated to interval computation are also provided. In Chapter 11, readers are given the basic information needed to build their own C++ interval libraries or to use the PROFIL/BIAS library. The implementation of the main algorithms described in the book and their application to illustrative examples is considered in detail through exercises. The source code corresponding to the solutions of all these exercises can be downloaded from the WEB site

`http://www.lss.supelec.fr/books/intervals`

Part II

Tools

2. Interval Analysis

2.1 Introduction

Before using interval analysis as a basic tool in the following chapters, we shall now introduce its main concepts. Section 2.2 recalls fundamental notions on set operators, set functions and set calculus. Section 2.3 then presents basic notions of interval analysis. Section 2.4 is dedicated to the important notion of inclusion function. Finally, Section 2.5 deals with the extension to intervals of logical tests that are almost invariably present in the algorithms of interest to us.

2.2 Operations on Sets

Interval computation is a special case of computation on sets, and set theory provides the foundations for interval analysis. The reader interested only in interval computation may skip this section and go directly to Section 2.3.

The operations on sets fall into two categories. The first one, considered in Section 2.2.1, consists of operations that have a meaning only in a set-theoretic context (such as union, intersection, Cartesian product...). The second one, considered in Section 2.2.2, consists of the extension to sets of operations that are already defined for numbers (or vectors).

2.2.1 Purely set-theoretic operations

Consider two sets $\mathbb{X}$ and $\mathbb{Y}$. Their *intersection* is

$$\mathbb{X} \cap \mathbb{Y} \triangleq \{x \mid x \in \mathbb{X} \text{ and } x \in \mathbb{Y}\}, \tag{2.1}$$

and their *union* is

$$\mathbb{X} \cup \mathbb{Y} \triangleq \{x \mid x \in \mathbb{X} \text{ or } x \in \mathbb{Y}\}. \tag{2.2}$$

$\mathbb{X}$ deprived of $\mathbb{Y}$ is defined by

$$\mathbb{X} \setminus \mathbb{Y} \triangleq \{x \mid x \in \mathbb{X} \text{ and } x \notin \mathbb{Y}\}. \tag{2.3}$$

The *Cartesian product* of $\mathbb{X}$ and $\mathbb{Y}$ is

L. Jaulin et al., *Applied Interval Analysis*

$$\mathbb{X} \times \mathbb{Y} \triangleq \{(x, y) \mid x \in \mathbb{X} \text{ and } y \in \mathbb{Y}\}. \tag{2.4}$$

If $\mathbb{Z} = \mathbb{X} \times \mathbb{Y}$, then the *projection* of a subset $\mathbb{Z}_1$ of $\mathbb{Z}$ onto $\mathbb{X}$ (with respect to $\mathbb{Y}$) is defined as

$$\text{proj}_{\mathbb{X}}(\mathbb{Z}_1) \triangleq \{x \in \mathbb{X} \mid \exists y \in \mathbb{Y} \text{ such that } (x, y) \in \mathbb{Z}_1\}.$$

These operations are illustrated by Figure 2.1.

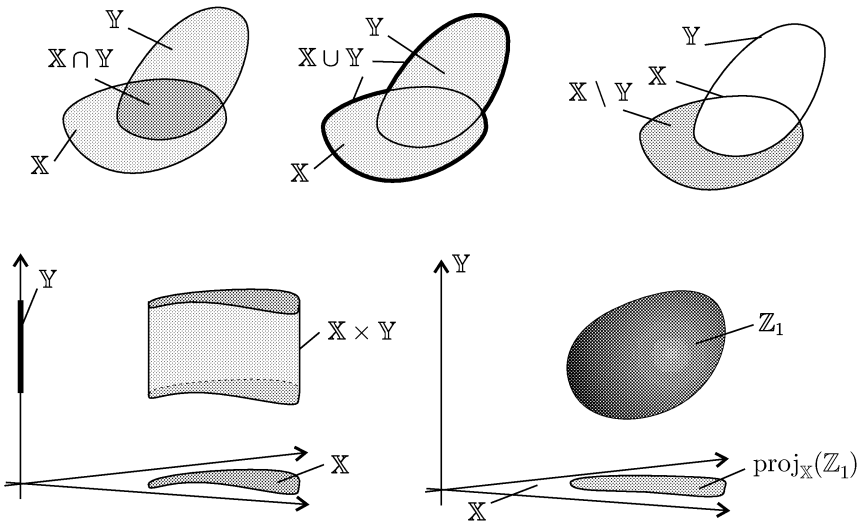


Fig. 2.1. Operations on sets

The *inclusion* of $\mathbb{X}$ in $\mathbb{Y}$ is defined by

$$\mathbb{X} \subset \mathbb{Y} \Leftrightarrow \forall x \in \mathbb{X}, x \in \mathbb{Y}, \tag{2.5}$$

and the *equality* of $\mathbb{X}$ and $\mathbb{Y}$ by

$$\mathbb{X} = \mathbb{Y} \Leftrightarrow \mathbb{X} \subset \mathbb{Y} \text{ and } \mathbb{Y} \subset \mathbb{X}. \tag{2.6}$$

2.2.2 Extended operations

Consider two sets $\mathbb{X}$ and $\mathbb{Y}$ and a function $f : \mathbb{X} \to \mathbb{Y}$. If $\mathbb{X}_1 \subset \mathbb{X}$, the *direct image* of $\mathbb{X}_1$ by f is

$$f(\mathbb{X}_1) \triangleq \{f(x) \mid x \in \mathbb{X}_1\}. \tag{2.7}$$

If $\mathbb{Y}_1 \subset \mathbb{Y}$, the *reciprocal image* of $\mathbb{Y}_1$ by f is

$$f^{-1}(\mathbb{Y}_1) \triangleq \{x \in \mathbb{X} \mid f(x) \in \mathbb{Y}_1\}. \tag{2.8}$$

If $\varnothing$ denotes the empty set, then the previous definitions imply that

$$f(\varnothing) = f^{-1}(\varnothing) = \varnothing. \tag{2.9}$$

It is trivial to show that if $\mathbb{X}_1$ and $\mathbb{X}_2$ are subsets of $\mathbb{X}$ and if $\mathbb{Y}_1$ and $\mathbb{Y}_2$ are subsets of $\mathbb{Y}$, then

$$\begin{aligned}
&f(\mathbb{X}_1 \cap \mathbb{X}_2) \subset f(\mathbb{X}_1) \cap f(\mathbb{X}_2),\\
&f(\mathbb{X}_1 \cup \mathbb{X}_2) = f(\mathbb{X}_1) \cup f(\mathbb{X}_2),\\
&f^{-1}(\mathbb{Y}_1 \cap \mathbb{Y}_2) = f^{-1}(\mathbb{Y}_1) \cap f^{-1}(\mathbb{Y}_2),\\
&f^{-1}(\mathbb{Y}_1 \cup \mathbb{Y}_2) = f^{-1}(\mathbb{Y}_1) \cup f^{-1}(\mathbb{Y}_2),\\
&f\left(f^{-1}(\mathbb{Y})\right) \subset \mathbb{Y},\\
&f^{-1}(f(\mathbb{X})) \supset \mathbb{X},\\
&\mathbb{X}_1 \subset \mathbb{X}_2 \Rightarrow f(\mathbb{X}_1) \subset f(\mathbb{X}_2),\\
&\mathbb{Y}_1 \subset \mathbb{Y}_2 \Rightarrow f^{-1}(\mathbb{Y}_1) \subset f^{-1}(\mathbb{Y}_2),\\
&\mathbb{X} \subset \mathbb{Y}_1 \times \mathbb{Y}_2 \Rightarrow \mathbb{X} \subset \mathrm{proj}_{\mathbb{Y}_1}(\mathbb{X}) \times \mathrm{proj}_{\mathbb{Y}_2}(\mathbb{X}).
\end{aligned} \tag{2.10}$$

In the same manner, it is possible to extend operations on numbers (or vectors) to operations on sets. Denote by $\mathcal{P}(\mathbb{X})$ the *power set* of $\mathbb{X}$, *i.e.*, the set of all subsets of $\mathbb{X}$. Let $\diamond$ be a binary operator from $\mathbb{X} \times \mathbb{Y}$ to $\mathbb{Z}$. It can be extended as a set operator as follows:

$$\mathbb{X}_1 \diamond \mathbb{Y}_1 \triangleq \{x_1 \diamond y_1 \mid x_1 \in \mathbb{X}_1, y_1 \in \mathbb{Y}_1\}, \tag{2.11}$$

where $\diamond$ is now an operator from $\mathcal{P}(\mathbb{X} \times \mathbb{Y})$ to $\mathcal{P}(\mathbb{Z})$. For instance, if $\mathbb{X}_1$ and $\mathbb{Y}_1$ are subsets of $\mathbb{R}^n$, then

$$\mathbb{X}_1 + \mathbb{Y}_1 = \{\mathbf{x} + \mathbf{y} \mid \mathbf{x} \in \mathbb{X}_1, \mathbf{y} \in \mathbb{Y}_1\}, \tag{2.12}$$

$$\mathbb{X}_1 - \mathbb{Y}_1 = \{\mathbf{x} - \mathbf{y} \mid \mathbf{x} \in \mathbb{X}_1, \mathbf{y} \in \mathbb{Y}_1\}. \tag{2.13}$$

Note that the set $\mathbb{X}_1 - \mathbb{X}_1 = \{\mathbf{x} - \mathbf{y} \mid \mathbf{x} \in \mathbb{X}_1, \mathbf{y} \in \mathbb{X}_1\}$ should not be confused with the set $\{\mathbf{x} - \mathbf{x} \mid \mathbf{x} \in \mathbb{X}_1\} = \{\mathbf{0}\}$.

When the operator $\diamond$ applies to an element x_1 of $\mathbb{X}$ together with a subset $\mathbb{Y}_1$ of $\mathbb{Y}$, x_1 is cast into the singleton $\{x_1\}$, so

$$x_1 \diamond \mathbb{Y}_1 \triangleq \{x_1\} \diamond \mathbb{Y}_1 = \{x_1 \diamond y_1 \mid y_1 \in \mathbb{Y}_1\}. \tag{2.14}$$

For instance, if $\mathbb{D}$ is the disk of $\mathbb{R}^2$ with centre $\mathbf{c}$ and radius 4, then $3 * \mathbb{D}$ is the disk of $\mathbb{R}^2$ with centre $3\mathbf{c}$ and radius 12. If $\mathbb{Y}_1$ is also a singleton $\{y_1\}$, then

$$x_1 \diamond y_1 = \{x_1\} \diamond \{y_1\} = \{x_1 \diamond y_1\}, \tag{2.15}$$

and the usual rules of $\diamond$ for punctual arguments apply.

2.2.3 Properties of set operators

Some properties of operators acting on numbers extend to their set counterparts. Consider, as an illustration, a set $\mathbb{X}$ equipped with the binary operator $\diamond$. Assume that $\mathbb{X}$ is *closed* with respect to $\diamond$ (*i.e.*, if x and y belong to $\mathbb{X}$,

then $x \diamond y$ belongs to $\mathbb{X}$). Assume also that $\diamond$ has been extended to $\mathcal{P}(\mathbb{X})$ as described in the previous section. Some properties true for $(\mathbb{X}, \diamond)$ remain true for $(\mathcal{P}(\mathbb{X}), \diamond)$. For instance, if $\diamond$ is commutative for $\mathbb{X}$, it is also commutative for $\mathcal{P}(\mathbb{X})$, *i.e.*,

$$\begin{aligned} &(\forall x_1 \in \mathbb{X}, \forall x_2 \in \mathbb{X}, x_1 \diamond x_2 = x_2 \diamond x_1) \\ \Rightarrow\ &(\forall \mathbb{X}_1 \in \mathcal{P}(\mathbb{X}), \forall \mathbb{X}_2 \in \mathcal{P}(\mathbb{X}), \mathbb{X}_1 \diamond \mathbb{X}_2 = \mathbb{X}_2 \diamond \mathbb{X}_1)\,. \end{aligned} \tag{2.16}$$

If $\diamond$ is associative for $\mathbb{X}$, it is also associative for $\mathcal{P}(\mathbb{X})$, *i.e.*,

$$\begin{aligned} &\left(\forall\,(x_1, x_2, x_3) \in \mathbb{X}^3, x_1 \diamond (x_2 \diamond x_3) = (x_1 \diamond x_2) \diamond x_3\right) \\ \Rightarrow\ &\left(\forall\,(\mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3) \in (\mathcal{P}(\mathbb{X}))^3\,, \mathbb{X}_1 \diamond (\mathbb{X}_2 \diamond \mathbb{X}_3) = (\mathbb{X}_1 \diamond \mathbb{X}_2) \diamond \mathbb{X}_3\right). \end{aligned} \tag{2.17}$$

If $\diamond$ admits a unit element (denoted by 0) in $\mathbb{X}$, then it also admits a unit element in $\mathcal{P}(\mathbb{X})$, which is the singleton $\{0\}$.

On the other hand, some properties true for $(\mathbb{X}, \diamond)$ may become false for $(\mathcal{P}(\mathbb{X}), \diamond)$. For instance, if each element x of $(\mathbb{X}, \diamond)$ admits a symmetric element this is no longer true in $(\mathcal{P}(\mathbb{X}), \diamond)$. We only have

$$\begin{aligned} &(\forall x_1 \in \mathbb{X}, \exists y_1 \in \mathbb{X} \mid x_1 \diamond y_1 = 0) \\ \Rightarrow\ &(\forall \mathbb{X}_1 \in \mathcal{P}(\mathbb{X}), \exists \mathbb{Y}_1 \in \mathcal{P}(\mathbb{X}) \mid \mathbb{X}_1 \diamond \mathbb{Y}_1 \ni 0)\,. \end{aligned} \tag{2.18}$$

Thus, if $(\mathbb{X}, \diamond)$ is a group, $(\mathcal{P}(\mathbb{X}), \diamond)$ is only a monoid.

It is not our purpose here to give an exhaustive view of the extensions of operators on numbers to sets, but just to stress that special care should be exercised when dealing with set arithmetic. For instance, if $\mathbb{X}_1$ is a subset of a group $(\mathbb{X}, +)$, an expression such as

$$\mathbb{Z} = \mathbb{X}_1 - \mathbb{X}_1 \tag{2.19}$$

should not be interpreted as

$$\mathbb{Z} = \{\mathbf{x} - \mathbf{x} \mid \mathbf{x} \in \mathbb{X}_1\}\,, \tag{2.20}$$

but as

$$\mathbb{Z} = \{\mathbf{x} - \mathbf{y} \mid \mathbf{x} \in \mathbb{X}_1, \mathbf{y} \in \mathbb{X}_1\}, \tag{2.21}$$

see (2.13). To avoid such confusions, the use of algebraic manipulations on sets will be limited as much as possible in this book.

The following theorem will have important consequences in the context of interval computation.

Theorem 2.1 *Consider the function*

$$f : \begin{array}{l} \mathbb{X}(1) \times \cdots \times \mathbb{X}(n) \to \mathbb{Y} \\ (x_1, \ldots, x_n) \qquad\quad \mapsto y \end{array}$$

for which a formal expression involving operators and functions is available. Let $\mathbb{F}(\mathbb{X}_1, \ldots, \mathbb{X}_n)$ *be a function from* $\mathcal{P}(\mathbb{X}(1)) \times \cdots \times \mathcal{P}(\mathbb{X}(n))$ *to* $\mathcal{P}(\mathbb{Y})$*. Assume that* $\mathbb{F}(\mathbb{X}_1, \ldots, \mathbb{X}_n)$ *has the same formal expression as* $f(\mathbb{X}_1, \ldots, \mathbb{X}_n)$*. Then*

$$f(\mathbb{X}_1, \ldots, \mathbb{X}_n) \subset \mathbb{F}(\mathbb{X}_1, \ldots, \mathbb{X}_n), \tag{2.22}$$

where

$$f(\mathbb{X}_1, \ldots, \mathbb{X}_n) = \{f(x_1, \ldots, x_n) \mid x_1 \in \mathbb{X}_1, \ldots, x_n \in \mathbb{X}_n\}.$$

Moreover, if each x_i *occurs at most once in the formal expression of* f*, then*

$$f(\mathbb{X}_1, \ldots, \mathbb{X}_n) = \mathbb{F}(\mathbb{X}_1, \ldots, \mathbb{X}_n). \tag{2.23}$$

■

A proof of Theorem 2.1 in the case where all functions involved in the formal expression of f are continuous and where the sets $\mathbb{X}_i$ are closed intervals may be found in Moore (1979). The extension to more general sets (as presented in Theorem 2.1) is a direct consequence of the propagation theorem (Jaulin, Kieffer, Braems and Walter, 2001), the principle of which will be recalled in Section 6.4.4, page 174.

In general, unfortunately, $\mathbb{F}(\mathbb{X}_1, \ldots, \mathbb{X}_n)$ is only an outer approximation of $f(\mathbb{X}_1, \ldots, \mathbb{X}_n)$, because of multiple occurrences of variables in the formal expression of f. Set computation is thus *pessimistic*, because of the *dependency effect* illustrated by the following example.

Example 2.1 *Consider the function* $f : \mathbb{R}^2 \to \mathbb{R}$*,* $f(x_1, x_2) = x_1 + x_2 - x_1$*. Then*

$$\begin{aligned} f(\mathbb{X}_1, \mathbb{X}_2) &= \{x_1 + x_2 - x_1 \mid x_1 \in \mathbb{X}_1, x_2 \in \mathbb{X}_2\} \\ &= \{x_2 \mid x_2 \in \mathbb{X}_2\} = \mathbb{X}_2, \end{aligned} \tag{2.24}$$

and

$$\begin{aligned} \mathbb{F}(\mathbb{X}_1, \mathbb{X}_2) &= \mathbb{X}_1 + \mathbb{X}_2 - \mathbb{X}_1 \\ &= \{x_1 + x_2 - x_3 \mid x_1 \in \mathbb{X}_1, x_2 \in \mathbb{X}_2, x_3 \in \mathbb{X}_1\}. \end{aligned} \tag{2.25}$$

It is clear that $f(\mathbb{X}_1, \mathbb{X}_2) \subset \mathbb{F}(\mathbb{X}_1, \mathbb{X}_2)$*. The dependency between* x_1 *and* x_3 *(*$x_1 = x_3$*) has been forgotten in (2.25), thus adding one degree of freedom in the elaboration of the set* $\mathbb{F}(\mathbb{X}_1, \mathbb{X}_2)$*; hence the pessimism.* ■

Note that multiple occurrences of variables associated to singletons do not cause pessimism.

2.2.4 Wrappers

Consider a set $\mathbb{X}$ and a set $\mathbb{IX}$ of subsets of $\mathbb{X}$. $\mathbb{IX}$ is a *set of wrappers* for $\mathbb{X}$ if $\mathbb{X}$ and each singleton of $\mathbb{X}$ belong to $\mathbb{IX}$ and if $\mathbb{IX}$ is closed by intersection (*i.e.*, if $\mathbb{X}_1 \in \mathbb{IX}$ and $\mathbb{X}_2 \in \mathbb{IX}$, then $\mathbb{X}_1 \cap \mathbb{X}_2 \in \mathbb{IX}$). The empty set $\varnothing$ must thus belong to $\mathbb{IX}$, unless $\mathbb{X}$ is a singleton.

Example 2.2 *A set of wrappers for* $\mathbb{X} = \{a, b, c, d\}$ *is*

$$\mathbb{IX} = \{\varnothing, a, b, c, \{a, b\}, \{a, d\}, \{a, b, c, d\}\}. \tag{2.26}$$

■

Let $\mathbb{X}_1$ be a subset of $\mathbb{X}$; the *smallest wrapper* $[\mathbb{X}_1]$ of $\mathbb{X}_1$ is the smallest element of $\mathbb{IX}$ containing $\mathbb{X}_1$:

$$[\mathbb{X}_1] = \bigcap \{\mathbb{Y} \in \mathbb{IX} \mid \mathbb{X}_1 \subset \mathbb{Y}\}. \tag{2.27}$$

Example 2.3 *The set* $\mathcal{P}(\mathbb{X})$ *of all subsets of* $\mathbb{X}$ *is a set of wrappers for* $\mathbb{X}$. *If* $\mathbb{X}_1$ *is a subset of* $\mathbb{X}$, *then* $[\mathbb{X}_1] = \mathbb{X}_1$. ■

Example 2.4 *When* $\mathbb{X} = \mathbb{R}^n$, *the set* $\mathbb{IX}$ *of all convex sets of* $\mathbb{R}^n$ *is a set of wrappers for* $\mathbb{X}$. $[\mathbb{X}_1]$ *is then the convex hull of* $\mathbb{X}_1$. ■

In practice, *wrappers* should be simple enough to allow the computation of outer approximations of sets. The wrappers to be considered in this book are intervals of $\mathbb{R}$, axis-aligned boxes of $\mathbb{R}^n$, subsets of the Boolean set {true, false} and unions of intervals or boxes. Other types of wrapper could be used, such as parallelotopes, zonotopes, convex sets or unions of such sets. Although they are not closed with respect to intersection, ellipsoids can also be used (Milanese et al., 1996).

Let $\diamond$ be a binary operator from $\mathbb{X} \times \mathbb{Y}$ to $\mathbb{Z}$. Let $\mathbb{IX}$, $\mathbb{IY}$ and $\mathbb{IZ}$ be sets of wrappers for $\mathbb{X}$, $\mathbb{Y}$ and $\mathbb{Z}$. The operator $\diamond$ can be extended to these sets of wrappers as follows. If $\mathbb{X}_1 \in \mathbb{IX}$ and $\mathbb{Y}_1 \in \mathbb{IY}$, then

$$\mathbb{X}_1 \ [\diamond] \ \mathbb{Y}_1 \triangleq [\{x_1 \diamond y_1 \mid x_1 \in \mathbb{X}_1, y_1 \in \mathbb{Y}_1\}], \tag{2.28}$$

where the wrapping operator [.] on the right-hand side is defined as in (2.27). Because of the definition of $\mathbb{X}_1 \diamond \mathbb{Y}_1$ given in (2.11),

$$\mathbb{X}_1 \ [\diamond] \ \mathbb{Y}_1 \supset \mathbb{X}_1 \diamond \mathbb{Y}_1. \tag{2.29}$$

Consider two sets $\mathbb{X}$ and $\mathbb{Y}$, a function $f : \mathbb{X} \to \mathbb{Y}$, and two sets of wrappers $\mathbb{IX}$ and $\mathbb{IY}$ for $\mathbb{X}$ and $\mathbb{Y}$ respectively. If $\mathbb{X}_1 \in \mathbb{IX}$, then f can be extended to wrappers as

$$[f](\mathbb{X}_1) \triangleq [\{f(x) \mid x \in \mathbb{X}_1\}]. \tag{2.30}$$

Again,

$$[f](\mathbb{X}_1) \supset f(\mathbb{X}_1). \tag{2.31}$$

Consider three sets $\mathbb{X}$, $\mathbb{Y}$, $\mathbb{Z}$, their sets of wrappers $\mathbb{IX}$, $\mathbb{IY}$, $\mathbb{IZ}$ and three functions $f : \mathbb{X} \to \mathbb{Y}$, $g : \mathbb{Y} \to \mathbb{Z}$, $h = g \circ f$ (where $\circ$ denotes the composition operator). Then, $g(f(\mathbb{X}_1)) = h(\mathbb{X}_1)$, but one can only write that

$$[g]([f](\mathbb{X}_1)) \supset [h](\mathbb{X}_1). \tag{2.32}$$

This is known as the *wrapping effect*, illustrated by Figure 2.2 when the wrappers are convex sets of $\mathbb{R}^2$.

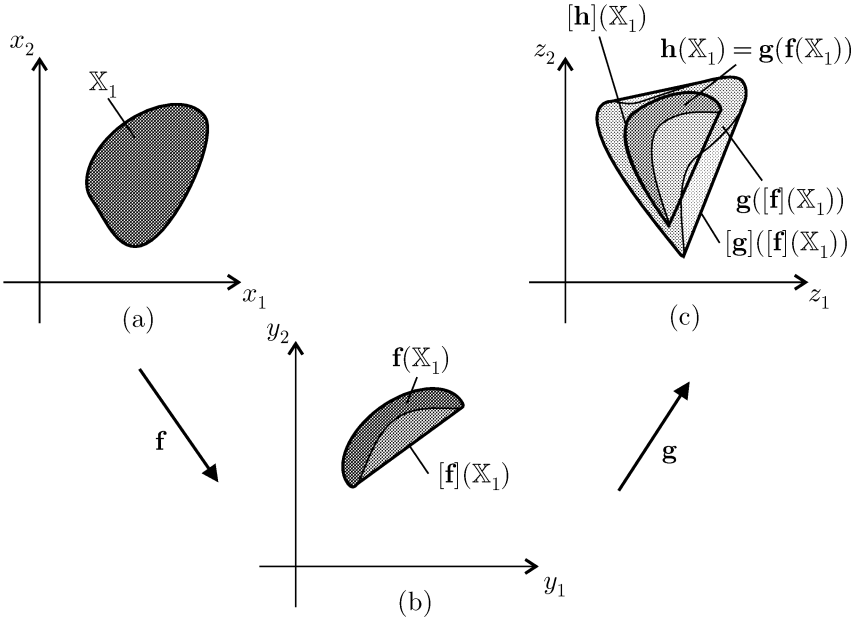


Fig. 2.2. Wrapping effect when the wrappers are the convex sets of $\mathbb{R}^2$; (a) $\mathbb{X}_1$ is convex and thus a wrapper; (b) $\mathbf{f}(\mathbb{X}_1)$, in dark grey, is not convex; its convex hull is the wrapper $[\mathbf{f}](\mathbb{X}_1)$; (c) since $\mathbf{f}(\mathbb{X}_1) \subset [\mathbf{f}](\mathbb{X}_1)$, we have $\mathbf{g}(\mathbf{f}(\mathbb{X}_1)) \subset \mathbf{g}([\mathbf{f}](\mathbb{X}_1)) \subset [\mathbf{g}]([\mathbf{f}](\mathbb{X}_1))$; therefore, $[\mathbf{g}]([\mathbf{f}](\mathbb{X}_1))$ is a convex set that contains the convex hull $[\mathbf{h}](\mathbb{X}_1)$ of $\mathbf{g}(\mathbf{f}(\mathbb{X}_1))$; the undesirable points of $[\mathbf{g}]([\mathbf{f}](\mathbb{X}_1))$ that are outside $[\mathbf{h}](\mathbb{X}_1)$ are a direct consequence of the wrapping effect

As the dependency effect, the wrapping effect introduces pessimism when computing with wrappers. Some properties that held true for set computation are thus no longer true for computation with wrappers. This is the case for Theorem 2.1; when wrappers are used, the inclusion property (2.22) is still satisfied, but does not necessarily transform into the equality (2.23) when each input variable occurs at most once in the formal expression of f.

Remark 2.1 *In this book, computation will be with wrappers and not with generic sets. When this creates no ambiguity,* $\mathbb{X}_1 \,[\diamond]\, \mathbb{Y}_1$ *will be shortened into* $\mathbb{X}_1 \diamond \mathbb{Y}_1$. ■

2.3 Interval Analysis

In interval analysis, the wrappers to be used are intervals when dealing with $\mathbb{R}$ and axis-aligned boxes when dealing with $\mathbb{R}^n$. Using the concepts of set computation recalled in Section 2.2, we shall now present interval computation.

2.3.1 Intervals

An *interval real* $[x]$ is a connected subset of $\mathbb{R}$. Even when the interval is not closed, we shall keep to the notation $[x]$. When no confusion may arise, $[x]$ will often merely be called an *interval*. Whether the empty set $\varnothing$ should be considered as an interval is still a subject of discussion. We choose to answer positively, if only to ensure that the set of intervals is closed with respect to intersection, and also because $\varnothing$ represents the absence of solution of a problem. The *lower bound* $\mathrm{lb}([x])$ of an interval $[x]$, also denoted by $\underline{x}$, is defined as

$$\underline{x} = \mathrm{lb}([x]) \triangleq \sup\{a \in \mathbb{R} \cup \{-\infty, \infty\} \mid \forall x \in [x], a \leqslant x\}. \tag{2.33}$$

Its *upper bound* $\mathrm{ub}([x])$, also denoted by $\overline{x}$, is defined as

$$\overline{x} = \mathrm{ub}([x] \triangleq \inf\{b \in \mathbb{R} \cup \{-\infty, \infty\} \mid \forall x \in [x], x \leqslant b\}. \tag{2.34}$$

Thus, $\underline{x}$ is the largest number on the left of $[x]$ and $\overline{x}$ is the smallest number on its right. For instance if $[x] =]-3, 7]$ then $\underline{x} = -3$ and $\overline{x} = 7$; if $[x] =]-\infty, \infty[$ then $\underline{x} = -\infty$ and $\overline{x} = \infty$. The *width* of any non-empty interval $[x]$ is

$$w([x]) \triangleq \overline{x} - \underline{x}, \tag{2.35}$$

so $w(]3, \infty[) = \infty$. The *midpoint* (or *centre*) of any bounded and non-empty interval $[x]$ is defined as

$$\mathrm{mid}([x]) \triangleq \frac{\underline{x} + \overline{x}}{2}. \tag{2.36}$$

The set-theoretic operations of Section 2.2.1 can be applied to intervals. The *intersection* of two intervals $[x]$ and $[y]$, defined by

$$[x] \cap [y] \triangleq \{z \in \mathbb{R} \mid z \in [x] \text{ and } z \in [y]\}, \tag{2.37}$$

is always an interval. This is not the case for their *union*

$$[x] \cup [y] \triangleq \{z \in \mathbb{R} \mid z \in [x] \text{ or } z \in [y]\}. \tag{2.38}$$

To make the set of intervals closed with respect to union, define the *interval hull* of a subset $\mathbb{X}$ of $\mathbb{R}$ as the smallest interval $[\mathbb{X}]$ that contains it. This is consistent with (2.27). For instance, the interval hull of $]2, 3] \cup [5, 7]$ is the interval $]2, 7]$. Define the *interval union* of $[x]$ and $[y]$, denoted by $[x] \sqcup [y]$, as the interval hull of $[x] \cup [y]$, *i.e.*,

$$[x] \sqcup [y] \triangleq [[x] \cup [y]]. \tag{2.39}$$

In the same manner,

$$[x] \ [\backslash] \ [y] = [[x] \setminus [y]] = [\{x \in [x] \mid x \notin [y]\}]. \tag{2.40}$$

For instance, $[0, 5[\ [\backslash] \ ]3, 4[= [0, 5[$ and $[0, 5[\ [\backslash] \ ]3, 5[= [0, 3]$. The Cartesian product of two intervals is not an interval but a box of $\mathbb{R}^2$; it thus corresponds to an external operation, to be treated in Section 2.3.4.

2.3.2 Interval computation

The four classical operations of real arithmetic, namely addition ($+$), subtraction ($-$), multiplication ($*$) and division ($/$) can be extended to intervals. For any such binary operator, denoted by $\diamond$, performing the operation associated with $\diamond$ on the intervals $[x]$ and $[y]$ means computing

$$[x] \diamond [y] = [\{x \diamond y \in \mathbb{R} \mid x \in [x], y \in [y]\}], \tag{2.41}$$

which is a direct consequence of (2.28). For instance

$$(]1, 2.2] * [0, 2[) +]1, 3] = [0, 4.4[+]1, 3] =]1, 7.4[, \tag{2.42}$$

$$1/[-2, 2[=] - \infty, \infty[, \tag{2.43}$$

$$[3, 4]/[0, 0] = \varnothing. \tag{2.44}$$

The rule (2.41) was first presented in the context of bounded and closed intervals by Moore (1959) and then extended to open-ended unbounded intervals (Hanson, 1968; Kahan, 1968; Davis, 1987).

Remark 2.2 *The result of (2.44) is based on a mathematical interpretation of the interval* $[0, 0]$*. We shall see in Chapter 10 that two types of zero are distinguished in the floating-point representation to be used when implementing interval computation, which sometimes allow different results.* ■

Remark 2.3 *Definition (2.41) differs from that given for sets in (2.11) because it is required that* $\diamond$ *returns an interval and not a possibly non-connected subset of* $\mathbb{R}$*. Because of wrapping, we should have written* $[x]$ $[\diamond]$ $[y]$ *instead of* $[x] \diamond [y]$ *but, as for generic wrappers, we shall keep to the simpler notation* $[x] \diamond [y]$*. Note that (2.41) and (2.11) match when* $\diamond$ *is continuous, as the set* $\{x \diamond y \in \mathbb{R} \mid x \in [x], y \in [y]\}$ *is then an interval.* ■

Remark 2.4 *The operator* $\diamond$ *defined for intervals can also be extended to unions of intervals (also called* discontinuous intervals*) (Hyvönen, 1992). For instance, one may write*

$$1/[-2, 2[=] - \infty, -1/2[\cup [1/2, \infty[, \tag{2.45}$$

instead of $]-\infty, \infty[$ *as in (2.43). The computation on unions of intervals may, however, lead to an explosion of the number of subintervals to be handled and will no longer be considered in this book.* ■

As already mentioned for generic sets, the properties of the basic operators for intervals differ from their properties in $\mathbb{R}$. For instance, $[x]-[x]$ is generally not equal to $[0, 0]$. This is because $[x] - [x] = \{x - y \mid x \in [x], y \in [x]\}$, rather than $\{x - x \mid x \in [x]\}$. The subtraction thus does not take the dependency of the two occurrences of $[x]$ into account. Addition and multiplication remain associative and commutative, but multiplication is no longer distributive with respect to addition. Instead,

$$[x] * ([y] + [z]) \subset [x] * [y] + [x] * [z], \tag{2.46}$$

a property known as *subdistributivity*, which is a direct consequence of the dependency effect, as $[x]$ appears only once on the left-hand side but twice on the right-hand side. As a result, it is recommended to factorize expanded forms as much as possible.

Elementary functions such as exp, tan, sin, cos... extend to intervals as indicated by (2.30). If f is a function from $\mathbb{R}$ to $\mathbb{R}$, then its interval counterpart $[f]$ satisfies

$$[f]([x]) = [\{f(x) \mid x \in [x]\}]\,. \tag{2.47}$$

For any continuous elementary function, $[f]([x])$ is thus equal to the image set $f([x])$. For instance,

$$\begin{aligned}
&[\text{arctan}]([0,\infty[) = [0,\pi/2[,\\
&[\text{sqr}]([-1,3[) = [0,9[,\\
&[\text{exp}]([0,1[) = [\text{exp}(0),\text{exp}(1)[= [1,e[,\\
&[\text{sqrt}]([4,25]) = [\text{sqrt}(4),\ \text{sqrt}(25)] = [2,5],\\
&[\text{sqrt}]([-25,-4]) = \varnothing,\\
&[\text{sqrt}]([-50,1]) = [\text{sqrt}]([0,1]) = [0,1],\\
&[\text{ln}]([-50,1[) =]-\infty,0[\,.
\end{aligned} \tag{2.48}$$

2.3.3 Closed intervals

This section presents rules for computing on subsets of $\mathbb{R}$ in the special case where the wrappers are closed intervals of $\mathbb{R}$. Denote by $\mathbb{IR}$ the set of all such closed intervals. Since $\mathbb{R}$ and $\varnothing$ are both open and closed, they both belong to $\mathbb{IR}$, and any element of $\mathbb{IR}$ can be written in one of the following forms: $[a,b]$, $]-\infty,a]$, $[a,\infty[$, $]-\infty,\infty[$ or $\varnothing$, where a and b are real numbers such that $a \leqslant b$. Any $[x]$ of $\mathbb{IR}$ can be specified in a unique way by its lower bound $\underline{x}$ and its upper bound $\overline{x}$. For simplicity, we shall often write $[x] = [\underline{x},\overline{x}]$, even if bounds may be infinite. Thus, $[0,\infty]$ should be interpreted as $[0,\infty[$. Note the dual nature of closed intervals, which may be viewed as *sets* (on which standard set operations apply), and as *couples of elements* of $\mathbb{R}$ on which an arithmetic can be built. Couples of the form $[\infty,\infty]$, $[-\infty,-\infty]$ and $[a,b]$ with $a > b$ do not correspond to intervals (see *modal interval arithmetic* (Gardenes et al., 1985), however, where the notation $[a,b]$ with $a > b$ receives a meaning). When $\underline{x}$ and $\overline{x}$ are equal, the interval $[x]$ is said to be *punctual* (or degenerate). Any real number could thus be represented as a punctual interval and vice versa.

The operations of Section 2.3.2 can be redefined, in the context of closed intervals, as operations on their bounds: the bounds of the result of an interval operation are expressed as functions of the bounds of its interval arguments. The remainder of this section is devoted to illustrating how this can be done.

The interval union of two non-empty closed intervals $[x]$ and $[y]$, defined by (2.39), satisfies

$$\forall [x] \in \mathbb{IR}, \forall [y] \in \mathbb{IR}, \quad [x] \sqcup [y] = \left[\min\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}\right]. \tag{2.49}$$

The intersection of two non-empty closed intervals $[x]$ and $[y]$, defined by (2.37), satisfies

$$\begin{aligned} [x] \cap [y] &= \left[\max\{\underline{x}, \underline{y}\}, \min\{\overline{x}, \overline{y}\}\right] \text{ if } \max\{\underline{x}, \underline{y}\} \leqslant \min\{\overline{x}, \overline{y}\}, \\ &= \varnothing \text{ otherwise.} \end{aligned} \tag{2.50}$$

If α is a real number and $[x]$ a non-empty interval, then the interval

$$\alpha[x] \triangleq \{\alpha x \mid x \in [x]\} \tag{2.51}$$

is given by

$$\begin{aligned} \alpha[x] &= [\alpha\underline{x}, \alpha\overline{x}] \quad \text{if } \alpha \geqslant 0 \\ &= [\alpha\overline{x}, \alpha\underline{x}] \quad \text{if } \alpha < 0. \end{aligned} \tag{2.52}$$

For non-empty closed intervals,

$$\begin{aligned} [x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \\ [x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}], \\ [x] * [y] &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]. \end{aligned} \tag{2.53}$$

The product of two intervals will be denoted indifferently by $[x] * [y]$, or $[x][y]$.

For division, (2.41) leads to

$$\begin{aligned} 1/[y] &= \varnothing && \text{if } [y] = [0, 0], \\ &= [1/\overline{y}, 1/\underline{y}] && \text{if } 0 \notin [y], \\ &= [1/\overline{y}, \infty[&& \text{if } \underline{y} = 0 \text{ and } \overline{y} > 0, \\ &=]-\infty, 1/\underline{y}] && \text{if } \underline{y} < 0 \text{ and } \overline{y} = 0, \\ &=]-\infty, \infty[&& \text{if } \underline{y} < 0 \text{ and } \overline{y} > 0, \end{aligned} \tag{2.54}$$

and

$$[x]/[y] = [x] * (1/[y]). \tag{2.55}$$

Of course, when applied to punctual intervals $[x]$ and $[y]$, the previous rules simplify into the usual rules of real arithmetic, which is why interval arithmetic can claim to be an extension of the latter.

Elementary interval functions can also be expressed in terms of bounds. For instance, for any non-empty $[x]$,

$$[\exp]([x]) = [\exp(\underline{x}), \exp(\overline{x})]. \tag{2.56}$$

For non-monotonic functions, the situation is more complicated. For example, $[\sin]([0, \pi]) = [0, 1]$ differs from the interval $[\sin(0), \sin(\pi)] = [0, 0]$. Specific

Table 2.1. Algorithm for the interval evaluation of the sine function

Algorithm sin(in: $[x]$; out: $[r]$)	
1	if $\exists k \in \mathbb{Z} \mid 2k\pi - \pi/2 \in [x]$ then $\underline{r} = -1$;
2	else $\underline{r} = \min(\sin \underline{x}, \sin \overline{x})$;
3	if $\exists k \in \mathbb{Z} \mid 2k\pi + \pi/2 \in [x]$ then $\overline{r} = 1$;
4	else $\overline{r} = \max(\sin \underline{x}, \sin \overline{x})$.

algorithms must therefore be built. An algorithm for the sine function is given by Table 2.1.

Figure 2.3 illustrates the computation of $\sin([2.6, 7.2]) = [-1, 0.7937]$. Inclusion functions for other trigonometric functions are obtained in the same way or by expressing them as functions of the sine function. The hyperbolic functions receive a similar treatment.

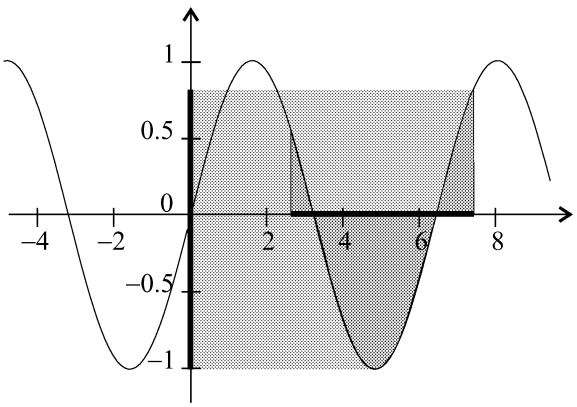


Fig. 2.3. Computation of $\sin([x])$

Remark 2.5 *Most often, an operation or the evaluation of a function with an empty interval argument should yield an empty result. Obvious exceptions to this rule are for the union operator* $\cup$ *and interval union operator* $\sqcup$*, as* $[x] \cup \varnothing = [x]$ *and* $[x] \sqcup \varnothing = [x]$ ■

As already mentioned, the arithmetical rules for intervals differ from those for real numbers. For instance, $x^2 - x = (x - \frac{1}{2})^2 - \frac{1}{4}$ whereas $[x]^2 - [x]$ differs from $([x] - \frac{1}{2})^2 - \frac{1}{4}$, as illustrated by the following example.

Example 2.5 *At* $[x] = [-1, 3]$,

$$[x]^2 - [x] = [-1, 3]^2 - [-1, 3] = [0, 9] + [-3, 1] = [-3, 10], \tag{2.57}$$

$$\left([x] - \tfrac{1}{2}\right)^2 - \tfrac{1}{4} = \left[-\tfrac{3}{2}, \tfrac{5}{2}\right]^2 - \tfrac{1}{4} = \left[0, \tfrac{25}{4}\right] - \tfrac{1}{4} = \left[-\tfrac{1}{4}, 6\right]. \quad (2.58)$$

The first result is a pessimistic approximation of the image set of $x^2 - x$ *at* $[-1, 3]$*, whereas the second one is equal to this image set (see Theorem 2.1).* ■

One is therefore well advised to transform expressions in such a way as to reduce the number of occurrences of each variable as much as possible, which is sometimes easier said than done.

Since computation on closed intervals reduces to computation on their bounds, traditional interval software considers only closed intervals (except packages such as INC++ or PROLOG 4, which also support open and half-open intervals, at the cost of a much more complex implementation).

2.3.4 Interval vectors

An *interval real vector* $[\mathbf{x}]$ is a subset of $\mathbb{R}^n$ that can be defined as the Cartesian product of n closed intervals. When there is no ambiguity, $[\mathbf{x}]$ will simply be called an interval vector, or a *box*. It will be written as

$$[\mathbf{x}] = [x_1] \times [x_2] \times \cdots \times [x_n], \text{ with } [x_i] = [\underline{x}_i, \overline{x}_i] \text{ for } i = 1, \ldots, n. \quad (2.59)$$

Its ith *interval component* $[x_i]$ is the projection of $[\mathbf{x}]$ onto the ith axis. The empty set of $\mathbb{R}^n$ should thus be written as $\varnothing \times \cdots \times \varnothing$ because all of its interval components are empty. Expressions such as

$$[\mathbf{x}] = \varnothing \times [0, 1] \quad (2.60)$$

are therefore prohibited, because $[0, 1]$ is not the projection of $[\mathbf{x}]$ onto the second axis. This guarantees the uniqueness of notation of a given box. The set of all n-dimensional boxes will be denoted by $\mathbb{IR}^n$. Non-empty boxes are n-dimensional axis-aligned parallelepipeds. Figure 2.4 illustrates the case $n = 2$, with $[\mathbf{x}] = [x_1] \times [x_2]$.

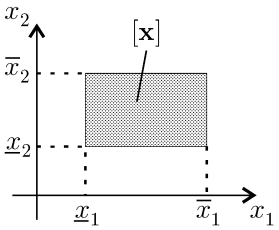


Fig. 2.4. A box $[\mathbf{x}]$ of $\mathbb{IR}^2$

Many of the notions introduced in Section 2.3.2 for intervals extend without difficulty to boxes. For instance, a box will be said to be punctual if *all* its interval components are. Any box with at least one punctual component

has a zero volume, so a box with a zero volume may not be punctual. The *lower bound* $\mathrm{lb}([\mathbf{x}])$ of a box $[\mathbf{x}]$ is the punctual vector consisting of the lower bounds of its interval components:

$$\underline{\mathbf{x}} = \ \mathrm{lb}([\mathbf{x}]) \triangleq (\mathrm{lb}([x_1]), \mathrm{lb}([x_2]), \cdots, \mathrm{lb}([x_n]))^{\mathrm{T}} = (\underline{x}_1, \underline{x}_2, \cdots, \underline{x}_n)^{\mathrm{T}}.$$

Similarly, the *upper bound* $\mathrm{ub}([\mathbf{x}])$ of $[\mathbf{x}]$ is the punctual vector

$$\overline{\mathbf{x}} = \mathrm{ub}([\mathbf{x}]) \triangleq (\mathrm{ub}([x_1]), \mathrm{ub}([x_2]), \cdots, \mathrm{ub}([x_n]))^{\mathrm{T}} = (\overline{x}_1, \overline{x}_2, \cdots, \overline{x}_n)^{\mathrm{T}}.$$

The *width* of $[\mathbf{x}] = ([x_1], [x_2], \ldots, [x_n])^{\mathrm{T}}$ is

$$w([\mathbf{x}]) \triangleq \max_{1 \leqslant i \leqslant n} w([x_i]). \tag{2.61}$$

If $[\mathbf{x}]$ is bounded and non-empty, then its *midpoint* (or *centre*) is

$$\mathrm{mid}([\mathbf{x}]) \triangleq (\mathrm{mid}([x_1]), \ldots, \mathrm{mid}([x_n]))^{\mathrm{T}}. \tag{2.62}$$

The *interval hull* of a subset $\mathbb{A}$ of $\mathbb{R}^n$ is the smallest box of $\mathbb{IR}^n$ that contains $\mathbb{A}$, denoted by $[\mathbb{A}]$.

The intersection of the boxes $[\mathbf{x}]$ and $[\mathbf{y}]$ of $\mathbb{IR}^n$ satisfies

$$[\mathbf{x}] \cap [\mathbf{y}] \triangleq ([x_1] \cap [y_1]) \times \ldots \times ([x_n] \cap [y_n]), \tag{2.63}$$

provided that $[\mathbf{x}] \cap [\mathbf{y}]$ is non-empty. Applying this rule for the intersection of $[1, 3] \times [-1, 2]$ and $[2, 4] \times [3, 7]$ would lead to $[2, 3] \times \varnothing$, which is actually correct but inconsistent with the convention that $[2, 3]$ should then be the projection of the result onto the first axis.

Most often the union of two boxes $[\mathbf{x}]$ and $[\mathbf{y}]$ is not a box. A favourable case is when there exists a unique $i \in \{1, \ldots, n\}$ such that $[x_j] = [y_j]$ for all $j \neq i$ and $[x_i] \cap [y_i] \neq \varnothing$, because then $[\mathbf{x}] \cup [\mathbf{y}] = ([x_1], \ldots, [x_{i-1}], [x_i] \cup [y_i], [x_{i+1}], \ldots, [x_n])^{\mathrm{T}}$. This seemingly unlikely event turns out to be frequently encountered with some of the algorithms to be considered in this book. In all other cases, the interval hull $[[\mathbf{x}] \cup [\mathbf{y}]]$ of the union of $[\mathbf{x}]$ and $[\mathbf{y}]$ can be computed as

$$[\mathbf{x}] \sqcup [\mathbf{y}] \triangleq ([x_1] \sqcup [y_1]) \times \ldots \times ([x_n] \sqcup [y_n]), \tag{2.64}$$

a process that extends to any number of boxes, some of which may be the empty set. We also have

$$[\mathbf{x}] \subset [\mathbf{y}] \Leftrightarrow [x_1] \subset [y_1] \text{ and } \ldots \text{ and } [x_n] \subset [y_n], \tag{2.65}$$

and

$$\mathbf{x} \in [\mathbf{y}] \Leftrightarrow x_1 \in [y_1] \text{ and } \ldots \text{ and } x_n \in [y_n]. \tag{2.66}$$

Classical operations for interval vectors are direct extensions of the same operations for punctual vectors. For instance, if $[\mathbf{x}]$ and $[\mathbf{y}]$ are boxes of $\mathbb{IR}^n$, and if α is a real number, then

$$\begin{aligned}
&\alpha[\mathbf{x}] \triangleq (\alpha[x_1]) \times \cdots \times (\alpha[x_n])\,,\\
&[\mathbf{x}]^{\mathrm{T}} * [\mathbf{y}] \triangleq [x_1] * [y_1] + \cdots + [x_n] * [y_n],\\
&[\mathbf{x}] + [\mathbf{y}] \triangleq ([x_1] + [y_1]) \times \cdots \times ([x_n] + [y_n])\,.
\end{aligned} \tag{2.67}$$

These definitions are consistent with the more general set operation of (2.11), page 13.

2.3.5 Interval matrices

Let $\mathbb{R}^{m\times n}$ be the set of all matrices with real coefficients, m rows and n columns. An $(m \times n)$-dimensional interval matrix is a subset of $\mathbb{R}^{m\times n}$ that can be defined as the Cartesian product of mn closed intervals. The interval matrix $[\mathbf{A}]$ will be written indifferently in any of the following forms:

$$\begin{aligned}
[\mathbf{A}] &= \begin{pmatrix} [a_{11}] & \dots & [a_{1n}] \\ \vdots & & \vdots \\ [a_{m1}] & \dots & [a_{mn}] \end{pmatrix}\\
&= [a_{11}] \times [a_{12}] \times \cdots \times [a_{mn}] = ([a_{ij}])_{1\leqslant i\leqslant m, 1\leqslant j\leqslant n}\,,
\end{aligned} \tag{2.68}$$

where $[a_{ij}] = [\underline{a}_{ij}, \overline{a}_{ij}]$ is the projection of $[\mathbf{A}]$ onto the (i,j)th axis. This convention makes unique the representation of the empty matrix. For instance,

$$[\mathbf{A}] = \begin{pmatrix} \varnothing & [0,1] \\ \varnothing & \varnothing \end{pmatrix} \tag{2.69}$$

is not allowed because the projection of the matrix represented by the Cartesian product $\varnothing \times [0,1] \times \varnothing \times \varnothing$ onto the $(1,2)$-axis is empty and not equal to $[0,1]$. The set of all $m \times n$ interval matrices is denoted by $\mathbb{IR}^{m\times n}$. An interval matrix is said to be punctual if all its entries are punctual. The *lower bound* $\mathrm{lb}([\mathbf{A}])$ of an interval matrix $[\mathbf{A}]$ is the punctual matrix made up with the lower bounds of its interval components:

$$\underline{\mathbf{A}} = \ \mathrm{lb}([\mathbf{A}]) \triangleq \begin{pmatrix} \underline{a}_{11} & \cdots & \underline{a}_{1n} \\ \vdots & & \vdots \\ \underline{a}_{m1} & \cdots & \underline{a}_{mn} \end{pmatrix}, \tag{2.70}$$

Similarly, its *upper bound* $\mathrm{ub}([\mathbf{A}])$ is the punctual matrix

$$\overline{\mathbf{A}} = \mathrm{ub}([\mathbf{A}]) \triangleq \begin{pmatrix} \overline{a}_{11} & \dots & \overline{a}_{1n} \\ \vdots & & \vdots \\ \overline{a}_{m1} & \dots & \overline{a}_{mn} \end{pmatrix}. \tag{2.71}$$

Its *width* $w([\mathbf{A}])$ is

$$w([\mathbf{A}]) \triangleq \max_{1\leqslant i\leqslant m, 1\leqslant j\leqslant n} w([a_{ij}]). \tag{2.72}$$

If $[\mathbf{A}] \in \mathbb{IR}^{m\times n}$ is bounded and non-empty, then its *midpoint* (or *centre*) is given by

$$\mathrm{mid}([\mathbf{A}]) = (\mathrm{mid}([a_{ij}]))_{1\leqslant i\leqslant m, 1\leqslant j\leqslant n}\,. \tag{2.73}$$

For $[\mathbf{A}]$ and $[\mathbf{B}]$ in $\mathbb{IR}^{m\times n}$ and $\mathbf{C}$ in $\mathbb{R}^{m\times n}$,

$$[\mathbf{A}] \subset [\mathbf{B}] \Leftrightarrow [a_{ij}] \subset [b_{ij}] \text{ for } 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant n, \tag{2.74}$$

$$\mathbf{C} \in [\mathbf{B}] \Leftrightarrow c_{ij} \in [b_{ij}] \text{ for } 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant n. \tag{2.75}$$

The *interval hull* of a set $\mathbb{A}$ of matrices of $\mathbb{R}^{n\times m}$ is the smallest element of $\mathbb{IR}^{n\times m}$ that contains $\mathbb{A}$.

If $[\mathbf{A}]$ and $[\mathbf{B}]$ are intervals, interval vectors or interval matrices of appropriate dimensions and if $\diamond$ is a binary operator, then

$$[\mathbf{A}] \diamond [\mathbf{B}] = [\{\mathbf{A} \diamond \mathbf{B} \mid \mathbf{A} \in [\mathbf{A}] \text{ and } \mathbf{B} \in [\mathbf{B}]\}]\,. \tag{2.76}$$

For instance, if $[\mathbf{A}]$ and $[\mathbf{B}]$ are in $\mathbb{IR}^{n\times n}$, $[\mathbf{x}]$ is in $\mathbb{IR}^n$ and α is in $\mathbb{R}$, then

$$\begin{array}{l} \alpha[\mathbf{A}] = (\alpha[a_{11}]) \times \cdots \times (\alpha[a_{nn}])\,, \\ [\mathbf{A}] + [\mathbf{B}] = ([a_{ij}] + [b_{ij}])_{1\leqslant i\leqslant n, 1\leqslant j\leqslant n}\,, \\ [\mathbf{A}] * [\mathbf{B}] = (\sum_{k=1}^{n} [a_{ik}] * [b_{kj}])_{1\leqslant i\leqslant n, 1\leqslant j\leqslant n}\,, \\ [\mathbf{A}] * [\mathbf{x}] = \left(\sum_{j=1}^{n} [a_{ij}] * [x_j]\right)_{1\leqslant i\leqslant n}. \end{array} \tag{2.77}$$

As with intervals, the product of two interval matrices will be denoted indifferently by $[\mathbf{A}] * [\mathbf{B}]$, or $[\mathbf{A}][\mathbf{B}]$. Some classical properties of matrices in a punctual context are no longer true. For instance, product is no longer associative

$$([\mathbf{A}][\mathbf{B}])[\mathbf{C}] \neq [\mathbf{A}]([\mathbf{B}][\mathbf{C}]), \tag{2.78}$$

or commutative with respect to scalars

$$[\alpha]([\mathbf{A}][\mathbf{x}]) \neq [\mathbf{A}]([\alpha][\mathbf{x}]). \tag{2.79}$$

Part of these specificities can be explained by the wrapping effect, as illustrated by the following example where it is shown that

$$\mathbf{A}[\mathbf{x}] \supset \{\mathbf{A}\mathbf{x} \mid \mathbf{x} \in [\mathbf{x}]\}. \tag{2.80}$$

Example 2.6 *Take*

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad [\mathbf{x}] = \begin{pmatrix} [-1, 0] \\ [1, 2] \end{pmatrix}. \tag{2.81}$$

Then

$$\mathbf{A}\,[\mathbf{x}] = \begin{pmatrix} [0, 2] \\ [1, 2] \end{pmatrix}, \tag{2.82}$$

which implies that $(0,2)^T$ *belongs to* $\mathbf{A}[\mathbf{x}]$*, whereas it does not belong to the actual value set* $\mathbb{B} = \{\mathbf{Ax} \mid \mathbf{x} \in [\mathbf{x}]\}$*, as Figure 2.5 makes clear.* ■

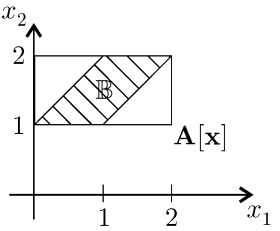


Fig. 2.5. Pessimism introduced by the wrapping effect

Of course, linear algebra also involves more sophisticated operations such as matrix inversion and the computation of eigenvalues and eigenvectors, which raise difficulties that go beyond this introductory chapter. See Neumaier (1990) for more details.

2.4 Inclusion Functions

2.4.1 Definitions

Consider a function $\mathbf{f}$ from $\mathbb{R}^n$ to $\mathbb{R}^m$. The interval function $[\mathbf{f}]$ from $\mathbb{IR}^n$ to $\mathbb{IR}^m$ is an *inclusion function* for $\mathbf{f}$ if

$$\forall [\mathbf{x}] \in \mathbb{IR}^n, \quad \mathbf{f}([\mathbf{x}]) \subset [\mathbf{f}]([\mathbf{x}]). \tag{2.83}$$

The interval function $[\mathbf{f}]([\mathbf{x}]) = \mathbb{R}^m$, for all $[\mathbf{x}] \in \mathbb{IR}^n$, is an example of a (not very useful) inclusion function for all functions $\mathbf{f}$ from $\mathbb{R}^n$ to $\mathbb{R}^m$. One of the purposes of interval analysis is to provide, for a large class of functions $\mathbf{f}$, inclusion functions that can be evaluated reasonably quickly and such that $[\mathbf{f}]([\mathbf{x}])$ is not too large. The function $\mathbf{f}$ may, for instance, be polynomial (Malan et al., 1992; Garloff, 2000), or given by an algorithm (Moore, 1979). It may even be defined as the solution of a set of differential equations (Lohner, 1987; Berz and Makino, 1998; Kühn, 1998).

To illustrate the notion of inclusion function, consider a function $\mathbf{f}$ from $\mathbb{R}^2$ to $\mathbb{R}^2$, with variables x_1 and x_2 that vary within $[x_1]$ and $[x_2]$. The image set $\mathbf{f}([\mathbf{x}])$ may have any shape. It may be non-convex (*i.e.*, there are points of $\mathbf{f}([\mathbf{x}])$ such that the line segment connecting them is not in $\mathbf{f}([\mathbf{x}])$), or even disconnected (*i.e.*, $\mathbf{f}([\mathbf{x}])$ is a union of disjoint sets) if $\mathbf{f}$ is discontinuous. Whatever the shape of $\mathbf{f}([\mathbf{x}])$, an inclusion function $[\mathbf{f}]$ of $\mathbf{f}$ makes it possible to compute a box $[\mathbf{f}]([\mathbf{x}])$ guaranteed to contain it (Figure 2.6).

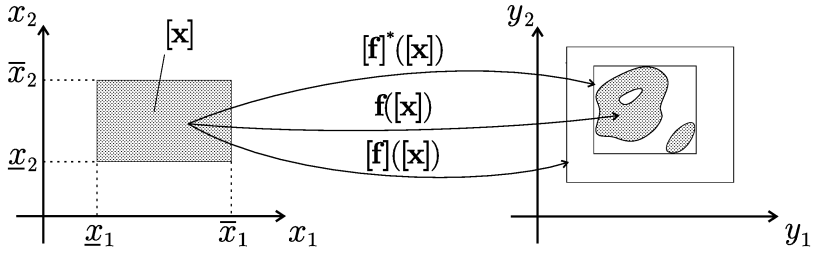


Fig. 2.6. Images of a box by a vector function $\mathbf{f}$ and two of its inclusion functions $[\mathbf{f}]$ and $[\mathbf{f}]^*$; $[\mathbf{f}]^*$ is minimal

Actually, as suggested by Figure 2.6, $[\mathbf{f}]([\mathbf{x}])$ may offer a very pessimistic vision of $\mathbf{f}([\mathbf{x}])$. But remarkable properties of $\mathbf{f}$, such as the positivity of some of its components, may be preserved by $[\mathbf{f}]$. Given that it is far easier to manipulate boxes than generic sets, this is a very interesting standpoint for the observation of a vector function.

An inclusion function $[\mathbf{f}]$ for $\mathbf{f}$ is *thin* if, for any punctual interval vector $[\mathbf{x}] = \mathbf{x}$, $[\mathbf{f}]([\mathbf{x}]) = \mathbf{f}(\mathbf{x})$. It is *convergent* if, for any sequence of boxes $[\mathbf{x}](k)$,

$$\lim_{k\to\infty} w([\mathbf{x}](k)) = 0 \Rightarrow \lim_{k\to\infty} w([\mathbf{f}]([\mathbf{x}](k))) = 0. \tag{2.84}$$

This property is illustrated by Figure 2.7. Note that if $[\mathbf{f}]$ is convergent, it is necessarily thin. The convergence of inclusion functions is required for proving the convergence of interval solvers such as those presented in Chapter 5. The inclusion function $[\mathbf{f}]$ is *minimal* if for any $[\mathbf{x}]$, $[\mathbf{f}]([\mathbf{x}])$ is the smallest box that contains $\mathbf{f}([\mathbf{x}])$. The minimal inclusion function for $\mathbf{f}$ is unique and will be denoted by $[\mathbf{f}]^*$ (see Figure 2.6).

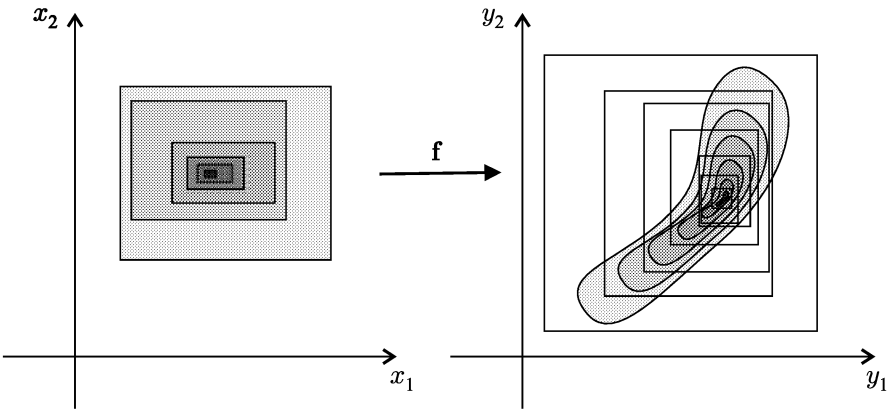


Fig. 2.7. A convergent inclusion function

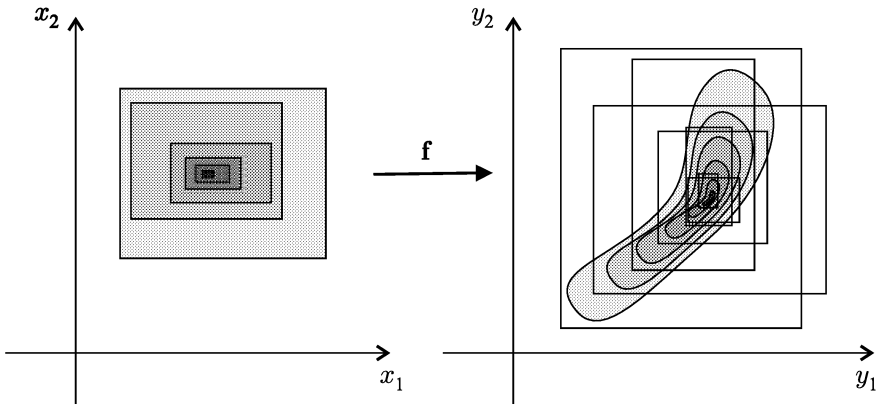


Fig. 2.8. A convergent inclusion function that is not inclusion monotonic

$[\mathbf{f}]$ is *inclusion monotonic* if

$$[\mathbf{x}] \subset [\mathbf{y}] \Rightarrow [\mathbf{f}]([\mathbf{x}]) \subset [\mathbf{f}]([\mathbf{y}]). \tag{2.85}$$

It is trivial to check that a minimal inclusion function is inclusion monotonic but not necessarily convergent (because $\mathbf{f}$ may be discontinuous). A convergent inclusion function may not be inclusion monotonic (see Figure 2.8).

Consider a function $\mathbf{f}$ from $\mathbb{R}^n$ to $\mathbb{R}^m$, and let $[f_j]$, $j = 1, \ldots, m$, be m inclusion functions from $\mathbb{IR}^n$ to $\mathbb{IR}$ associated with the coordinate functions f_j of $\mathbf{f}$. An inclusion function for $\mathbf{f}$ is then given by

$$[\mathbf{f}]([\mathbf{x}]) = [f_1]([\mathbf{x}]) \times \cdots \times [f_m]([\mathbf{x}]). \tag{2.86}$$

$[\mathbf{f}]$ is convergent (thin, minimal, inclusion monotonic, respectively) if all its coordinate functions $[f_i]([\mathbf{x}])$ are convergent (thin, minimal, inclusion monotonic, respectively). The construction of inclusion functions for $\mathbf{f}$ can therefore be cast into that of inclusion functions for each of its coordinate functions. This is why we shall focus attention on getting inclusion functions for real-valued functions.

2.4.2 Natural inclusion functions

The first idea that comes to mind in order to build an inclusion function for $f : \mathbb{R}^n \to \mathbb{R}$ is to perform two optimizations to compute the *infimum* and *supremum* of f when each x_i is constrained to belong to $[x_i]$. At least in principle, one should thus get the smallest interval containing $f([x_1], [x_2], \ldots, [x_n])$, denoted by $[f]^*([x_1], [x_2], \ldots, [x_n])$. However, these optimization problems turn out to be far from trivial in general.

An alternative and much more tractable approach uses the following theorem, which is a direct consequence of Theorem 2.1, page 14.

Theorem 2.2 *Consider a function*

$$\begin{aligned} &f : \mathbb{R}^n \to \mathbb{R}, \\ &\quad (x_1, \ldots, x_n) \mapsto f(x_1, \ldots, x_n), \end{aligned} \tag{2.87}$$

expressed as a finite composition of the operators $+, -, *, /$ *and elementary functions (*$\sin, \cos, \exp,$ sqr$\ldots$*). An inclusion monotonic and thin inclusion function* $[f] : \mathbb{IR}^n \to \mathbb{IR}$ *for* f *is obtained by replacing each real variable* x_i *by an interval variable* $[x_i]$ *and each operator or function by its interval counterpart. This function is called the* natural inclusion function *of* f*. If* f *involves only continuous operators and continuous elementary functions, then* $[f]$ *is convergent. If, moreover, each of the variables* $(x_1, \ldots, x_n)$ *occurs at most once in the formal expression of* f *then* $[f]$ *is minimal.* ■

Remark 2.6 *Contrary to set computation, it is not sufficient that each input variable* x_i *appears at most once for the natural inclusion function to be minimal. Because of the wrapping effect, it is also required that all functions and operators involved in the expression of* f *be continuous. Consider, for instance, the continuous function* $f(x) = (\text{sign}(x))^2$*, where* $\text{sign}(x)$ *is equal to* 1 *if* $x \geqslant 0$ *and to* -1 *otherwise. Its natural inclusion function* $[f]$ *satisfies* $[f]([-1,1]) = [-1,1]^2 = [0,1]$*. Although* x *occurs only once in the formal expression of* $f(x)$*,* $[f]$ *is not minimal as* $f([-1,1]) = 1$*.* ■

Natural inclusion functions are not minimal in general, because of the dependency and wrapping effects. The accuracy of the resulting interval strongly depends on the expression of f, as illustrated by the three following examples. The first one presents a function of one variable to allow a graphical illustration. The next one involves a function of two variables. The last one shows how to deal with transcendental functions.

Example 2.7 *Consider the following four formal expressions of the same function* $f(x)$*:*

$$f_1(x) = x(x+1), \tag{2.88}$$

$$f_2(x) = x * x + x, \tag{2.89}$$

$$f_3(x) = x^2 + x, \tag{2.90}$$

$$f_4(x) = (x + \frac{1}{2})^2 - \frac{1}{4}. \tag{2.91}$$

Evaluate their natural inclusion functions for $[x] = [-1,1]$*:*

$$[f_1]([x]) = [x]([x]+1) = [-2,2], \tag{2.92}$$

$$[f_2]([x]) = [x] * [x] + [x] = [-2,2], \tag{2.93}$$

$$[f_3]([x]) = [x]^2 + [x] = [-1,2], \tag{2.94}$$

$$[f_4]([x]) = \left([x] + \frac{1}{2}\right)^2 - \frac{1}{4} = \left[-\frac{1}{4}, 2\right]. \tag{2.95}$$

The accuracy of the interval result thus depends on the formal expression of f (see Figure 2.9). Since $[x]$ occurs only once in f_4 and f_4 is continuous, $[f_4]$ is minimal. Thus $[f_4]([x]) = f([x]) = \left[-\frac{1}{4}, 2\right]$. ■

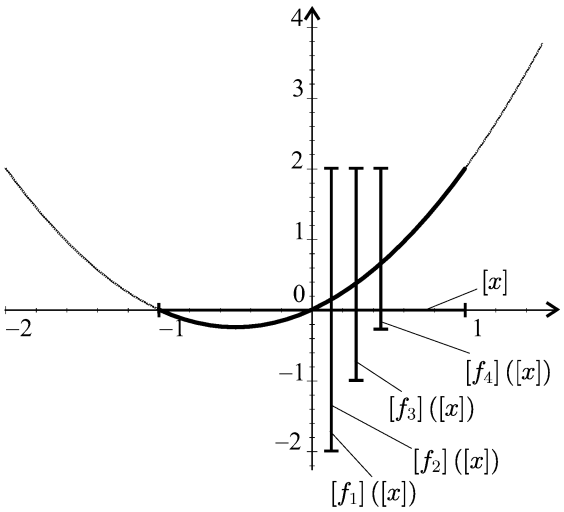


Fig. 2.9. Four natural inclusion functions for the same function

Example 2.8 *Consider the real function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by*

$$f(x_1, x_2) = \frac{x_1 - x_2}{x_1 + x_2}, \qquad \text{with } x_1 \in [-1, 2] \text{ and } x_2 \in [3, 5]. \tag{2.96}$$

The natural inclusion function $[f]_1$ for f is obtained by replacing each real variable by an interval variable, and each real operation by its interval counterpart:

$$[f]_1([x_1], [x_2]) = \frac{[x_1] - [x_2]}{[x_1] + [x_2]}, \tag{2.97}$$

so

$$\begin{aligned}[f]_1([-1, 2], [3, 5]) &= \frac{[-1, 2] - [3, 5]}{[-1, 2] + [3, 5]} = \frac{[-6, -1]}{[2, 7]} \\ &= [-6, -1] * [\frac{1}{7}, \frac{1}{2}] = [-3, -\frac{1}{7}].\end{aligned} \tag{2.98}$$

A second interval extension $[f]_2$ can similarly be obtained after rewriting f in such a way that x_1 and x_2 each appear only once:

$$[f]_2([x_1],[x_2]) = 1 - \frac{2}{1+[x_1]/[x_2]}. \tag{2.99}$$

Then

$$\begin{aligned}[f]_2([-1,2],[3,5]) &= 1 - \frac{2}{1+[-1,2]/[3,5]} = 1 - \frac{2}{1+[-1/3,2/3]} \\ &= 1 - \frac{2}{[2/3,5/3]} = 1 - [6/5,3] \\ &= [-2,-1/5]. \end{aligned} \tag{2.100}$$

$[f]_1$ and $[f]_2$ are both interval extensions of f. $[f]_2$ is more accurate than $[f]_1$, which suffers from the dependency effect. The interval computed by $[f]_2$ is minimal, and thus equal to the image set $f([-1,2],[3,5])$. ■

Example 2.9 *Consider the real function f defined by*

$$f(x_1,x_2) = \ln(e^{x_1} + \sin(x_2)), \tag{2.101}$$

with $x_1 \in [0,1]$ and $x_2 \in [\pi/4, 4\pi/3]$. Its natural inclusion function is

$$[f]([x_1],[x_2]) = \ln(\exp([x_1]) + \sin([x_2])), \tag{2.102}$$

so

$$\begin{aligned}[f]([0,1],[\pi/4,4\pi/3]) &= \ln(\exp([0,1]) + \sin([\pi/4,4\pi/3])) \\ &= \ln([1,e] + [-\sqrt{3}/2,1]) \\ &= \ln([1-\sqrt{3}/2, e+1]) \\ &= [\ln(1-\sqrt{3}/2), \ln(e+1)] \\ &\subset [-2.0101, 1.3133]. \end{aligned} \tag{2.103}$$

Since each variable appears only once, and since all the functions and operators involved in the formal expression of f are continuous, the penultimate interval is the exact image set $f([0,1],[\pi/4,4\pi/3])$. Note that the minimum and maximum values taken by f have thus been computed without performing a single optimization, although f is not monotonic. The last interval is a guaranteed numerical estimate of the previous one, obtained by outward rounding, see Chapter 10. ■

The use of natural inclusion functions is not always to be recommended, however. Their efficiency depends strongly on the number of occurrences of each variable, which is often difficult to reduce. An important field of investigation of interval analysis has thus been to propose other types of inclusion function that would provide less pessimistic results (Ratschek and Rokne, 1984), as shown in Sections 2.4.3 to 2.4.5.

2.4.3 Centred inclusion functions

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a scalar function of a vector $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$. Assume that f is differentiable over the box $[\mathbf{x}]$, and denote $\mathrm{mid}([\mathbf{x}])$ by $\mathbf{m}$. The mean-value theorem then implies that

$$\forall \mathbf{x} \in [\mathbf{x}], \exists \mathbf{z} \in [\mathbf{x}] \mid f(\mathbf{x}) = f(\mathbf{m}) + \mathbf{g}^{\mathrm{T}}(\mathbf{z})(\mathbf{x} - \mathbf{m}), \tag{2.104}$$

where $\mathbf{g}$ is the *gradient* of f, *i.e.*, a column vector with entries $g_i = \partial f / \partial x_i$, $i = 1, \ldots, n$. Thus,

$$\forall \mathbf{x} \in [\mathbf{x}], f(\mathbf{x}) \in f(\mathbf{m}) + [\mathbf{g}^{\mathrm{T}}]([\mathbf{x}])(\mathbf{x} - \mathbf{m}), \tag{2.105}$$

where $[\mathbf{g}^{\mathrm{T}}]$ is an inclusion function for $\mathbf{g}^{\mathrm{T}}$, so

$$f([\mathbf{x}]) \subseteq f(\mathbf{m}) + [\mathbf{g}^{\mathrm{T}}]([\mathbf{x}])([\mathbf{x}] - \mathbf{m}). \tag{2.106}$$

Therefore, the interval function

$$[f_{\mathrm{c}}]([\mathbf{x}]) \triangleq f(\mathbf{m}) + [\mathbf{g}^{\mathrm{T}}]([\mathbf{x}])([\mathbf{x}] - \mathbf{m}) \tag{2.107}$$

is an inclusion function for $\mathbf{f}$, which we shall call the *centred inclusion function*. To illustrate the interest of this function in the one-dimensional case, consider the function $[f_{\mathrm{c}}](x)$ from $\mathbb{R}$ to $\mathbb{IR}$ defined by

$$[f_{\mathrm{c}}](x) \triangleq f(m) + [f']([x])(x - m) \tag{2.108}$$

for any given $[x]$. This function can be viewed as affine in x with an uncertain slope belonging to $[f']([x])$. The graph of $[f_{\mathrm{c}}](x)$ can thus be represented by a cone with centre $(m, f(m))$ as illustrated by Figure 2.10 for decreasing widths of $[x]$. The smaller $w([x])$ is, the better the cone approximates the function. The figure illustrates the fact that

$$\frac{w([f_{\mathrm{c}}]([x]))}{w(f([x]))} \to 1 \tag{2.109}$$

when the width of $[x]$ tends to 0, which is not the case in general for a natural inclusion function.

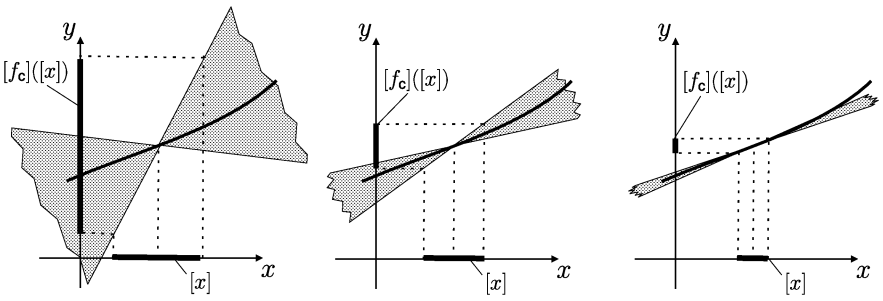


Fig. 2.10. Interpretation of the centred inclusion function

When the width of $[\mathbf{x}]$ is small, the effect of the pessimism possibly resulting from the interval evaluation of $[\mathbf{g}]([\mathbf{x}])$ is reduced by the scalar product with $[\mathbf{x}] - \mathbf{m}$, which is a small interval centred on zero.

2.4.4 Mixed centred inclusion functions

The centred inclusion function for a function f from $\mathbb{R}^n$ to $\mathbb{R}$ can be noticeably improved at the cost of a slightly more complicated formulation (Hansen, 1968). Recall that, for a function φ from $\mathbb{R}$ to $\mathbb{R}$,

$$\varphi(x) \in \varphi(m) + \varphi'([x])([x] - m), \tag{2.110}$$

where $m = \text{ mid}([x])$. The main idea to get a mixed centred inclusion function is to apply (2.110) n times, considering each variable of f in turn. The case $n = 3$ will be treated first, to simplify exposition. Consider $f(x_1, x_2, x_3)$ as a function of x_3 only and take $m_3 = \text{ mid}([x_3])$; (2.110) then implies that

$$f(x_1, x_2, x_3) \in f(x_1, x_2, m_3) + g_3(x_1, x_2, [x_3]) * ([x_3] - m_3). \tag{2.111}$$

Consider now $f(x_1, x_2, m_3)$ as a function of x_2 only and take $m_2 = \text{ mid}([x_2])$; (2.110) then yields

$$f(x_1, x_2, m_3) \in f(x_1, m_2, m_3) + g_2(x_1, [x_2], m_3) * ([x_2] - m_2). \tag{2.112}$$

Finally, consider $f(x_1, m_2, m_3)$ as a function of x_1 and take $m_1 = \text{ mid}([x_1])$; (2.110) then leads to

$$f(x_1, m_2, m_3) \in f(m_1, m_2, m_3) + g_1([x_1], m_2, m_3) * ([x_1] - m_1). \tag{2.113}$$

Combine these three equations to get

$$\begin{aligned} f(x_1, x_2, x_3) \in\ & f(m_1, m_2, m_3) + g_1([x_1], m_2, m_3) * ([x_1] - m_1) \\ & + g_2(x_1, [x_2], m_3) * ([x_2] - m_2) \\ & + g_3(x_1, x_2, [x_3]) * ([x_3] - m_3). \end{aligned} \tag{2.114}$$

Thus

$$\begin{aligned} f([x_1], [x_2], [x_3]) \subset\ & f(m_1, m_2, m_3) + g_1([x_1], m_2, m_3) * ([x_1] - m_1) \\ & + g_2([x_1], [x_2], m_3) * ([x_2] - m_2) \\ & + g_3([x_1], [x_2], [x_3]) * ([x_3] - m_3). \end{aligned} \tag{2.115}$$

This expression can be generalized for a function f of n variables. With $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$ and $\mathbf{m} = \text{mid}([\mathbf{x}])$, one gets

$$f([\mathbf{x}]) \subset f(\mathbf{m}) + \sum_{i=1}^{n} [g_i]([x_1], \ldots, [x_i], m_{i+1}, \ldots, m_n) * ([x_i] - m_i), \tag{2.116}$$

and the right-hand side of (2.116) defines the *mixed centred inclusion function*. The main difference with (2.107) lies in the arguments of the gradient. In

(2.116), interval and punctual arguments are mixed, which allows pessimism to be decreased, as

$$[\mathbf{g}](\mathrm{mid}([\mathbf{x}]), [\mathbf{x}]) \subset [\mathbf{g}]([\mathbf{x}]). \tag{2.117}$$

2.4.5 Taylor inclusion functions

Iterating the reasoning that led to the centred inclusion function, one may think of using Taylor series expansion to approximate a function f from $\mathbb{R}^n$ to $\mathbb{R}$ at a higher order. This leads to the *Taylor inclusion function*. Consider a second-order expansion as an illustration:

$$[f]_{\mathrm{T}}([\mathbf{x}]) = f(\mathbf{m}) + \mathbf{g}^{\mathrm{T}}(\mathbf{m})([\mathbf{x}] - \mathbf{m}) + \frac{1}{2}([\mathbf{x}] - \mathbf{m})^{\mathrm{T}}[\mathbf{H}]([\mathbf{x}])([\mathbf{x}] - \mathbf{m}), \tag{2.118}$$

where $\mathbf{m} = \mathrm{mid}([\mathbf{x}])$, $\mathbf{g}$ is again the gradient of f and $[\mathbf{H}]([\mathbf{x}])$ is the interval *Hessian matrix*. The entry $[\mathbf{H}]_{ij}$ of $[\mathbf{H}]$ is an inclusion function of

$$h_{ij} = \begin{cases} \partial^2 f/\partial x_i^2 & \text{if } j = i \quad (i = 1, \ldots, n), \\ 2\partial^2 f/\partial x_i x_j & \text{if } j < i \quad (i = 1, \ldots, n), \\ 0 & \text{otherwise.} \end{cases} \tag{2.119}$$

A symmetric form of the Hessian matrix ($h_{ij} = \partial^2 f/\partial x_i x_j$ for all i and j) could also be used, but the resulting increase in the number of interval components in $[\mathbf{H}]([\mathbf{x}])$ would then lead to an increase in the pessimism of $[f]_{\mathrm{T}}$. Pessimism can be reduced by replacing $[\mathbf{H}]([\mathbf{x}])$ by a mixed expression $[\mathbf{H}](\mathrm{mid}([\mathbf{x}]), [\mathbf{x}])$, as was done for the gradient in the mixed centred form of Section 2.4.4.

When f has only one variable, the nth-order Taylor inclusion function is given by

$$[f]_{\mathrm{T}}([x]) = f(m) + f'(m)([x] - m) + \cdots + f^{n-1}(m)\frac{([x] - m)^{n-1}}{(n-1)!} + [f^n]([x])\frac{([x] - m)^n}{n!}. \tag{2.120}$$

Evaluation of the Taylor inclusion function of order n thus requires computation of the derivatives of f up to nth order, which may entail cumbersome manipulations.

2.4.6 Comparison

Under mild technical conditions (Moore, 1979), the natural, centred and Taylor inclusion functions are convergent. Roughly speaking, the *convergence rate* of a convergent inclusion function is the largest α such that

$$\exists \beta \mid w([f]([\mathbf{x}])) - w(f([\mathbf{x}])) \leqslant \beta w([\mathbf{x}])^{\alpha} \tag{2.121}$$

when $w([\mathbf{x}])$ tends to 0. When an inclusion function is minimal, its convergence rate is infinite. The convergence rate of a natural inclusion function is at least linear ($\alpha \geqslant 1$), whereas the convergence rate of a centred form is at least quadratic ($\alpha \geqslant 2$). The convergence rate of a Taylor inclusion function is also at least quadratic for any order $n \geqslant 2$. Quadratic convergence looks of course more interesting than linear convergence, but it should be remembered that it only means that more accurate results will be obtained in the case of infinitesimal boxes. Nothing similar can be said on the behaviour of these inclusion functions for boxes of a more realistic size. When the box involved is large, the natural inclusion function is generally more satisfactory than the centred inclusion function, whereas the latter performs better when the box is small, with the mixed version superior to the standard version.

No approach to building an inclusion function can claim to be uniformly the best, and a compromise between complexity and efficiency must often be struck. One may also use several inclusion functions and take the intersection of their image sets to get a better approximation of the image set of the original function.

Example 2.10 *Consider the function f defined by*

$$f(x) = x^2 + \sin(x), \tag{2.122}$$

and the intervals

$$[x] = [\frac{2\pi}{3}, \frac{4\pi}{3}] \text{ and } [y] = [\frac{99\pi}{100}, \frac{101\pi}{100}]. \tag{2.123}$$

We shall compare the approximations of $f([x])$ and $f([y])$ obtained when using the natural, centred, Taylor of order two and minimal inclusion functions, which will be respectively denoted by $[f]_{\rm n}$, $[f]_{\rm c}$, $[f]_{\rm T}$, and $[f]^*$. See also Exercise 11.10, page 318. The first three of these functions are given by

$$[f]_{\rm n}([x]) = [x]^2 + \sin([x]), \tag{2.124}$$

$$[f]_{\rm c}([x]) = f(\pi) + ([x] - \pi)[f']([x]), \tag{2.125}$$

$$[f]_{\rm T}([x]) = f(\pi) + ([x] - \pi)f'(\pi) + \frac{([x] - \pi)^2}{2}[f'']([x]), \tag{2.126}$$

with

$$f'(x) = 2x + \cos(x) \quad \text{and} \quad f''(x) = 2 - \sin(x). \tag{2.127}$$

The minimal inclusion function is trivial to evaluate after noticing that f is increasing over $[x]$ (and thus over $[y] \subset [x]$). So

$$[f]^*([x]) = [\ \underline{x}^2 + \sin(\underline{x}),\ \overline{x}^2 + \sin(\overline{x})]. \tag{2.128}$$

The results obtained by evaluating each of these inclusion functions over $[x]$ and $[y]$ are indicated in Table 2.2, where $\Delta([f]([x]))$ stands for the value of $w([f]([x])) - w(f([x]))$.

Table 2.2. Comparing inclusion functions

	$[x] = \left[\frac{2\pi}{3}, \frac{4\pi}{3}\right]$		$[y] = \left[\frac{99\pi}{100}, \frac{101\pi}{100}\right]$	
$[f]$	$[f]([x])$	$\Delta([f]([x]))$	$[f]([y])$	$\Delta([f]([y]))$
$[f]_{\mathrm{n}}$	$[3.52046, 18.41199]$	3.46410	$[9.64178, 10.09940]$	0.12564
$[f]_{\mathrm{c}}$	$[1.62022, 18.11899]$	5.07134	$[9.70163, 10.03758]$	0.00397
$[f]_{\mathrm{T}}$	$[4.33706, 16.97362]$	1.20913	$[9.70362, 10.03659]$	0.00099
$[f]^*$	$[5.25251, 16.67994]$	0	$[9.70461, 10.03658]$	0

The numerical values are given with an accuracy of 10^{-5}. *It turns out that the natural inclusion function remains competitive for the larger interval* $[x]$, *which is an additional incentive to use it, besides its simplicity. The centred and Taylor inclusion functions are more efficient than the natural inclusion function for the smaller interval* $[y]$. *The Taylor inclusion function brings a noticeable improvement compared to the natural and centred inclusion functions, even for the larger interval. Finally, this example reminds us that it may be useful to check whether the function considered is monotonic, in which case obtaining a minimal form is trivial. Unfortunately, the expressions encountered in engineering applications are seldom as cooperative as the one considered here, which limits the practical interest of this remark.* ■

Example 2.11 *Consider now a vector function* $\mathbf{f}$ *from* $\mathbb{R}^2$ *to* $\mathbb{R}^2$, *defined by*

$$\begin{aligned} f_1(x_1, x_2) &= x_1^2 + x_1 \exp(x_2) - x_2^2, \\ f_2(x_1, x_2) &= x_1^2 - x_1 \exp(x_2) + x_2^2, \end{aligned} \tag{2.129}$$

where x_1 *and* x_2 *belong to* $[x_1]$ *and* $[x_2]$ *respectively. The natural inclusion function* $[\mathbf{f}]_{\mathrm{n}}$ *for* $\mathbf{f}$ *is given by*

$$[\mathbf{f}]_{\mathrm{n},1}([\mathbf{x}]) = [x_1]^2 + [x_1] \exp([x_2]) - [x_2]^2, \tag{2.130}$$

$$[\mathbf{f}]_{\mathrm{n},2}([\mathbf{x}]) = [x_1]^2 - [x_1] \exp([x_2]) + [x_2]^2. \tag{2.131}$$

The centred inclusion function $[\mathbf{f}]_{\mathrm{c}}$ *is given by*

$$[\mathbf{f}]_{\mathrm{c}}([\mathbf{x}]) = \mathbf{f}(\mathrm{mid}([\mathbf{x}])) + [\mathbf{J_f}]([\mathbf{x}]) * ([\mathbf{x}] - \mathrm{mid}([\mathbf{x}])). \tag{2.132}$$

All the arguments of the interval Jacobian matrix $[\mathbf{J_f}]$ *are intervals and the* i*th row of* $[\mathbf{J_f}]([\mathbf{x}])$ *is given by* $[\mathbf{g}_i^{\mathrm{T}}]([\mathbf{x}])$, *with* $\mathbf{g}_i$ *the gradient of the* i*th component of* $\mathbf{f}$ *(see (2.107) page 33):*

$$[\mathbf{J_f}]([\mathbf{x}]) = \begin{pmatrix} 2[x_1] + \exp([x_2]) & -2[x_2] + [x_1] \exp([x_2]) \\ 2[x_1] - \exp([x_2]) & 2[x_2] - [x_1] \exp([x_2]) \end{pmatrix}. \tag{2.133}$$

The mixed centred inclusion function $[\mathbf{f}]_{\mathrm{m}}$ *is given by*

$$[\mathbf{f}]_{\mathrm{m}}([\mathbf{x}]) = \mathbf{f}(\mathrm{mid}([\mathbf{x}])) + [\mathbf{J_f}](\mathrm{mid}([\mathbf{x}]), [\mathbf{x}]) * ([\mathbf{x}] - \mathrm{mid}([\mathbf{x}])), \tag{2.134}$$

where $[\mathbf{J_f}]$ *now depends on a mixture of punctual and interval arguments*

$$[\mathbf{J_f}](\text{mid}([\mathbf{x}]), [\mathbf{x}]) = \begin{pmatrix} 2[x_1] + \exp(\text{mid}([x_2])) & -2[x_2] + [x_1]\exp([x_2]) \\ 2[x_1] - \exp(\text{mid}([x_2])) & 2[x_2] - [x_1]\exp([x_2]) \end{pmatrix}. \tag{2.135}$$

As expected, (2.135) contains less interval arguments than (2.133). Table 2.3 compares the efficiency of the natural, centred and mixed centred inclusion functions on two boxes $[\mathbf{x}]$ *and* $[\mathbf{y}]$*. The definition of* Δ *is as in Example 2.10. The conclusions are similar to those of the scalar case: for the larger box* $[\mathbf{x}]$*, the natural inclusion function is more satisfactory, whereas the centred inclusion function performs better for the smaller box* $[\mathbf{y}]$*, with the mixed version superior to the non-mixed one.* ■

Table 2.3. Comparison of vector inclusion functions

	$[x_1] = [0.5, 1.5]\,; [x_2] = [1.5, 2.5]$		$[y_1] = [0.9, 1.1]\,; [y_2] = [1.9, 2.1]$	
$[\mathbf{f}]$	$[\mathbf{f}]([\mathbf{x}])$	$\Delta[[\mathbf{f}]([\mathbf{x}])]$	$[\mathbf{f}]([\mathbf{y}])$	$\Delta[[\mathbf{f}]([\mathbf{y}])]$
$[f_1]_{\text{n}}$	$[-3.75916, 18.2737]$	7.67904	$[2.41730, 6.58279]$	1.60000
$[f_2]_{\text{n}}$	$[-15.7737, 6.25916]$	11.67904	$[-4.56279, -0.39730]$	2.40001
$[f_1]_{\text{c}}$	$[-10.8391, 19.6172]$	16.10248	$[2.83416, 5.94395]$	0.54430
$[f_2]_{\text{c}}$	$[-15.6172, 10.8391]$	16.10248	$[-3.54395, -1.23416]$	0.54431
$[f_1]_{\text{m}}$	$[-8.44234, 17.2205]$	11.30902	$[2.91187, 5.86624]$	0.38888
$[f_2]_{\text{m}}$	$[-13.2205, 8.44234]$	11.30902	$[-3.46624, -1.31187]$	0.38889
$[f_1]^*$	$[-0.08008, 14.27374]$	0	$[3.21730, 5.78279]$	0
$[f_2]^*$	$[-9.77374, 0.58008]$	0	$[-3.36279, -1.59731]$	0

2.5 Inclusion Tests

Inclusions tests can be used to prove that all the points in a given box satisfy a given property, or to prove that none of them does. These tests involve interval Booleans, which will be presented first.

2.5.1 Interval Booleans

Set computation as defined in Section 2.2 can be used for the Boolean set

$$\mathbb{B} \triangleq \{\mathit{false}, \mathit{true}\}, \tag{2.136}$$

but there is no need to use wrappers for outer approximating the Boolean sets to be handled, since $\mathbb{B}$ is finite. There will thus be no wrapping effect, but

the dependency effect will still be present. A *Boolean number* is an element of $\mathbb{B}$. By extension[1], an *interval Boolean* is a subset of $\mathbb{B}$. Thus, the set of all interval Booleans is

$$\mathbb{IB} = \{\varnothing, 0, 1, [0,1]\}, \tag{2.137}$$

where $\varnothing$ stands for *impossible,* 0 for *false*, 1 for *true*, and $[0,1]$ for *indeterminate.* Operations on interval Booleans are easily defined in the framework of set computation:

$$\begin{aligned}
[a] \vee [b] &= \{a \vee b \mid a \in [a], b \in [b]\}, \\
[a] \wedge [b] &= \{a \wedge b \mid a \in [a], b \in [b]\}, \\
\neg [a] &= \{\neg a \mid a \in [a]\}, \\
[a] \cap [b] &= \{\max(\underline{a}, \underline{b}), \min(\overline{a}, \overline{b})\}, \\
[a] \cup [b] &= \{\min(\underline{a}, \underline{b}), \max(\overline{a}, \overline{b})\},
\end{aligned} \tag{2.138}$$

where $\wedge$ and $\vee$ respectively stand for the `AND` and `OR` operators and where $\neg$ is the complementation operator, such that$\neg 0 = 1$ and $\neg 1 = 0$. For instance,

$$([0,1] \vee 1) \wedge ([0,1] \wedge 1) = 1 \wedge [0,1] = [0,1]. \tag{2.139}$$

If $[a] \in \mathbb{IB}$, then

$$0 \wedge [a] = 0;\ 1 \wedge [a] = [a];\ [a] \wedge [a] = [a]; \tag{2.140}$$

$$0 \vee [a] = [a];\ 1 \vee [a] = 1;\ [a] \vee [a] = [a]. \tag{2.141}$$

The dependency effect is still present when the complementation operator is used. For instance,

$$[a] \subset ([a] \wedge [b]) \vee ([a] \wedge \neg [b]), \tag{2.142}$$

and the values of the two sides of (2.142) differ for $a = 0$ (or 1) and $[b] = [0,1]$, whereas $a = (a \wedge b) \vee (a \wedge \neg b)$.

Any function β from $\mathbb{B}^n$ to $\mathbb{B}$ will be called a *Boolean function*. The notion of inclusion function developed for real functions readily extends to Boolean functions. $[\beta] : \mathbb{IB}^n \to \mathbb{IB}$ is an *inclusion function* for β if

$$\forall ([b_1], \ldots, [b_n]) \in \mathbb{IB}^n, \beta([b_1], \ldots, [b_n]) \subset [\beta]([b_1], \ldots, [b_n]). \tag{2.143}$$

The *natural inclusion function* $[\beta]$ of β is obtained by replacing all arguments and operators of β by their interval counterparts. $[\beta]([b_1], \ldots, [b_n])$ is *minimal* if

$$\forall ([b_1], \ldots, [b_n]) \in \mathbb{IB}^n, \beta([b_1], \ldots, [b_n]) = [\beta]([b_1], \ldots, [b_n]). \tag{2.144}$$

[1] For any set equipped with a partial ordering $(\mathbb{S}, \leqslant)$, one can always define the set $\mathbb{IS}$, of the pairs $[a,b]$ such that $a \in \mathbb{S}$, $b \in \mathbb{S}$ and $a \leqslant b$. The elements of $\mathbb{IS}$ will be called intervals. $\mathbb{S}$ may for instance stand for $\mathbb{R}^n$, for the set of all Boolean numbers $\mathbb{B}$ or for the set of all compact sets. In the last case, the partial ordering is $\subset$.

As with real functions, whenever the expression $\beta(b_1, \ldots, b_n)$ is non-decreasing with respect to all its variables, the minimal inclusion function is given by

$$[\beta]^*([b_1], \ldots, [b_n]) = \left[\beta(\underline{b}_1, \ldots, \underline{b}_n), \beta(\overline{b}_1, \ldots, \overline{b}_n)\right]. \tag{2.145}$$

Note that this is a frequently encountered situation. It is the case, for instance, when the complementation operator is not used, *i.e.*, when β is a polynomial. There are nevertheless many Boolean expressions that are not monotonic, such as the *exclusive or*

$$\beta(b_1, b_2) = (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2). \tag{2.146}$$

Even when a Boolean expression is not monotonic, it is always possible, at least in principle, to evaluate a minimal inclusion function for it, because each of the interval Booleans may take four values at most. Consider, for example, $\beta([b_1], [b_2])$, with $\beta(b_1, b_2) = (b_1 \wedge b_2) \vee (b_1 \wedge \neg b_2)$, for $[b_1] = 1$ and $[b_2] = [0, 1]$. The natural interval extension of β yields $[\beta](1, [0, 1]) = (1 \wedge [0, 1]) \vee (1 \wedge \neg [0, 1]) = [0, 1]$. A minimal evaluation is obtained by writing $\beta(1, [0, 1]) = \beta(1, 0) \cup \beta(1, 1) = 1$. Of course, this approach leads to a combinatorial explosion when the number of variables increases, which can sometimes be avoided by manipulating the Boolean expressions, for instance with the help of Karnaugh tables or by taking advantage of well-known simplification rules to reduce the number of occurrences of the Boolean variables.

2.5.2 Tests

A *test* is a function t from $\mathbb{R}^n$ to $\mathbb{B}$. An *inclusion test* for t is a function $[t]$ from $\mathbb{IR}^n$ to $\mathbb{IB}$ such that for any $[\mathbf{x}] \in \mathbb{IR}^n$,

$$\begin{array}{l} ([t]([\mathbf{x}]) = 1) \Rightarrow (\forall \mathbf{x} \in [\mathbf{x}],\ t(\mathbf{x}) = 1), \\ ([t]([\mathbf{x}]) = 0) \Rightarrow (\forall \mathbf{x} \in [\mathbf{x}],\ t(\mathbf{x}) = 0). \end{array} \tag{2.147}$$

An inclusion test $[t]$ is *thin* if $[t](\mathbf{x}) = t(\mathbf{x})$ for any $\mathbf{x} \in \mathbb{R}^n$. It is *minimal* if

$$\forall [\mathbf{x}] \in \mathbb{IR}^n, [t]([\mathbf{x}]) = \{t(\mathbf{x}) \mid \mathbf{x} \in [\mathbf{x}]\}. \tag{2.148}$$

A minimal test is necessarily thin.

Example 2.12 *Consider the test*

$$\begin{array}{rccc} t: & \mathbb{R}^2 & \to & \{0, 1\} \\ & (x_1, x_2)^T & \mapsto & (x_1 + x_2 \leqslant 5), \end{array} \tag{2.149}$$

which means that

$$t(\mathbf{x}) = \begin{cases} 1 \text{ if } x_1 + x_2 \leqslant 5, \\ 0 \text{ if } x_1 + x_2 > 5. \end{cases} \tag{2.150}$$

The minimal inclusion test $[t]$ *associated with* t *is given by*

$$[t]([\mathbf{x}]) = \begin{cases} 1 & \text{if } \overline{x}_1 + \overline{x}_2 \leqslant 5, \\ 0 & \text{if } \underline{x}_1 + \underline{x}_2 > 5, \\ [0,1] & \text{otherwise,} \end{cases} \tag{2.151}$$

which can be written more concisely as

$$[t]([\mathbf{x}]) \Leftrightarrow ([x_1] + [x_2] \leqslant 5)\,. \tag{2.152}$$

It is minimal and thin. ■

Any Boolean operator on real numbers, such as $(\leqslant, \geqslant, <, >$, integer, odd, even, prime...$)$ can be similarly extended to intervals. For instance,

$$\begin{aligned} &([a,b] \leqslant [c,d]) = 1 \qquad \text{if } b \leqslant c, \\ &([a,b] \leqslant [c,d]) = 0 \qquad \text{if } a > d, \\ &([a,b] \leqslant [c,d]) = [0,1] \text{ if neither } b \leqslant c \text{ nor } a > d. \end{aligned} \tag{2.153}$$

The Boolean comparison operator $=$ cannot be extended in this way, because it has already been given a bivalued meaning by set theory:

$$\begin{aligned} ([a,b] = [c,d]) &= 1 \text{ if } a = c \text{ and } b = d, \\ &= 0 \text{ otherwise.} \end{aligned} \tag{2.154}$$

With the help of interval analysis and the notion of inclusion function, it is easy to build an inclusion test for any test that can be put in the form

$$t(\mathbf{x}) = \beta(t_1(\mathbf{x}), \ldots, t_n(\mathbf{x})), \tag{2.155}$$

with

$$t_i(\mathbf{x}) \Leftrightarrow (f_i(\mathbf{x}) \geqslant 0)\,, \quad i = 1, \ldots, n, \tag{2.156}$$

and $\beta : \mathbb{B}^n \to \mathbb{B}$ a Boolean expression. This inclusion test is given by

$$[t]([\mathbf{x}]) = [\beta]([t_1]([\mathbf{x}]), \ldots, [t_n]([\mathbf{x}])), \tag{2.157}$$

with

$$[t_i]([\mathbf{x}]) \Leftrightarrow ([f_i]([\mathbf{x}]) \geqslant 0)\,, \quad i = 1, \ldots, n, \tag{2.158}$$

and $[\beta]$ some inclusion function for β. Note that even if the tests $[t_i]([\mathbf{x}])$ are all minimal and if $[\beta]$ is minimal too, the dependency effect is still lurking, so pessimism can still be introduced. For instance, the test $t(x) = (x \leqslant 7) \vee (x \geqslant 6)$ admits the inclusion test $[t]([x]) = ([x] \leqslant 7) \vee ([x] \geqslant 6)$. Despite the fact that $[t]([x])$ consists of two minimal inclusion tests and a polynomial and thus increasing expression $\beta(b_1, b_2) = b_1 \vee b_2$, this inclusion test is pessimistic. For instance for $[x] = [5,8]$,

$$[t]([x]) = ([5,8] \leqslant 7) \vee ([5,8] \geqslant 6) = [0,1] \ \vee \ [0,1] = [0,1], \tag{2.159}$$

whereas $t([x]) = \{t(x) \mid x \in [5,8]\} = 1$.

2.5.3 Inclusion tests for sets

Let $\mathbb{A}$ be a set of $\mathbb{R}^n$; an *inclusion test* $[t_{\mathbb{A}}]$ *for* $\mathbb{A}$ is an inclusion test for the test $t_{\mathbb{A}}(\mathbf{x}) \Leftrightarrow (\mathbf{x} \in \mathbb{A})$, *i.e.*, $[t_{\mathbb{A}}]$ satisfies

$$\begin{array}{l} [t_{\mathbb{A}}]([\mathbf{x}]) = 1 \Rightarrow (\forall \mathbf{x} \in [\mathbf{x}]\,,\ t_{\mathbb{A}}(\mathbf{x}) = 1) \Leftrightarrow ([\mathbf{x}] \subset \mathbb{A})\,, \\ [t_{\mathbb{A}}]([\mathbf{x}]) = 0 \Rightarrow (\forall \mathbf{x} \in [\mathbf{x}]\,,\ t_{\mathbb{A}}(\mathbf{x}) = 0) \Leftrightarrow ([\mathbf{x}] \cap \mathbb{A} = \varnothing)\,. \end{array} \tag{2.160}$$

When $[t_{\mathbb{A}}]([\mathbf{x}]) = [0,1]$, nothing can be concluded as to the inclusion of $[\mathbf{x}]$ in $\mathbb{A}$.

The notion of inclusion test for sets will simplify the presentation of algorithms in future chapters.

Previous definitions can be adapted to inclusion tests for sets:

$[t_{\mathbb{A}}]([\mathbf{x}])$ is *inclusion monotonic* iff	$([\mathbf{x}] \subset [\mathbf{y}]) \Rightarrow ([t_{\mathbb{A}}]([\mathbf{x}]) \subset [t_{\mathbb{A}}]([\mathbf{y}]))$
$[t_{\mathbb{A}}]$ is *minimal* iff	$\forall [\mathbf{x}] \in \mathbb{IR}^n,\ [t_{\mathbb{A}}]([\mathbf{x}]) = t_{\mathbb{A}}([\mathbf{x}])$
$[t_{\mathbb{A}}]$ is *thin* iff	$\forall \mathbf{x} \in \mathbb{R}^n,\ [t_{\mathbb{A}}](\mathbf{x}) \neq [0,1]$

The inclusion test $[t_{\mathbb{A}}]$ will be said to be *more accurate* than the inclusion test $[t'_{\mathbb{A}}]$ iff

$$\forall [\mathbf{x}] \in \mathbb{IR}^n, [t_{\mathbb{A}}]([\mathbf{x}]) \subset [t'_{\mathbb{A}}]([\mathbf{x}])\,. \tag{2.161}$$

The following properties can be used to build inclusion tests for sets defined from elementary set operations such as union, intersection or complementation. If $[t_{\mathbb{A}}]([\mathbf{x}])$ and $[t_{\mathbb{B}}]([\mathbf{x}])$ are thin inclusion tests for the sets $\mathbb{A}$ and $\mathbb{B}$, define

$$\begin{array}{ll} [t_{\mathbb{A}\cap\mathbb{B}}]([\mathbf{x}]) & \triangleq ([t_{\mathbb{A}}] \cap [t_{\mathbb{B}}])([\mathbf{x}]) = [t_{\mathbb{A}}]([\mathbf{x}]) \cap [t_{\mathbb{B}}]([\mathbf{x}]), \\ [t_{\mathbb{A}\cup\mathbb{B}}]([\mathbf{x}]) & \triangleq ([t_{\mathbb{A}}] \cup [t_{\mathbb{B}}])([\mathbf{x}]) = [t_{\mathbb{A}}]([\mathbf{x}]) \cup [t_{\mathbb{B}}]([\mathbf{x}]), \\ [t_{\neg\mathbb{A}}]([\mathbf{x}]) & \triangleq \neg\,[t_{\mathbb{A}}]([\mathbf{x}]) = 1 - [t_{\mathbb{A}}]([\mathbf{x}]). \end{array} \tag{2.162}$$

$[t_{\mathbb{A}\cap\mathbb{B}}]$, $[t_{\mathbb{A}\cup\mathbb{B}}]$ and $[t_{\neg\mathbb{A}}]$ are then thin inclusion tests for the sets $\mathbb{A} \cap \mathbb{B}$, $\mathbb{A} \cup \mathbb{B}$ and $\neg\mathbb{A} \triangleq \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{x} \notin \mathbb{A}\}$, respectively.

2.6 Conclusions

Computation on sets should in general be viewed as an idealization, because generic sets cannot be represented and handled exactly by a computer. This is why the notion of set of wrappers was introduced. Wrappers are simple sets easily manipulated by computers, to be used to approximate more complicated sets.

Interval analysis uses intervals and boxes as wrappers, which makes it possible to compute outer approximations of ranges of functions. When evaluated for interval arguments, inclusion functions yield outer approximations

of the actual image sets of interest. Interval computation is thus usually pessimistic. This pessimism is due to the *dependency and wrapping effects*. The dependency effect, already present when computing on sets, takes place when variables occur several times in the formal expression of the function to be evaluated. The wrapping effect is due to the fact that generic sets are contained in intervals or boxes.

Pessimism may be reduced by transforming the formal expression of functions in order to decrease the number of occurrences of the variables or by using more sophisticated inclusion functions than the natural inclusion function obtained by replacing each operator and elementary function by its interval counterpart.

Intervals and boxes alone cannot describe all sets of interest with sufficient accuracy. The next chapter will show how this can be done by using unions of intervals or of boxes.

3. Subpavings

3.1 Introduction

As we have seen in the previous chapter, intervals and boxes form an attractive class of wrappers, easily manipulated. These wrappers, however, are not by themselves general enough satisfactorily to describe all types of sets of interest to us, which are of course not restricted to intervals and boxes and include, for instance, unions of disconnected subsets.

The policy to be followed is based upon covering the set of interest $\mathbb{X}$ with subsets of $\mathbb{R}^n$ that are easy to represent and manipulate. The class of these subsets could be that of ellipsoids, boxes, polytopes, zonotopes, etc. (see Schweppe, 1968; Fogel and Huang, 1982; Milanese and Belforte, 1982; Walter and Piet-Lahanier, 1989; Milanese et al., 1996; Kühn, 1998, and the references therein). Important properties of $\mathbb{X}$ can be proved by using such a covering. If, for instance, the covering is empty, then $\mathbb{X}$ is empty too. In this book, $\mathbb{X}$ will be covered with sets of non-overlapping boxes of $\mathbb{R}^n$, or *subpavings*. We shall also *bracket* $\mathbb{X}$ between inner and outer approximations (Jaulin and Walter, 1993a, 1993c; Jaulin, 1994). Two subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ will then be computed, such that

$$\underline{\mathbb{X}} \subset \mathbb{X} \subset \overline{\mathbb{X}}. \tag{3.1}$$

The knowledge of the pair $[\underline{\mathbb{X}}, \overline{\mathbb{X}}]$ provides valuable information about $\mathbb{X}$. For instance, $\text{vol}(\underline{\mathbb{X}}) \leqslant \text{vol}(\mathbb{X}) \leqslant \text{vol}(\overline{\mathbb{X}})$, if $\overline{\mathbb{X}}$ is empty then $\mathbb{X}$ is empty too, and if $\underline{\mathbb{X}}$ is non-empty then $\mathbb{X}$ is non-empty too. It may even be possible to prove that $\mathbb{X}$ is connected or disconnected. For instance, if $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ both consist of two disconnected subsets: $\underline{\mathbb{X}} = \underline{\mathbb{X}}_1 \cup \underline{\mathbb{X}}_2$ with $\underline{\mathbb{X}}_1 \cap \underline{\mathbb{X}}_2 = \varnothing$ and $\overline{\mathbb{X}} = \overline{\mathbb{X}}_1 \cup \overline{\mathbb{X}}_2$ with $\overline{\mathbb{X}}_1 \cap \overline{\mathbb{X}}_2 = \varnothing$, and if moreover $\underline{\mathbb{X}}_1 \subset \overline{\mathbb{X}}_1$ and $\underline{\mathbb{X}}_2 \subset \overline{\mathbb{X}}_2$ then $\mathbb{X}$ is not connected. This type of information could not be obtained from a representation of $\mathbb{X}$ by a cloud of points, obtained, for example, by a Monte-Carlo method or by systematic gridding.

Section 3.2 recalls the notion of distance between sets to be used for the evaluation of the quality of the approximation of a set by another one. Section 3.3 introduces the approximation of sets by subpavings and explains how these objects can be implemented. Finally, Section 3.4 presents algorithms evaluating the direct and inverse images of a compact set by a given function.

L. Jaulin et al., *Applied Interval Analysis*

3.2 Set Topology

3.2.1 Distances between compact sets

Let $\mathcal{C}(\mathbb{R}^n)$ be the set of all compact sets of $\mathbb{R}^n$. To quantify the quality of a given representation of a compact set, a measure of the distance between $\mathbb{A}$ and $\mathbb{B}$ of $\mathcal{C}(\mathbb{R}^n)$ will be needed. Recall that compact sets of $\mathbb{R}^n$ are closed and bounded subsets of $\mathbb{R}^n$. Equip $\mathbb{R}^n$ with the distance

$$L_\infty(\mathbf{x}, \mathbf{y}) \triangleq \max_{i \in \{1,\ldots,n\}} |y_i - x_i|. \tag{3.2}$$

The unit ball $\mathbb{U} \triangleq [-1,1]^{\times n}$ of $(\mathbb{R}^n, L_\infty)$ is then a hypercube with width two. The *proximity* of $\mathbb{A}$ to $\mathbb{B}$ is

$$h^0_\infty(\mathbb{A}, \mathbb{B}) \triangleq \inf\left\{r \in \mathbb{R}^+ \mid \mathbb{A} \subset \mathbb{B} + r\mathbb{U}\right\}. \tag{3.3}$$

Figure 3.1 illustrates this notion; to get $h^0_\infty(\mathbb{A}, \mathbb{B})$ one inflates $\mathbb{B}$ until it contains $\mathbb{A}$ and to get $h^0_\infty(\mathbb{B}, \mathbb{A})$ one inflates $\mathbb{A}$ until it contains $\mathbb{B}$. Note that h^0_∞ may also be applied to non-compact sets, in which case their proximity may be infinite.

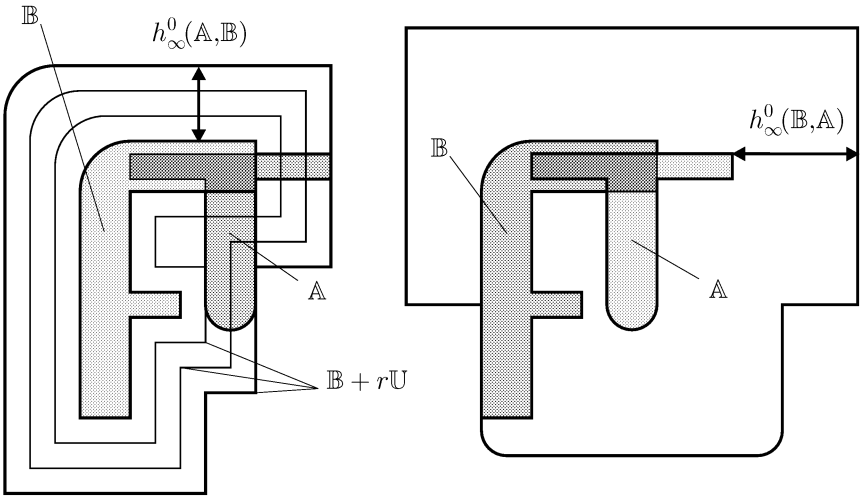


Fig. 3.1. The Hausdorff distance $h_\infty(\mathbb{A}, \mathbb{B})$ is equal to $\max\left\{h^0_\infty(\mathbb{A}, \mathbb{B}), h^0_\infty(\mathbb{B}, \mathbb{A})\right\}$

The *Hausdorff distance* (Berger, 1979) between $\mathbb{A}$ and $\mathbb{B}$ is given by

$$h_\infty(\mathbb{A}, \mathbb{B}) \triangleq \max\left\{h^0_\infty(\mathbb{A}, \mathbb{B}), h^0_\infty(\mathbb{B}, \mathbb{A})\right\}. \tag{3.4}$$

It is a distance for $\mathcal{C}(\mathbb{R}^n)$ as the following three requirements are satisfied

$$
\begin{array}{lll}
(i) & \text{separability} & h_\infty(\mathbb{A},\mathbb{B}) = 0 \Rightarrow \mathbb{A} = \mathbb{B}, \\
(ii) & \text{symmetry} & h_\infty(\mathbb{A},\mathbb{B}) = h_\infty(\mathbb{B},\mathbb{A}), \\
(iii) & \text{triangular inequality} & h_\infty(\mathbb{A},\mathbb{C}) \leqslant h_\infty(\mathbb{A},\mathbb{B}) + h_\infty(\mathbb{B},\mathbb{C}).
\end{array} \tag{3.5}
$$

Consider a compact set $\mathbb{A}$ and a point $\mathbf{a}$ far from $\mathbb{A}$. The set $\mathbb{A}_1 = \mathbb{A} \cup \{\mathbf{a}\}$ is also h_∞-far from $\mathbb{A}$. On the other hand, the set obtained by drilling small holes into $\mathbb{A}$ remains h_∞-close to $\mathbb{A}$. This illustrates the coarseness of the characterization of the differences between compact sets provided by the Hausdorff distance. A finer characterization will be needed to analyze the convergence properties of the algorithm SIVIA to be presented in Section 3.4.1.

Define the *complementary Hausdorff semi-distance* $\bar{h}_\infty$ between $\mathbb{A}$ and $\mathbb{B}$ of $\mathcal{C}(\mathbb{R}^n)$ as

$$
\begin{aligned}
\bar{h}_\infty(\mathbb{A},\mathbb{B}) &\triangleq h_\infty(\mathbb{R}^n\backslash\mathbb{A}, \mathbb{R}^n\backslash\mathbb{B}) \\
&= \max\left\{h^0_\infty(\mathbb{R}^n\backslash\mathbb{A}, \mathbb{R}^n\backslash\mathbb{B}), h^0_\infty(\mathbb{R}^n\backslash\mathbb{B}, \mathbb{R}^n\backslash\mathbb{A})\right\} \\
&= \max\left\{\bar{h}^0_\infty(\mathbb{A},\mathbb{B}), \bar{h}^0_\infty(\mathbb{B},\mathbb{A})\right\},
\end{aligned} \tag{3.6}
$$

where $\mathbb{R}^n\backslash\mathbb{A}$ is the complementary set of $\mathbb{A}$ in $\mathbb{R}^n$ and where $\bar{h}^0_\infty(\mathbb{A},\mathbb{B}) \triangleq h^0_\infty(\mathbb{R}^n\backslash\mathbb{A}, \mathbb{R}^n\backslash\mathbb{B})$. Figure 3.2 illustrates this definition. $\bar{h}^0_\infty(\mathbb{A},\mathbb{B})$ is obtained by deflating $\mathbb{B}$ until it is contained in $\mathbb{A}$; $\bar{h}^0_\infty(\mathbb{B},\mathbb{A})$ is obtained by deflating $\mathbb{A}$ until it is contained in $\mathbb{B}$. For the situation represented by Figure 3.2, $\bar{h}_\infty(\mathbb{A},\mathbb{B})$ is equal to $\bar{h}^0_\infty(\mathbb{A},\mathbb{B})$, as $\bar{h}^0_\infty(\mathbb{A},\mathbb{B})$ is larger than $\bar{h}^0_\infty(\mathbb{B},\mathbb{A})$. The operator $\bar{h}_\infty$ is a semi-distance on $\mathcal{C}(\mathbb{R}^n)$, because it does not satisfy the separability requirement to be a distance since $\bar{h}_\infty(\mathbb{A},\mathbb{B}) = 0$ whenever $\mathbb{A}$ and $\mathbb{B}$ are singletons.

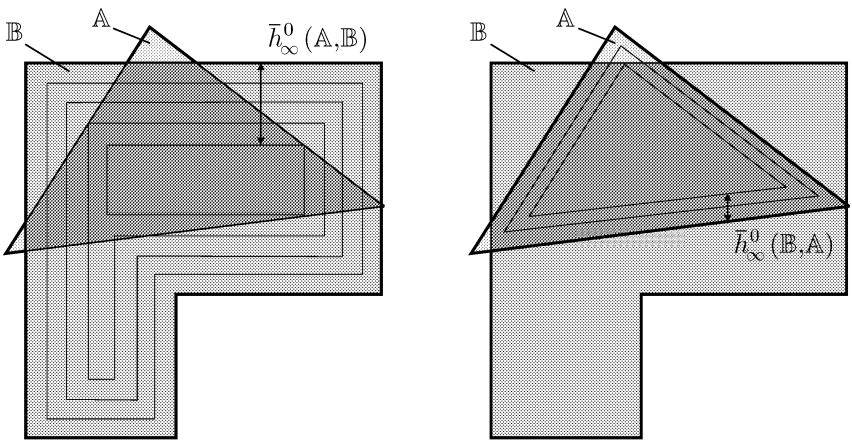


Fig. 3.2. The complementary Hausdorff semi-distance is equal to $\max\left\{\bar{h}^0_\infty(\mathbb{A},\mathbb{B}), \bar{h}^0_\infty(\mathbb{B},\mathbb{A})\right\}$

As h_∞, the complementary Hausdorff semi-distance $\bar{h}_\infty$ fails to give a fine characterization of the differences between compact sets; if the compact set obtained by drilling a single small hole in $\mathbb{A}$ is not $\bar{h}_\infty$-close to $\mathbb{A}$, the compact set obtained by adding to $\mathbb{A}$ a finite numbers of vectors far from $\mathbb{A}$ remains $\bar{h}_\infty$-close to $\mathbb{A}$.

Based on h_∞ and $\bar{h}_\infty$, a new distance can be defined (Jaulin and Walter, 1993c), which avoids the defects of each of them:

$$m_\infty(\mathbb{A},\mathbb{B}) \triangleq \max(h_\infty(\mathbb{A},\mathbb{B}),\bar{h}_\infty(\mathbb{A},\mathbb{B})). \tag{3.7}$$

Example 3.1 *Consider three compact subsets of* $\mathbb{R}$ *given by* $\mathbb{X}=[1,7]$, $\mathbb{Y}=[1,7]\cup[9-\varepsilon,9]$ *and* $\mathbb{Z}=[1,5]\cup[5+\varepsilon,7]$ *where* ε *is a positive number tending to zero. Then*

$$h_\infty(\mathbb{X},\mathbb{Y})=2;\ \bar{h}_\infty(\mathbb{X},\mathbb{Y})=\varepsilon/2;\ m_\infty(\mathbb{X},\mathbb{Y})=\max(2,\varepsilon/2)=2, \tag{3.8}$$

$$h_\infty(\mathbb{X},\mathbb{Z})=\varepsilon/2;\ \bar{h}_\infty(\mathbb{X},\mathbb{Z})=2;\ m_\infty(\mathbb{X},\mathbb{Z})=\max(\varepsilon/2,2)=2, \tag{3.9}$$

$$h_\infty(\mathbb{Y},\mathbb{Z})=2;\ \bar{h}_\infty(\mathbb{Y},\mathbb{Z})=2;\ m_\infty(\mathbb{Y},\mathbb{Z})=\max(2,2)=2. \tag{3.10}$$

The Hausdorff distance h_∞ *does not capture the difference between* $\mathbb{X}$ *and* $\mathbb{Z}$*, and the complementary Hausdorff semi-distance* $\bar{h}_\infty$ *does not capture the difference between* $\mathbb{X}$ *and* $\mathbb{Y}$*, but* m_∞ *captures both.* ■

3.2.2 Enclosure of compact sets between subpavings

A *subpaving* of a box $[\mathbf{x}]\subset\mathbb{R}^n$ is a union of non-overlapping subboxes of $[\mathbf{x}]$ with non-zero width. Two boxes in the same subpaving may have a non-empty intersection if they have a boundary in common, but their interiors must have an empty intersection. Subpavings can be employed to approximate compact sets in a guaranteed way. Computation on subpavings allows approximate computation on these compact sets, and forms the basic ingredient of the parameter and state estimation algorithms to be presented in Chapter 6.

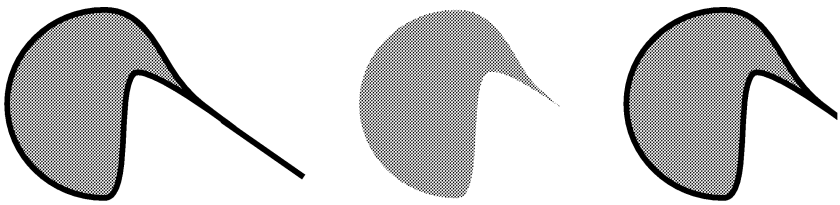
When a subpaving $\mathbb{P}$ of $[\mathbf{x}]$ covers $[\mathbf{x}]$, it is a *paving* of $[\mathbf{x}]$. The *accumulation set* of a subpaving $\mathbb{P}$ is the limit of the subset of $\mathbb{R}^n$ formed by the union of all boxes of $\mathbb{P}$ with width lower that ε when ε tends to zero. Since subpavings only contain boxes with non-zero width, the accumulation set of a finite subpaving is necessarily empty.

Let $(\mathcal{C}(\mathbb{R}^n),\subset,m_\infty)$ be the set of all compact sets of $\mathbb{R}^n$ equipped with the partial ordering $\subset$ and the distance m_∞. The set of finite subpavings is dense from outside in $(\mathcal{C}(\mathbb{R}^n),\subset,m_\infty)$, *i.e.*, for any compact set $\mathbb{X}$ we can find a subpaving $\overline{\mathbb{X}}$ containing $\mathbb{X}$ and as m_∞-close to $\mathbb{X}$ as desired. It may be impossible, however, to find a subpaving $\underline{\mathbb{X}}$ contained in $\mathbb{X}$. Consider, for instance, a segment of a line of $\mathbb{R}^2$. It can be approximated as closely as desired by a subpaving of $\mathbb{R}^2$ from the outside but not from the inside. To avoid this, we sometimes restrict consideration to the (large) class of *full compact sets*, *i.e.*, of compact sets that are equal to the closure of their

interiors. Figure 3.3 gives an example of a compact set that is not full. Denote by $\mathcal{C}_{\mathrm{f}}(\mathbb{R}^n)$ the set of all full compact sets of $\mathbb{R}^n$. The set of all finite subpavings of $\mathbb{R}^n$ is dense from inside and from outside in $(\mathcal{C}_{\mathrm{f}}(\mathbb{R}^n), \subset, m_\infty)$ (Jaulin and Walter, 1993c). Thus, for any full compact set $\mathbb{X}$, it is possible to find two finite subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ as m_∞-close to $\mathbb{X}$ as desired and such that $\underline{\mathbb{X}} \subset \mathbb{X} \subset \overline{\mathbb{X}}$. The set of compact sets

$$[\underline{\mathbb{X}}, \overline{\mathbb{X}}] \triangleq \left\{ \mathbb{X}' \in \mathcal{C}_{\mathrm{f}}(\mathbb{R}^n) \mid \underline{\mathbb{X}} \subset \mathbb{X}' \subset \overline{\mathbb{X}} \right\} \tag{3.11}$$

is then a neighbourhood of $\mathbb{X}$, the diameter $m_\infty\left(\underline{\mathbb{X}}, \overline{\mathbb{X}}\right)$ of which can be made as small as desired (Jaulin, 1994).



(a) (b) (c)

Fig. 3.3. (a): a compact set $\mathbb{A}$; (b): its interior $\mathbb{B}$; (c): the closure $\mathbb{C}$ of $\mathbb{B}$; since $\mathbb{A}$ and $\mathbb{C}$ differ, $\mathbb{A}$ is not full

3.3 Regular Subpavings

We shall first introduce some additional notation before defining subpavings more precisely and explaining how the useful class of regular subpavings can be represented in a computer. We shall then present a few elementary algorithms for the manipulation of regular subpavings.

Consider the box

$$[\mathbf{x}] = [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_n, \overline{x}_n] = [x_1] \times \cdots \times [x_n], \tag{3.12}$$

and take the index j of its first component of maximum width, *i.e.*,

$$j = \min\{ i \mid w([x_i]) = w([\mathbf{x}]) \}. \tag{3.13}$$

Define the boxes $L[\mathbf{x}]$ and $R[\mathbf{x}]$ as follows:

$$\begin{aligned} L[\mathbf{x}] &\triangleq [\underline{x}_1, \overline{x}_1] \times \cdots \times \left[\underline{x}_j, \left(\underline{x}_j + \overline{x}_j\right)/2\right] \times \cdots \times [\underline{x}_n, \overline{x}_n], \\ R[\mathbf{x}] &\triangleq [\underline{x}_1, \overline{x}_1] \times \cdots \times \left[\left(\underline{x}_j + \overline{x}_j\right)/2, \overline{x}_j\right] \times \cdots \times [\underline{x}_n, \overline{x}_n]. \end{aligned} \tag{3.14}$$

For instance, if $[\mathbf{x}] = [1,2] \times [2,4] \times [1,3]$, we get $w([\mathbf{x}]) = 2$, $j = 2$, $L[\mathbf{x}] = [1,2] \times [2,3] \times [1,3]$ and $R[\mathbf{x}] = [1,2] \times [3,4] \times [1,3]$. $L[\mathbf{x}]$ is the *left child* of

$[\mathbf{x}]$ and $R[\mathbf{x}]$ is the *right child* of $[\mathbf{x}]$. L and R may be viewed as operators from $\mathbb{IR}^n$ to $\mathbb{IR}^n$. The generation of these two children from $[\mathbf{x}]$ is called *bisection* of $[\mathbf{x}]$. The two boxes $L[\mathbf{x}]$ and $R[\mathbf{x}]$ are *siblings*. *Reunification* is the operation of merging two siblings $L[\mathbf{x}]$ and $R[\mathbf{x}]$ into their parent $[\mathbf{x}]$. This will be denoted by $[\mathbf{x}] := (L[\mathbf{x}] \mid R[\mathbf{x}])$.

3.3.1 Pavings and subpavings

A subpaving of $[\mathbf{x}]$ is *regular* if each of its boxes can be obtained from $[\mathbf{x}]$ by a finite succession of bisections and selections. Regular subpavings (Jaulin, 1994; Sam-Haroud and Faltings, 1996), also called n-trees (Samet, 1990), are a class of subsets of $\mathbb{R}^n$ easily manipulated with a computer as we shall see. Non-regular subpavings will also be used, but operations such as intersecting two subpavings will then become computationally much more demanding. Both types of subpavings share the ability to approximate full compact subsets of $\mathbb{R}^n$ as precisely as desired (see also Lozano-Pérez, 1981; Pruski, 1996; Pruski and Rohmer, 1997).

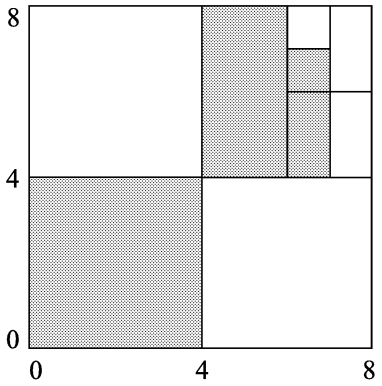


Fig. 3.4. Regular paving of a box; the boxes in grey form a regular subpaving

The set of all the regular subpavings of $[\mathbf{x}]$ will be denoted by $\mathcal{RSP}([\mathbf{x}])$. Figure 3.4 presents a regular paving $\mathbb{P}$ of the box $[\mathbf{x}] = [0,8] \times [0,8]$. The grey boxes form a regular subpaving $\mathbb{Q}$ of $[\mathbf{x}] = [0,8] \times [0,8]$. From any regular subpaving (or paving) $\mathbb{Q} \in \mathcal{RSP}([\mathbf{x}])$, define $L\mathbb{Q} \in \mathcal{RSP}(L[\mathbf{x}])$ as the regular subpaving that contains all the boxes of $\mathbb{Q}$ also included in $L[\mathbf{x}]$. Similarly, define $R\mathbb{Q} \in \mathcal{RSP}(R[\mathbf{x}])$ as the regular subpaving that contains all the boxes of $\mathbb{Q}$ also included in $R[\mathbf{x}]$. $L\mathbb{Q}$ and $R\mathbb{Q}$ are respectively the left and right children of $\mathbb{Q}$. If, for instance, $\mathbb{Q}$ is defined as in Figure 3.4, then

$$\begin{aligned} L\mathbb{Q} &= LL\,[\mathbf{x}] = [0,4] \times [0,4] \\ R\mathbb{Q} &= LRR\,[\mathbf{x}] \cup LLRRR\,[\mathbf{x}] \cup LLRRRR\,[\mathbf{x}] \\ &= [4,6] \times [4,8] \cup [6,7] \times [4,6] \cup [6,7] \times [6,7]. \end{aligned} \tag{3.15}$$

The box from which $\mathbb{Q}$ was derived by a succession of bisections and selections of boxes is the *root* of $\mathbb{Q}$. Thus, $\text{root}(\mathbb{Q}) = [\mathbf{x}]$, and $\text{root}(L\mathbb{Q}) = L\,[\mathbf{x}]$.

Remark 3.1 *The subpaving $\mathbb{Q}$ has a dual nature. It may be seen as a* subset *of $\mathbb{R}^2$, and we may write $[0,1]^2 \subset \mathbb{Q} \subset \mathbb{R}^2$. $\mathbb{Q}$ may also be viewed as a finite* list *of boxes*

$$\begin{aligned} &\{LL[\mathbf{x}], LRR[\mathbf{x}], LLRRR[\mathbf{x}], LLRRRR[\mathbf{x}]\} \\ &\quad = \{[0,4] \times [0,4]; [4,6] \times [4,8]; [6,7] \times [4,6]; [6,7] \times [6,7]\}. \end{aligned} \tag{3.16}$$

The notation $\mathbb{Q}$ will be used when the subpaving is considered as a set, and the notation $\mathcal{Q}$ will be used instead when the subpaving is viewed as a list of boxes. ■

Figure 3.5 illustrates the bracketing of the set

$$\mathbb{S} = \{(x,y) \,\big|\, x^2 + y^2 \in [1,2]\} \tag{3.17}$$

between subpavings with an increasing accuracy from left to right. The frame corresponds to the box $[-2,2] \times [-2,2]$. The subpaving $\Delta\mathbb{S}$ in grey contains the boundary of $\mathbb{S}$ whereas the subpaving $\underline{\mathbb{S}}$ in white is inside $\mathbb{S}$. Thus

$$\underline{\mathbb{S}} \subset \mathbb{S} \subset \overline{\mathbb{S}}, \text{ with } \overline{\mathbb{S}} \triangleq \underline{\mathbb{S}} \cup \Delta\mathbb{S}. \tag{3.18}$$

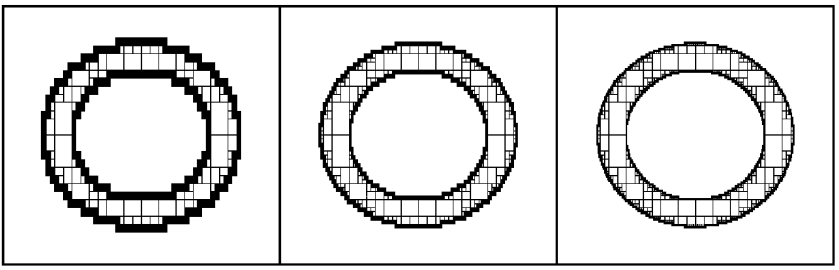


Fig. 3.5. Bracketing of a set between two subpavings; precision increases from left to right

3.3.2 Representing a regular subpaving as a binary tree

In a computer, a regular subpaving may be represented as a binary tree. A binary tree contains a finite set of *nodes*. This set may be empty, may contain

a single node, the *root* of the tree, or may contain two binary trees with an empty intersection, namely the *left* and *right subtrees*. Thus, the subpaving $\mathbb{Q}$ of Figure 3.4 is described by the binary tree of Figure 3.6, where 1 means that the corresponding node belongs to the subpaving. On this figure, A is the root of the tree. B and C are respectively its left and right children. They are siblings as they have the same parent node A. A has a left subtree and a right subtree; the right subtree of B is empty. A, B and C are nodes because they have at least one non-empty subtree. Finally, as D has no subtree, it is a degenerate node or *leaf*.

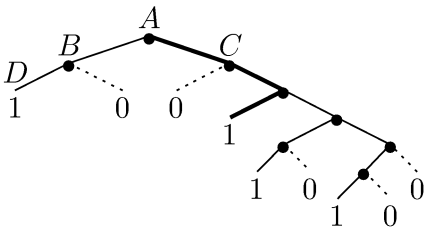


Fig. 3.6. Tree associated with the regular subpaving of Figure 3.4

The binary tree associated with a subpaving may be built from the list of its boxes. The growth of its branches is defined by how the initial box $[\mathbf{x}_0]$, which corresponds to the root of the tree, is bisected. Any non-degenerate node stands for a box that has been bisected. Any leaf indicates that the box it stands for belongs to the subpaving. For instance, the branch in bold on Figure 3.6 corresponds to the box $LRR\,[\mathbf{x}_0] = [4,6] \times [4,8]$. The *depth* of a box is the number of bisections necessary to get it from the root box. Thus, the depth of the box $[4,6] \times [4,8]$ is three.

A tree (or the corresponding subpaving) is *minimal* if it has no sibling leaves. Any non-minimal tree representative of a subpaving can be made minimal by discarding all sibling leaves so that their parents become leaves. This amounts to merging sibling boxes of the subpaving into single boxes.

Since the notions of binary tree and of regular subpaving are equivalent, the terminology of trees will also be employed for regular subpavings. In what follows, the representation of regular subpavings by binary trees will be used, because of the natural recursiveness of this data structure.

3.3.3 Basic operations on regular subpavings

The four basic operations to be considered are reuniting sibling subpavings, taking the union or the intersection of subpavings, and testing whether a box is included in a subpaving. All of them are facilitated by the use of binary

trees. For non-regular subpavings, they would be significantly more complicated. The computer implementation of regular subpavings is considered in Section 11.12, page 336.

Reuniting sibling subpavings: Consider a box $[\mathbf{x}]$ and two regular subpavings $\mathbb{X} \in \mathcal{RSP}\left(L[\mathbf{x}]\right)$ and $\mathbb{Y} \in \mathcal{RSP}\left(R[\mathbf{x}]\right)$. These subpavings have the same parent box $[\mathbf{x}]$. The *reunited* subpaving $\mathbb{Z} \triangleq (\mathbb{X} \mid \mathbb{Y}) \in \mathcal{RSP}([\mathbf{x}])$ is computed as follows:

Algorithm REUNITE(in: $\mathbb{X}, \mathbb{Y}$; out: $\mathbb{Z}$)	
1	if $\mathbb{X} = L[\mathbf{x}]$ and $\mathbb{Y} = R[\mathbf{x}]$, then $\mathbb{Z} := [\mathbf{x}]$;
2	else if $\mathbb{X} = \varnothing$ and $\mathbb{Y} = \varnothing$, then $\mathbb{Z} := \varnothing$;
3	else $L\mathbb{Z} := \mathbb{X}$ and $R\mathbb{Z} := \mathbb{Y}$.

When a binary tree representation is employed, each of the instructions in REUNITE is trivial to implement. For instance, the instructions $L\mathbb{Z} := \mathbb{X}$ and $R\mathbb{Z} := \mathbb{Y}$ amount to grafting the two trees $\mathbb{X}$ and $\mathbb{Y}$ to a node to form the tree $\mathbb{Z}$ (see Figure 3.7, case (ii)). Note that the number $\#\mathbb{Z}$ of boxes in the subpaving $\mathbb{Z}$ is not necessarily equal to $\#\mathbb{X}+\#\mathbb{Y}$. If, for instance, $[\mathbf{x}] = [0,2]^2$, $\mathbb{X} = [0,1] \times [0,2]$ and $\mathbb{Y} = [1,2] \times [0,2]$, then $\mathbb{X} = L[\mathbf{x}]$ and $\mathbb{Y} = R[\mathbf{x}]$. Thus $\#\mathbb{Z} = 1$ whereas $\#\mathbb{X} + \#\mathbb{Y} = 2$ (case (i) on Figure 3.7). In what follows, REUNITE$(\mathbb{X}, \mathbb{Y})$ will simply be written $(\mathbb{X} \mid \mathbb{Y})$. Note that reunification may be viewed as the inverse operation of applying L and R, since

$$\mathbb{Z} = (\mathbb{X} \mid \mathbb{Y}) \Leftrightarrow \mathbb{X} = L\mathbb{Z} \text{ and } \mathbb{Y} = R\mathbb{Z}. \tag{3.19}$$

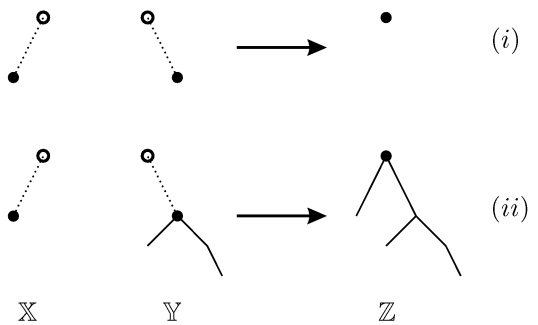


Fig. 3.7. Reuniting sibling subpavings; (i) sibling leaves, (ii) sibling subpavings

Intersecting subpavings: if $\mathbb{X} \in \mathcal{RSP}\left([\mathbf{x}]\right)$ and $\mathbb{Y} \in \mathcal{RSP}\left([\mathbf{x}]\right)$, then $\mathbb{Z} = \mathbb{X} \cap \mathbb{Y}$ is also a subpaving of $\mathcal{RSP}\left([\mathbf{x}]\right)$. It contains only the nodes shared by the binary trees representing $\mathbb{X}$ and $\mathbb{Y}$, and can be computed by the following recursive algorithm:

Algorithm INTER(in: $\mathbb{X}, \mathbb{Y}, [\mathbf{x}]$; out: $\mathbb{Z}$)
1 if $\mathbb{X} = \varnothing$ or $\mathbb{Y} = \varnothing$ then $\mathbb{Z} := \varnothing$;
2 else if $\mathbb{X} = [\mathbf{x}]$ then $\mathbb{Z} := \mathbb{Y}$;
3 else if $\mathbb{Y} = [\mathbf{x}]$ then $\mathbb{Z} := \mathbb{X}$;
4 else $\mathbb{Z} := (\text{INTER}(L\mathbb{X}, L\mathbb{Y}, L[\mathbf{x}]) \mid \text{INTER}(R\mathbb{X}, R\mathbb{Y}, R[\mathbf{x}]))$.

Taking the union of subpavings: if $\mathbb{X} \in \mathcal{RSP}([\mathbf{x}])$ and $\mathbb{Y} \in \mathcal{RSP}([\mathbf{x}])$, then $\mathbb{Z} = \mathbb{X} \cup \mathbb{Y}$ also belongs to $\mathcal{RSP}([\mathbf{x}])$. $\mathbb{Z}$ is computed by putting together all nodes of the two binary trees representing $\mathbb{X}$ and $\mathbb{Y}$. Again, this can be done recursively:

Algorithm UNION(in: $\mathbb{X}, \mathbb{Y}, [\mathbf{x}]$; out: $\mathbb{Z}$)
1 if $\mathbb{X} = \varnothing$ or $\mathbb{Y} = [\mathbf{x}]$ then $\mathbb{Z} := \mathbb{Y}$;
2 else if $\mathbb{Y} = \varnothing$ or $\mathbb{X} = [\mathbf{x}]$ then $\mathbb{Z} := \mathbb{X}$;
3 else $\mathbb{Z} := (\text{UNION}(L\mathbb{X}, L\mathbb{Y}, L[\mathbf{x}]) \mid \text{UNION}(R\mathbb{X}, R\mathbb{Y}, R[\mathbf{x}]))$.

Testing whether a box $[\mathbf{z}]$ is included in a subpaving $\mathbb{X}$ of $\mathcal{RSP}([\mathbf{x}])$. This test is straightforward in four cases. It holds true if $[\mathbf{z}]$ is empty, or if $\mathbb{X}$ is reduced to a single box $[\mathbf{x}]$ and $[\mathbf{z}] \subset [\mathbf{x}]$. It holds false if $\mathbb{X}$ is empty and $[\mathbf{z}]$ is not, or if $[\mathbf{z}]$ is not in the root box of $\mathbb{X}$. These basic tests will first be applied to the root of the tree representing the subpaving. If none of the four simple cases is satisfied, these basic tests are recursively applied on the left and right subtrees. The following algorithm summarizes the process:

Algorithm INSIDE(in: $[\mathbf{z}], \mathbb{X}$; out: t)
1 if $[\mathbf{z}] = \varnothing$ or if ($\mathbb{X}$ is a box $[\mathbf{x}]$ and $[\mathbf{z}] \subset [\mathbf{x}]$) then $t := 1$;
2 else if $\mathbb{X} = \varnothing$ then $t := 0$;
3 else $t := (\text{INSIDE}([\mathbf{z}] \cap L[\mathbf{x}], L\mathbb{X}) \cup \text{INSIDE}([\mathbf{z}] \cap R[\mathbf{x}], R\mathbb{X}))$.

When $[\mathbf{z}] \subset \mathbb{X}$, 1 is returned, when $[\mathbf{z}] \cap \mathbb{X} = \varnothing$, 0 is returned and when $[\mathbf{z}]$ overlaps the boundary of $\mathbb{X}$, $[0, 1]$ is returned.

Remark 3.2 *Many other algorithms operating on subpavings would be interesting to consider. For instance, the computation of the neighbours of a given box in a subpaving may be performed by Samet's algorithm (Samet, 1982). This algorithm could be very useful to study whether a subpaving is connected, as required in the context of path planning (see Section 8.3, page 234).* ■

3.4 Implementation of Set Computation

We shall now see how two important basic blocks of set computation can be implemented in an approximate but guaranteed way, based on the notions of

inclusion function and inclusion test presented in Chapter 2, and using regular subpavings as the basic class of objects to represent sets. The implementation of more ambitious set algorithms is deferred to the next chapters.

The first basic block to be considered is the computation of the reciprocal image

$$\mathbb{X} = \mathbf{f}^{-1}(\mathbb{Y}), \tag{3.20}$$

of a regular subpaving $\mathbb{Y}$ of $\mathbb{R}^m$ by a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$. We shall call this operation *set inversion*. A method to compute two subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ of $\mathbb{R}^n$ such that

$$\underline{\mathbb{X}} \subset \mathbb{X} \subset \overline{\mathbb{X}} \tag{3.21}$$

is proposed in Section 3.4.1.

The second basic block to be considered is the computation of the direct image

$$\mathbb{Y} = \mathbf{f}(\mathbb{X}), \tag{3.22}$$

of a subpaving $\mathbb{X}$ of $\mathbb{R}^n$ by a function $\mathbf{f}$. We shall call this operation *image evaluation*. An algorithm that computes an outer subpaving $\overline{\mathbb{Y}}$ for $\mathbb{Y}$ is proposed in Section 3.4.2. It will turn out that image evaluation is more difficult than set inversion. Moreover, up to now, no method to compute an inner approximation $\underline{\mathbb{Y}}$ of $\mathbb{Y}$ seems to be available, except *via* set inversion. This is why it may be advisable to recast image evaluation into the framework of set inversion when $\mathbf{f}$ is invertible.

3.4.1 Set inversion

Let $\mathbf{f}$ be a possibly non-linear function from $\mathbb{R}^n$ to $\mathbb{R}^m$ and let $\mathbb{Y}$ be a subset of $\mathbb{R}^m$ (for instance, a subpaving). Set inversion is the characterization of

$$\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{f}(\mathbf{x}) \in \mathbb{Y}\} = \mathbf{f}^{-1}(\mathbb{Y}). \tag{3.23}$$

For any $\mathbb{Y} \subset \mathbb{R}^m$ and for any function $\mathbf{f}$ admitting a convergent inclusion function $[\mathbf{f}](.)$, two regular subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ such that

$$\underline{\mathbb{X}} \subset \mathbb{X} \subset \overline{\mathbb{X}} \tag{3.24}$$

can be obtained with the algorithm SIVIA (Set Inverter Via Interval Analysis, Jaulin and Walter, 1993a and 1993c), to be described now.

SIVIA requires a (possibly very large) search box $[\mathbf{x}](0)$ to which $\overline{\mathbb{X}}$ is guaranteed to belong. To facilitate presentation, Figure 3.8 describes the basic steps of SIVIA, assuming that $\mathbb{Y}$ is a regular subpaving. The general procedure is easily derived from this simplified example. Four cases may be encountered.

1. If $[\mathbf{f}]([\mathbf{x}])$ has a non-empty intersection with $\mathbb{Y}$, but is not entirely in $\mathbb{Y}$, then $[\mathbf{x}]$ may contain a part of the solution set (Figure 3.8a); $[\mathbf{x}]$ is said to

be *undetermined*. If it has a width greater than a prespecified precision parameter ε, then it should be bisected (this implies the growth of two offspring from $[\mathbf{x}]$) and the test should be recursively applied to these newly generated boxes.

2. If $[\mathbf{f}]([\mathbf{x}])$ has an empty intersection with $\mathbb{Y}$, then $[\mathbf{x}]$ does not belong to $\mathbb{X}$ and can be cut off from the solution tree (Figure 3.8b).

3. If $[\mathbf{f}]([\mathbf{x}])$ is entirely in $\mathbb{Y}$, then $[\mathbf{x}]$ belongs to the solution subpaving $\mathbb{X}$, and is stored in $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ (Figure 3.8c).

4. The last case is depicted on Figure 3.8d. If the box considered is undetermined, but its width is lower than ε, then it is deemed small enough to be stored in the outer approximation $\overline{\mathbb{X}}$ of $\mathbb{X}$.

Table 3.1. Version of SIVIA based on an inclusion function

Algorithm SIVIA(in: $\mathbf{f}, \mathbb{Y}, [\mathbf{x}], \varepsilon$; inout: $\underline{\mathbb{X}}, \overline{\mathbb{X}}$)		
1	if $[\mathbf{f}]([\mathbf{x}]) \cap \mathbb{Y} = \varnothing$ return;	// Figure 3.8b
2	if $[\mathbf{f}]([\mathbf{x}]) \subset \mathbb{Y}$ then	
3	$\{\underline{\mathbb{X}} := \underline{\mathbb{X}} \cup [\mathbf{x}]$; $\overline{\mathbb{X}} := \overline{\mathbb{X}} \cup [\mathbf{x}]$; return;$\}$;	// Figure 3.8c
4	if $w([\mathbf{x}]) < \varepsilon$ then $\{\overline{\mathbb{X}} := \overline{\mathbb{X}} \cup [\mathbf{x}]$; return;$\}$;	// Figure 3.8d
5	SIVIA$(\mathbf{f}, \mathbb{Y}, L[\mathbf{x}], \varepsilon, \underline{\mathbb{X}}, \overline{\mathbb{X}})$; SIVIA$(\mathbf{f}, \mathbb{Y}, R[\mathbf{x}], \varepsilon, \underline{\mathbb{X}}, \overline{\mathbb{X}})$.	// Figure 3.8a

SIVIA is a recursive algorithm summarized by Table 3.1, where the subpavings $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$ have been initialized as empty.

The subpaving $\Delta\mathbb{X} \triangleq \overline{\mathbb{X}} \setminus \underline{\mathbb{X}}$ consisting of all boxes of $\overline{\mathbb{X}}$ that are not in $\underline{\mathbb{X}}$ is called the *uncertainty layer*. It is a regular subpaving, all boxes of which have a width smaller that ε.

Theorem 3.1 (Jaulin and Walter, 1993c) *If* $\mathbf{f}^{-1}$ *is* m_∞*-continuous around* $\mathbb{Y}$*, then when* ε *tends to zero*

$$\begin{array}{ll} (i) & \Delta\mathbb{X} \xrightarrow{\supset} \partial\mathbb{X}, \\ (ii) & \overline{\mathbb{X}} \xrightarrow{\supset} \mathbb{X}, \\ (iii) & \underline{\mathbb{X}} \xrightarrow{\subset} \mathbb{X} \text{ (if } \mathbb{X} \text{ is full)}, \end{array} \tag{3.25}$$

where $\xrightarrow{\supset}$ *and* $\xrightarrow{\subset}$ *respectively mean the* h_∞*-convergence from without and within and where* $\partial\mathbb{X}$ *denotes the boundary of the compact set* $\mathbb{X}$. ■

Provided that $\mathbb{X}$ is full, this theorem means that, the pair $[\underline{\mathbb{X}}, \overline{\mathbb{X}}]$ defines a neighbourhood of $\mathbb{X}$ with a diameter that can be chosen arbitrarily small. SIVIA terminates after less than

$$\left(\frac{w([\mathbf{x}](0))}{\varepsilon} + 1\right)^n$$

bisections and the computing time increases exponentially with the dimension of $\mathbf{x}$ (Jaulin and Walter, 1993a). When one is only interested in computing

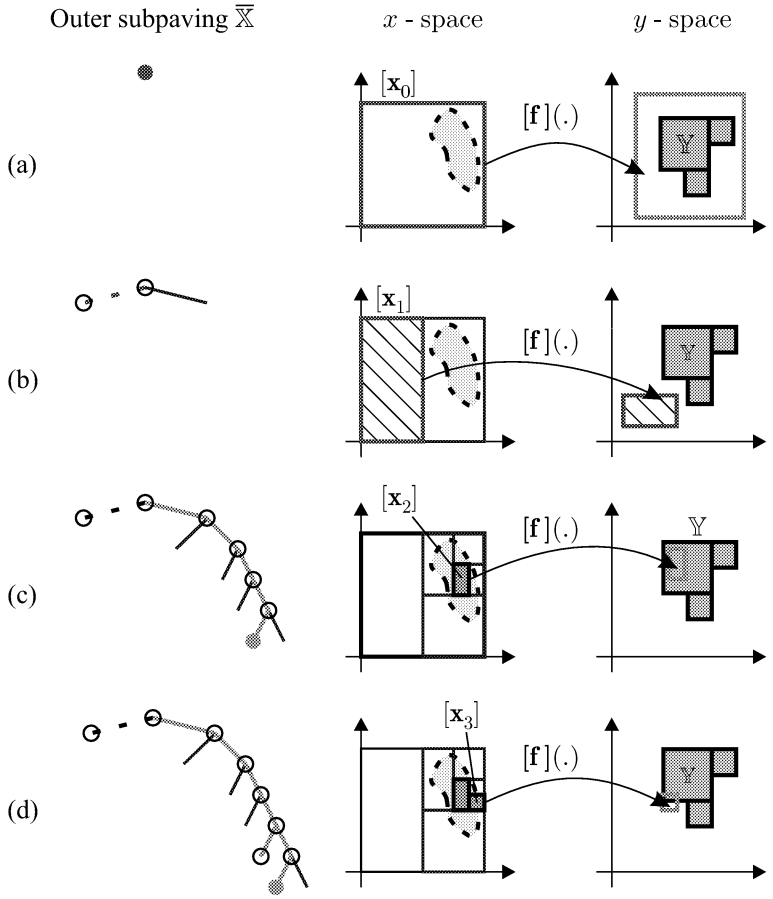


Fig. 3.8. Four situations encountered by SIVIA; in the second column, the set in light grey is the set $\mathbb{X} = \mathbf{f}^{-1}(\mathbb{Y})$ to be characterized; (a) the box $[\mathbf{x}_0]$ to be checked is undetermined and will be bisected; (b) the box $[\mathbf{f}]([\mathbf{x}_1])$ does not intersect $\mathbb{Y}$ and $[\mathbf{x}_1]$ is rejected; (c) the box $[\mathbf{f}]([\mathbf{x}_2])$ is entirely in $\mathbb{Y}$ and $[\mathbf{x}_2]$ is stored in $\underline{\mathbb{X}}$ and $\overline{\mathbb{X}}$; (d) the box $[\mathbf{x}_3]$ is undetermined but deemed too small to be bisected, it is stored in $\overline{\mathbb{X}}$ but not in $\underline{\mathbb{X}}$

a given characteristic of $\mathbb{X}$ such as its interval hull $[\mathbb{X}]$ or its volume, only the recursivity stack takes a significant place in the memory. This place is extraordinarily small, as it can be proved (Jaulin and Walter, 1993a) that

$$\#\text{stack} \leqslant n\left(\log_2\left(w\left([\mathbf{x}]\left(0\right)\right)\right) - \log_2\left(\varepsilon\right) + 1\right). \tag{3.26}$$

For instance, for $n = 100$, $w\left([\mathbf{x}]\left(0\right)\right) = 10^4$ and $\varepsilon = 10^{-10}$, (3.26) implies that $\#\text{stack} \leqslant 100\left(\log_2\left(10^4\right) - \log_2\left(10^{-10}\right) + 1\right) = 4751$.

The algorithm can be generalized to the case where the search space, which was assumed here to be a box $[\mathbf{x}](0)$, is replaced by a more general subpaving (Kieffer, 1999; **?**).

Table 3.2. Version of SIVIA based on an inclusion test

Algorithm SIVIA(in: $t, [\mathbf{x}], \varepsilon$; inout: $\underline{\mathbb{X}}, \overline{\mathbb{X}}$)	
1	if $[t]([\mathbf{x}]) = 0$ return;
2	if $[t]([\mathbf{x}]) = 1$ then $\{\underline{\mathbb{X}} := \underline{\mathbb{X}} \cup [\mathbf{x}]$; $\overline{\mathbb{X}} := \overline{\mathbb{X}} \cup [\mathbf{x}]$; return;$\}$;
3	if $w([\mathbf{x}]) < \varepsilon$ then $\{\overline{\mathbb{X}} := \overline{\mathbb{X}} \cup [\mathbf{x}]$; return;$\}$;
4	SIVIA$(t, L[\mathbf{x}], \varepsilon, \underline{\mathbb{X}}, \overline{\mathbb{X}})$; SIVIA$(t, R[\mathbf{x}], \varepsilon, \underline{\mathbb{X}}, \overline{\mathbb{X}})$.

SIVIA can also be presented with an inclusion test $[t](.)$ taking its values in $\{0, 1, [0, 1]\}$ in place of $[\mathbf{f}]$ and $\mathbb{Y}$, as indicated in Table 3.2. Both algorithms are initialized in the same way.

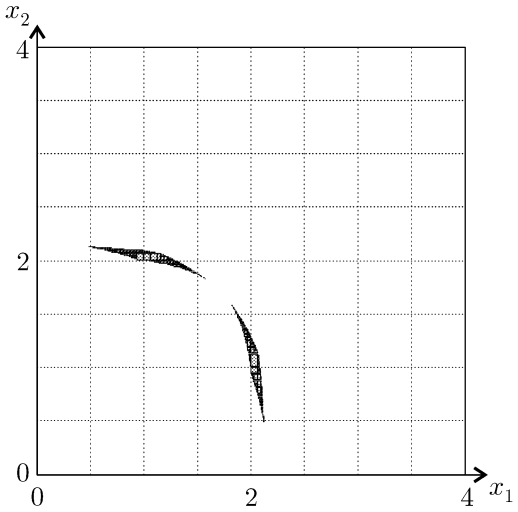


Fig. 3.9. Subpaving obtained with SIVIA for Example 3.2

Example 3.2 *Let $\mathbb{X}$ be the set of all $\mathbf{x}$s in $\mathbb{R}^2$ that satisfy*

$$\begin{cases} \exp(x_1) + \exp(x_2) & \in [10, 11], \\ \exp(2x_1) + \exp(2x_2) & \in [62, 72]. \end{cases} \tag{3.27}$$

Characterizing $\mathbb{X}$ is a set-inversion problem, as

$$\mathbb{X} = \mathbf{f}^{-1}([10, 11] \times [62, 72]), \tag{3.28}$$

with

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} \exp(x_1) + \exp(x_2) \\ \exp(2x_1) + \exp(2x_2) \end{pmatrix}. \tag{3.29}$$

For $[\mathbf{x}](0) = [0,4] \times [0,4]$ *and* $\varepsilon = 0.01$, SIVIA *yields the regular subpaving* $\overline{\mathbb{X}}$ *described by Figure 3.9 in less than* 2 *s on a* PENTIUM 133. ■

By using some of the contractors to be presented in Chapter 4, it may be possible to improve the quality of the description of a set obtained with SIVIA for a given number of bisections. The price to be paid is that the resulting subpaving may no longer be regular.

3.4.2 Image evaluation

Computing the direct image of a subpaving by a function is more complicated than computing a reciprocal image, because interval analysis does not directly provide any inclusion test for the point test $t(\mathbf{y}) = (\mathbf{y} \in \mathbf{f}(\mathbb{X}))$. Note that even this point test is very difficult to evaluate in general, contrary to the point test $t(\mathbf{x}) = \left(\mathbf{x} \in \mathbf{f}^{-1}(\mathbb{Y})\right)$ involved in set inversion. Indeed, to test whether $\mathbf{x} \in \mathbf{f}^{-1}(\mathbb{Y})$, it suffices to compute $\mathbf{f}(\mathbf{x})$ and to test whether the result belongs to $\mathbb{Y}$. On the other hand, to test whether $\mathbf{y} \in \mathbf{f}(\mathbb{X})$, one must study whether the set of equations $\mathbf{f}(\mathbf{x}) = \mathbf{y}$ admits at least one solution under the constraint $\mathbf{x} \in \mathbb{X}$, which is usually far from simple.

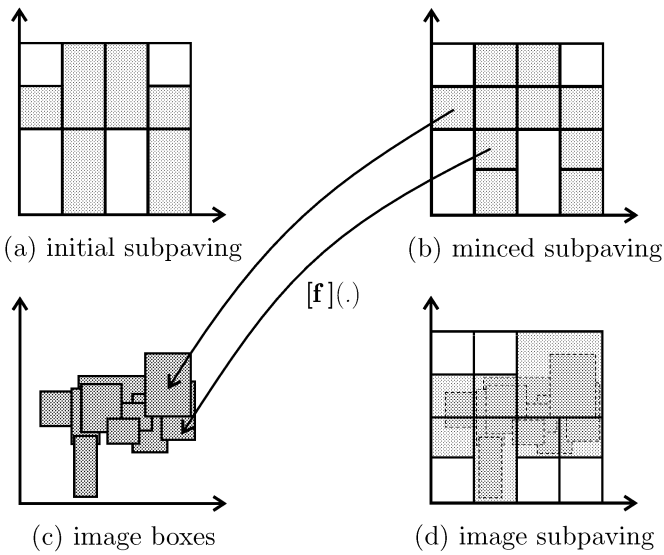


Fig. 3.10. The three steps of IMAGESP. (a) → (b): mincing; (b) → (c): evaluation; (c) → (d): regularization

Assume that $\mathbf{f}$ is continuous and that a convergent inclusion function $[\mathbf{f}]$ for $\mathbf{f}$ is available. The algorithm to be presented generates a regular subpaving $\overline{\mathbb{Y}}$ that contains the image $\mathbb{Y}$ of a regular subpaving $\mathbb{X}$ by $\mathbf{f}$ (see also Kieffer et al., 1998, **?**). The set $\mathbb{Y}$ is included in the box $[\mathbf{f}]([\mathbb{X}])$, *i.e.*, in the image by the inclusion function $[\mathbf{f}]$ of the smallest box containing $\mathbb{X}$. The algorithm proceeds in three steps, namely *mincing*, *evaluation*, and *regularization* (Figure 3.10). As with SIVIA, the precision of the outer approximation will be governed by the real $\varepsilon > 0$ to be chosen by the user. During mincing, a non-minimal regular subpaving $\mathbb{X}_\varepsilon$ is built, such that the width of each of its boxes is less than ε. During evaluation, a box $[\mathbf{f}]([\mathbf{x}])$ is computed for each box $[\mathbf{x}]$ of $\mathbb{X}_\varepsilon$, and all the resulting boxes are stored into a list $\mathcal{U}$. During regularization, a regular subpaving $\overline{\mathbb{Y}}$ is computed that contains the union $\mathbb{U}$ of all the boxes of $\mathcal{U}$. This regularization can be viewed as a call of SIVIA to invert $\mathbb{U}$ by the identity function, taking advantage of the fact that $\mathbf{f}(\mathbb{X}) \subset \mathbb{U}$ is equivalent to $\mathbf{f}(\mathbb{X}) \subset \mathrm{Id}^{-1}(\mathbb{U})$.

The resulting algorithm is described in Table 3.3.

Table 3.3. Algorithm for image evaluation based on subpavings

Algorithm IMAGESP(in: $\mathbf{f}$,$\mathbb{X}$, ε; out: $\overline{\mathbb{Y}}$)	
1	$\mathbb{X}_\varepsilon := \mathrm{mince}(\mathbb{X}, \varepsilon)$;
2	$\mathcal{U} := \varnothing$; // $\mathcal{U}$ is a list and $\mathbb{U}$ is the set of the boxes in $\mathcal{U}$
3	for each $[\mathbf{x}] \in \mathbb{X}_\varepsilon$, add $[\mathbf{f}]([\mathbf{x}])$ to the list $\mathcal{U}$;
4	SIVIA($\mathbf{y} \in \mathbb{U}, [\mathbf{f}]([\mathbb{X}]), \varepsilon, \underline{\mathbb{Y}}, \overline{\mathbb{Y}}$). // SIVIA of Table 3.2

The complexity and convergence properties of IMAGESP have been studied in Kieffer (1999).

Remark 3.3 *As only the outer approximation $\overline{\mathbb{Y}}$ is returned by* IMAGESP, *one may of course use a simplified version of* SIVIA *without the computation of an inner approximation $\underline{\mathbb{Y}}$.* ■

Remark 3.4 *Since $\mathbb{U}$ is not a subpaving, implementation is not trivial, see Section 11.12.3, page 342, for details.* ■

Theorem 3.2 (Jaulin, 2000b; **?**) *If $\mathbf{f}$ admits a convergent inclusion function $[\mathbf{f}]$, then the sets $\overline{\mathbb{Y}}$ and $\mathbb{U}$ evaluated by* IMAGESP$([\mathbf{f}], \mathbb{X}, \varepsilon)$ *satisfy the following properties:*

(i) $\mathbf{f}(\mathbb{X}) \subset \overline{\mathbb{Y}}$,

(ii) $h_\infty(\mathbb{U}, \mathbf{f}(\mathbb{X})) \leqslant \max_{[\mathbf{x}] \in \mathbb{X}_\varepsilon} w([\mathbf{f}]([\mathbf{x}]))$,

(iii) $h_\infty(\mathbb{U}, \mathbf{f}(\mathbb{X})) \to 0$ *when* $\varepsilon \to 0$,

(iv) $h_\infty\left(\overline{\mathbb{Y}}, \mathbf{f}(\mathbb{X})\right) \to 0$ *when* $\varepsilon \to 0$.

■

Example 3.3 *Consider the regular subpaving* $\overline{\mathbb{X}}$ *of Figure 3.11a. This subpaving covers the set*

$$\mathbb{X} = \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1^2 + x_2^2 \in [1, 2]\}. \tag{3.30}$$

ImageSp *is used to compute an outer approximation of the image of* $\mathbb{X}$ *by the function*

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1 x_2 \\ x_1 + x_2 \end{pmatrix}. \tag{3.31}$$

Mincing generates the regular subpaving $\overline{\mathbb{X}}_\varepsilon$ *of Figure 3.11b. Note that although* $\overline{\mathbb{X}}_\varepsilon$ *and* $\overline{\mathbb{X}}$ *represent exactly the same set* ($\overline{\mathbb{X}}_\varepsilon = \overline{\mathbb{X}}$)*, they do not contain the same lists of boxes* ($\overline{\mathcal{X}}_\varepsilon \neq \overline{\mathcal{X}}$) *since the number of boxes in* $\mathbb{X}_\varepsilon$ *is larger than that of* $\mathbb{X}$*, see Remark 3.1. The evaluation step generates a list* $\mathcal{Y}$ *of boxes and the union of these boxes contains* $\mathbf{f}(\mathbb{X}_\varepsilon)$ *(Figure 3.11c). Finally, regularization yields the regular subpaving* $\overline{\mathbb{Y}}$ *of Figure 3.11d.* ■

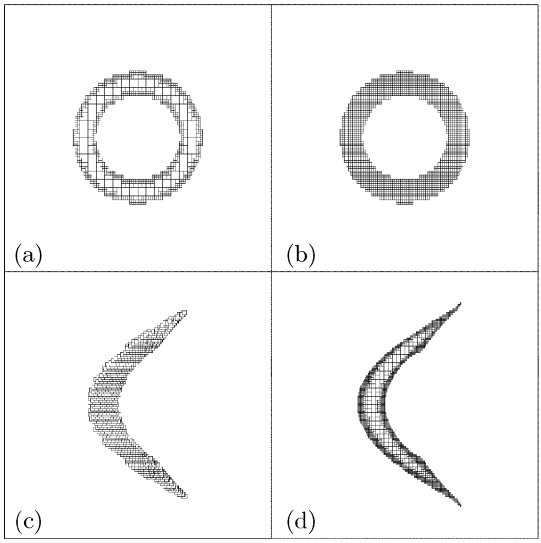


Fig. 3.11. Principle of ImageSp; (a) initial subpaving, (b) minced subpaving, (c) evaluated set, (d) regularized subpaving containing the evaluated set; all frames correspond to the box $[-3, 3] \times [-3, 3]$

The next example combines the use of Sivia and ImageSp.

Example 3.4 *This example is divided into three parts. The first one is the characterization of the set*

$$\mathbb{X}_1 = \left\{(x_1, x_2) \in \mathbb{R}^2 \,\middle|\, x_1^4 - x_1^2 + 4x_2^2 \in [-0.1, 0.1]\right\}. \tag{3.32}$$

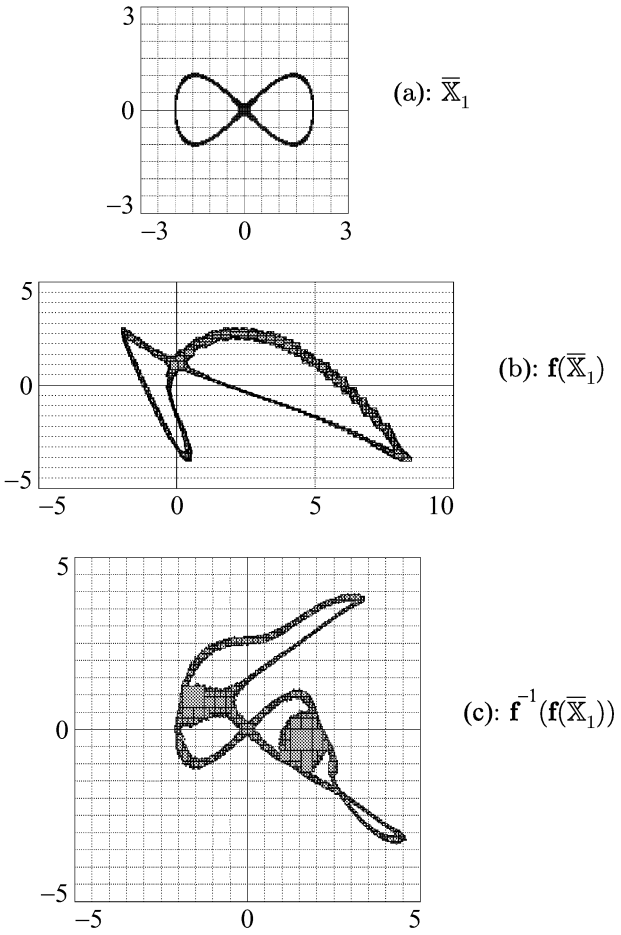


Fig. 3.12. Inverse and direct image evaluations

This set-inversion problem is solved by SIVIA *in the search box* $[\mathbf{x}]_1 = [-3,3]\times[-3,3]$ *for* $\varepsilon = 0.1$. *The resulting subpaving* $\overline{\mathbb{X}}_1$ *is represented on Figure 3.12a. The second part consists in the evaluation of an outer approximation of the image* $\mathbb{X}_2$ *of* $\overline{\mathbb{X}}_1$ *by the function*

$$\mathbf{f}(x_1,x_2) = \begin{pmatrix} (x_1-1)^2 - 1 + x_2 \\ -x_1^2 + (x_2-1)^2 \end{pmatrix}.$$

With $\varepsilon = 0.1$, IMAGESP *yields the subpaving* $\overline{\mathbb{X}}_2$ *depicted on Figure 3.12b. The last part is the characterization of the image of* $\overline{\mathbb{X}}_2$ *by the inverse of* $\mathbf{f}(.)$, i.e., *of* $\mathbb{X}_3 = \left\{\mathbf{f}^{-1}\left(\overline{\mathbb{X}}_2\right)\right\}$. *The function* $\mathbf{f}(.)$ *is not invertible (in the common sense) in* $\mathbb{R}^2$. *Thus, an explicit form of* $\mathbf{f}^{-1}(.)$ *is not available for*

the whole search domain and the problem is treated as a set inversion problem. SIVIA *is thus used in the search box* $[\mathbf{x}]_3 = [-5,5] \times [-5,5]$, *again for* $\varepsilon = 0.1$. *The outer subpaving* $\overline{\mathbb{X}}_3$ *is represented on Figure 3.12c. We have* $\overline{\mathbb{X}}_1 \subset \mathbf{f}^{-1}\left(\mathbf{f}\left(\overline{\mathbb{X}}_1\right)\right) \subset \overline{\mathbb{X}}_3$. *The initial set* $\overline{\mathbb{X}}_1$ *is clearly present in* $\overline{\mathbb{X}}_3$. *The result is slightly fatter, due to error accumulation during inverse and direct image evaluation. Additional parts have appeared because* $\mathbf{f}(.)$ *is only invertible in a set-theoretic sense.* ■

3.5 Conclusions

The notion of subpaving introduced in this chapter makes it possible to obtain, store and manipulate inner and outer approximations of compact sets. Subpavings will form a useful class of objects on which computations will be performed in what follows. Two basic algorithms have been presented to perform direct and inverse evaluation of functions on subpavings. The problem of getting an inner approximation for image sets is still open when the function is not invertible, because of the impossibility of casting the problem in the framework of set inversion.

Regular subpavings are simpler to store and manipulate than generic subpavings, but form an expensive representation of sets in terms of memory space. They are thus adapted to low-dimensional problems. For sets of higher dimension, the requirement of regularity of the subpavings may be relaxed to allow the use of contractors, presented in the next chapter.

4. Contractors

4.1 Introduction

Consider n_{x} variables $x_i \in \mathbb{R},\ i \in \{1, \ldots, n_{\mathrm{x}}\}$, linked by n_{f} relations (or constraints) of the form

$$f_j(x_1, x_2, \ldots, x_{n_{\mathrm{x}}}) = 0,\ j \in \{1, \ldots, n_{\mathrm{f}}\}\,. \tag{4.1}$$

Each variable x_i is known to belong to a *domain* $\mathbb{X}_i$. For simplicity, these domains will be intervals, denoted by $[x_i]$, but unions of intervals could be considered as well. Define the vector $\mathbf{x}$ as

$$\mathbf{x} = (x_1, \ldots, x_{n_{\mathrm{x}}})^{\mathrm{T}}, \tag{4.2}$$

and the prior domain for $\mathbf{x}$ as

$$[\mathbf{x}] = [x_1] \times \cdots \times [x_{n_{\mathrm{x}}}]. \tag{4.3}$$

Let $\mathbf{f}$ be the function whose coordinate functions are the f_js. Equation 4.1 can then be written in vector form as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. This corresponds to a *constraint satisfaction problem* (CSP) $\mathcal{H}$, which can be formulated as

$$\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0},\ \mathbf{x} \in [\mathbf{x}])\,. \tag{4.4}$$

The *solution set* of $\mathcal{H}$ is defined as

$$\mathbb{S} = \{\mathbf{x} \in [\mathbf{x}] \mid \mathbf{f}(\mathbf{x}) = \mathbf{0}\}. \tag{4.5}$$

Such CSPs may involve equality and inequality constraints. For instance, the set of constraints

$$\begin{cases} x_1 + \sin(x_2) \leqslant 0, \\ x_1 - x_2 = 3, \end{cases} \tag{4.6}$$

can be cast into the CSP framework by introducing a *slack variable* x_3 to get the set of constraints:

$$\begin{cases} x_1 + \sin(x_2) + x_3 = 0, \\ x_1 - x_2 - 3 = 0, \end{cases} \tag{4.7}$$

where the domains for the variables are $[x_3] = [0, \infty[$, $[x_1] = \mathbb{R}$, $[x_2] = \mathbb{R}$ and the coordinate functions f_j are $f_1(\mathbf{x}) = x_1 + \sin(x_2) + x_3$ and $f_2(\mathbf{x}) =$

L. Jaulin et al., *Applied Interval Analysis*

$x_1 - x_2 - 3$. Characterizing the solution set $\mathbb{S}$ is NP-hard in general, which means that no algorithm with a complexity polynomial in the number of variables is available to obtain an accurate approximation of $\mathbb{S}$ in the worst case.

Originally, CSPs were defined for discrete domains, *i.e.*, the values taken by the x_is belonged to finite sets (Clowes, 1971; Waltz, 1975; Mackworth, 1977a, 1977b; Freuder, 1978; Mackworth and Freuder, 1985; Dechter and Dechter, 1987). Later, CSPs were extended to continuous domains (subsets of $\mathbb{R}$ or intervals) (Cleary, 1987; Davis, 1987; Hyvönen, 1992; Lhomme, 1993; Benhamou et al., 1994; Haroud et al., 1995; Sam-Haroud, 1995; Sam-Haroud and Faltings, 1996; Deville et al., 1997; van Hentenryck et al., 1998; Lottaz et al., 1998). Most of the algorithms presented in these papers use consistency techniques such as those described below to find an outer approximation of the set $\mathbb{S}$ of all solutions of $\mathcal{H}$. The main advantage of these techniques is that they yield a guaranteed enclosure of $\mathbb{S}$ with a complexity that can be kept polynomial in time and space.

Contracting $\mathcal{H}$ means replacing $[\mathbf{x}]$ by a smaller domain $[\mathbf{x}']$ such that the solution set remains unchanged, *i.e.*, $\mathbb{S} \subset [\mathbf{x}'] \subset [\mathbf{x}]$. There exists an optimal contraction of $\mathcal{H}$, which corresponds to replacing $[\mathbf{x}]$ by the smallest box that contains $\mathbb{S}$. A *contractor* for $\mathcal{H}$ is any operator that can be used to contract it. In order to keep the time and space complexity polynomial, contractors will not be allowed to bisect domains. A contractor will be denoted by $\mathcal{C}$, with a subscript indicating the principle on which it is based. The contractors to be presented in this chapter are enumerated in Table 4.1.

Table 4.1. Contractors to be presented

Contractor	**Based on**	**Section**
$\mathcal{C}_{\mathrm{GE}}(\mathbf{Ap} - \mathbf{b} = \mathbf{0}, [\mathbf{A}], [\mathbf{p}], [\mathbf{b}])$	Gauss elimination	4.2.2
$\mathcal{C}_{\mathrm{GS}}(\mathbf{Ap} - \mathbf{b} = \mathbf{0}, [\mathbf{A}], [\mathbf{p}], [\mathbf{b}])$	Gauss–Seidel algorithm	4.2.3
$\mathcal{C}_{\mathrm{K}}(\mathbf{f}(\mathbf{x}) = \mathbf{0}, [\mathbf{x}])$	Krawczyk method	4.2.3
$\mathcal{C}_{\downarrow\uparrow}(\mathbf{f}(\mathbf{x}) = \mathbf{0}, [\mathbf{x}])$	forward–backward propagation	4.2.4
$\mathcal{C}_{\mathrm{LP}}(\mathbf{Ap} - \mathbf{b} = \mathbf{0}, [\mathbf{A}], [\mathbf{p}], [\mathbf{b}])$	linear programming	4.2.5
$\mathcal{C}_{\mathrm{GSP}}(\mathbf{Ap} - \mathbf{b} = \mathbf{0}, [\mathbf{A}], [\mathbf{p}], [\mathbf{b}])$	Gauss–Seidel with preconditioning	4.3.2
$\mathcal{C}_{\mathrm{GEP}}(\mathbf{Ap} - \mathbf{b} = \mathbf{0}, [\mathbf{A}], [\mathbf{p}], [\mathbf{b}])$	Gauss elimination with preconditioning	4.3.2
$\mathcal{C}_{\mathrm{N}}(\mathbf{f}(\mathbf{x}) = \mathbf{0}, [\mathbf{x}])$	Newton with preconditioning	4.3.3
$\mathcal{C}_{\parallel}(\mathbf{f}(\mathbf{x}) = \mathbf{0}, [\mathbf{x}])$	parallel linearization	4.3.4

Remark 4.1 *It is sometimes convenient to distinguish several types of domains for uncertain quantities, such as* $[\mathbf{A}]$, $[\mathbf{b}]$ *and* $[\mathbf{p}]$ *in Table 4.1, instead of pooling all of them in a single box* $[\mathbf{x}]$. ■

The basic contractors are $\mathcal{C}_{\rm GE}$, $\mathcal{C}_{\rm GS}$, $\mathcal{C}_{\rm K}$, $\mathcal{C}_{\downarrow\uparrow}$ and $\mathcal{C}_{\rm LP}$, to be presented in Section 4.2. These contractors are efficient on specific classes of problems only. Section 4.3 presents some tools to transform a CSP so that basic contractors become more widely applicable. Incorporating the transformation in the contraction procedure will yield new contractors ($\mathcal{C}_{\rm GSP}$, $\mathcal{C}_{\rm GEP}$, $\mathcal{C}_{\rm N}$ and $\mathcal{C}_{\|}$), able to deal with a much larger class of problems. In Section 4.4, all available contractors are made to collaborate in order to increase their efficiency. Section 4.5 presents the notion of contractor for sets. This notion does not bring anything new from a methodological viewpoint, but allows one to deal easily with contractors independently of the type of the constraints that define the set of interest. In the next chapter we shall see how contractors can be used to get efficient solvers to deal with various optimization problems, and with the resolution of systems of equations and inequalities.

4.2 Basic Contractors

In this section, some basic contractors will be presented. Some of them are interval counterparts of classical point algorithms such as the Gauss elimination, Gauss–Seidel and Newton algorithms. Others use constraint propagation. Each of them is efficient only for specific CSPs, but a suitable combination of these contractors, possibly supplemented with some formal transformation or preconditioning, makes them efficient for a much larger class of CSPs, as we shall see in Section 4.3. To prove useful, it suffices that a contractor extends the class of CSPs that can be handled efficiently, even if there are still some CSPs for which it turns out to fail.

Section 4.2.1 presents the notion of finite subsolvers used in Section 4.2.2 to build contractors by intervalization. Section 4.2.3 presents contractors obtained by intervalization of fixed-point methods. A contractor based on constraint propagation will be described in Section 4.2.4. The last basic contractor, to be presented in Section 4.2.5, takes advantage of linear programming techniques.

4.2.1 Finite subsolvers

Roughly speaking, a *finite subsolver* of the CSP $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ is a finite algorithm to compute the values of some variables x_i when some other variables x_j are known. Figure 4.1 illustrates a finite subsolver ϕ computing x_8 and x_9 from x_1, x_2, x_3 and x_4. The formal definition of a subsolver will require the definition of a subvector of a given vector.

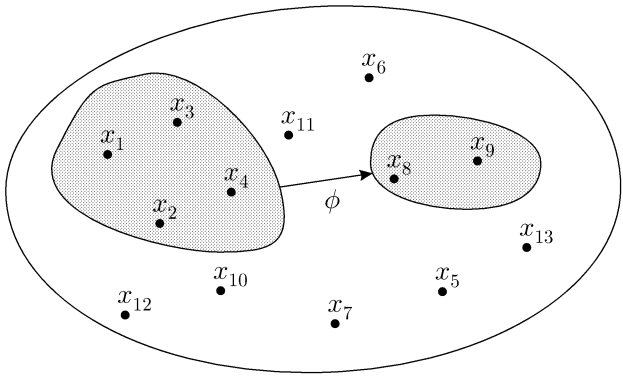


Fig. 4.1. Subsolver computing x_8 and x_9 when x_1, x_2, x_3 and x_4 are known

Definition 4.1 *The vector* $\mathbf{u} = (u_1, \ldots, u_{n_u})^T$ *is a* subvector *of* $\mathbf{x} = (x_1, \ldots, x_{n_x})^T$, *if there exists a subset* $\mathcal{I} = \{i_1, \ldots, i_{n_u}\}$ *of* $\{1, \ldots, n_x\}$ *such that* $\mathbf{u} = (x_{i_1}, \ldots, x_{i_{n_u}})^T$. $\mathcal{I}$ *is then called the* index set *of* $\mathbf{u}$, *and we shall write* $\mathbf{u} = \mathbf{x}_\mathcal{I}$. ■

Definition 4.2 *Consider* $\mathcal{I} = \{i_1, \ldots, i_{n_u}\}$ *and* $\mathcal{J} = \{j_1, \ldots, j_{n_\phi}\}$, *two index sets such that* $\mathcal{I} \cap \mathcal{J} = \varnothing$, *and two subvectors* $\mathbf{u} = \mathbf{x}_\mathcal{I}$ *and* $\mathbf{v} = \mathbf{x}_\mathcal{J}$ *of the same vector* $\mathbf{x}$. *A* finite subsolver *associated with* $\mathcal{H}$ *is a* finite *set-valued algorithm* $\boldsymbol{\phi} : \mathbf{u} \mapsto \boldsymbol{\phi}(\mathbf{u})$ *such that the following implication holds true:*

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \Rightarrow \mathbf{v} \in \boldsymbol{\phi}(\mathbf{u}). \tag{4.8}$$

The components of $\mathbf{u}$ *are called the* inputs *of* $\boldsymbol{\phi}$ *and those of* $\mathbf{v}$ *are called its* outputs. *In the situation represented in Figure 4.1, the inputs are* x_1, x_2, x_3, *and* x_4, *and the outputs are* x_8 *and* x_9. ■

Often, $\boldsymbol{\phi}(\mathbf{u})$ is a singleton, and the membership relation on the right-hand side of (4.8) can be understood as the equation $\mathbf{v} = \boldsymbol{\phi}(\mathbf{u})$. The following example shows that defining $\boldsymbol{\phi}$ as a set-valued function can be useful and illustrates the concept of subsolver.

Example 4.1 *Consider the CSP*

$$\mathcal{H} : \begin{pmatrix} x_1 x_2 - x_3 = 0 \\ x_2 - \sin(x_4) = 0 \\ [x_1] = [x_2] =]-\infty, 0], \ [x_3] = [x_4] = \mathbb{R} \end{pmatrix}. \tag{4.9}$$

Many subsolvers of $\mathcal{H}$ *can be obtained by elementary algebraic computation. Five of them are:*

$$\begin{array}{ll}
\phi_a(\text{in: } x_1, x_2\text{; out: } x_3) & \{x_3 := x_1 x_2\},\\
\phi_b(\text{in: } x_1, x_3\text{; out: } x_2) & \{x_2 := x_3/x_1 \textit{ if } x_1 \neq 0,\ \mathbb{R} \textit{ otherwise}\},\\
\phi_c(\text{in: } x_4\text{; out: } x_2) & \{x_2 := \sin(x_4)\},\\
\phi_d(\text{in: } x_1, x_3, x_4\text{; out: } x_2) & \{x_2 := \phi_b(x_1, x_3) \cap \phi_c(x_4)\},\\
\boldsymbol{\phi}_e(\text{in: } x_3, x_4\text{; out: } x_1, x_2) & \{x_2 := \sin(x_4),\\
 & \quad x_1 := x_3/x_2 \textit{ if } x_2 \neq 0,\ \mathbb{R} \textit{ otherwise}\}.
\end{array} \tag{4.10}$$

Note that ϕ_d may yield the empty set when x_1, x_3 and x_4 are inconsistent with $\mathcal{H}$. This is not necessarily the case as some inconsistent values for x_1, x_3 and x_4 are such that $\phi_d(x_1, x_3, x_4) \neq \varnothing$. For instance, $x_1 = x_4 = 0$ and $x_3 = 1$ are inconsistent (because $x_1 x_2 - x_3 \neq 0$), but $\phi_d(0, 1, 0) = \mathbb{R} \cap \{0\} = 0$. ■

Example 4.2 *Consider the CSP*

$$\mathcal{H} : \begin{pmatrix} \mathbf{Ap} - \mathbf{b} = \mathbf{0} \\ \mathbf{p} \in \mathbb{R}^{n_p}, \mathbf{b} \in [\mathbf{b}], \mathbf{A} \in [\mathbf{A}] \end{pmatrix}, \tag{4.11}$$

where $\mathbf{A}$ is a square matrix. The set of all variables of $\mathcal{H}$ is

$$\mathbf{x} = (a_{11}, \ldots, a_{n_p n_p}, p_1, \ldots, p_{n_p}, b_1, \ldots, b_{n_p})^T. \tag{4.12}$$

A possible subsolver is

$$\phi_f(\textit{in: } \mathbf{A}, \mathbf{p};\ \textit{out: } \mathbf{b})\ \ \{\mathbf{b} := \mathbf{Ap}\}, \tag{4.13}$$

where the input subvector consists of the coefficients of $\mathbf{A}$ and $\mathbf{p}$ and where the output subvector is $\mathbf{b}$. Another subsolver is

$$\phi_g(\textit{in: } \mathbf{A}, \mathbf{b};\ \textit{out: } \mathbf{p})\ \ \{\textit{code of a linear solver}\}. \tag{4.14}$$

The linear solver may, for instance, use Gauss elimination, which will be presented in the section. ■

4.2.2 Intervalization of finite subsolvers

In this section, we show how the knowledge of an interval counterpart of a finite subsolver of $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ makes it possible to contract $\mathcal{H}$. Let $\boldsymbol{\phi}$ be a finite subsolver of $\mathcal{H}$ with input vector $\mathbf{u}$ and output vector $\mathbf{v}$. An *inclusion function* $[\boldsymbol{\phi}]$ of $\boldsymbol{\phi}$, is a function $[\boldsymbol{\phi}]$ from $\mathbb{IR}^{n_\mathbf{u}}$ to $\mathbb{IR}^{n_\phi}$ such that for all boxes $[\mathbf{u}]$ of $\mathbb{IR}^{n_\mathbf{u}}$,

$$\boldsymbol{\phi}([\mathbf{u}]) \subset [\boldsymbol{\phi}]([\mathbf{u}]),\ \text{where } \boldsymbol{\phi}([\mathbf{u}]) \triangleq \bigcup_{\mathbf{u} \in [\mathbf{u}]} \boldsymbol{\phi}(\mathbf{u}). \tag{4.15}$$

This definition is a slight extension of that of Chapter 2, to take into account the fact that $\boldsymbol{\phi}(\mathbf{u})$ may not be a singleton. For instance, an intervalization of the subsolver $\boldsymbol{\phi}_e(x_3, x_4)$ of Example 4.1 is given by

$$
\begin{array}{l}
[\boldsymbol{\phi}_e]\,(\text{in: } [x_3]\,,[x_4]\,;\,\text{out: } [x_1]\,,[x_2])\\
\{\quad [x_2] := \sin([x_4]);\\
\qquad [x_1] := [x_3]\,/\,[x_2] \text{ if } 0 \notin [x_2]\,;\\
\qquad [x_1] := \mathbb{R} \text{ otherwise}\}.
\end{array}
\tag{4.16}
$$

Theorem 4.1 *Consider a CSP* $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ *and one of its finite subsolvers* $\boldsymbol{\phi}$ *with input vector* $\mathbf{u} = \mathbf{x}_{\mathcal{I}}$ *and output vector* $\mathbf{v} = \mathbf{x}_{\mathcal{J}}$*. If* $[\boldsymbol{\phi}]$ *is an inclusion function of* $\boldsymbol{\phi}$*, then a contraction of* $\mathcal{H}$ *can be performed by replacing each domain* $[x_j], j \in \mathcal{J}$ *by the domain* $[x_j] \cap \left[\phi_j\right]([\mathbf{u}])$*.* ■

Proof. Let $\mathbb{S}$ be the solution set of $\mathcal{H}$.

$$
\begin{array}{rl}
\mathbf{x} \in \mathbb{S} & \overset{(4.5)}{\Leftrightarrow} \quad \mathbf{x} \in [\mathbf{x}] \text{ and } \mathbf{f}(\mathbf{x}) = \mathbf{0}\\
& \overset{(4.8)}{\Leftrightarrow} \quad \mathbf{x} \in [\mathbf{x}] \text{ and } \mathbf{f}(\mathbf{x}) = \mathbf{0} \text{ and } \mathbf{v} \in \boldsymbol{\phi}(\mathbf{u})\\
& \overset{(4.15)}{\Leftrightarrow} \quad \mathbf{x} \in [\mathbf{x}] \text{ and } \mathbf{f}(\mathbf{x}) = \mathbf{0} \text{ and } [\mathbf{v}] \subset [\boldsymbol{\phi}]\,([\mathbf{u}]\,).
\end{array}
\tag{4.17}
$$

The replacement thus does not modify the solution set. ■

Remark 4.2 *The resulting contractor may of course leave the box* $[\mathbf{x}]$ *unchanged, in which case it will have failed to produce any useful result.* ■

Example 4.3 *Consider again the situation described in Example 4.1 with the subsolver* $\boldsymbol{\phi}_e$*. Theorem 4.1 and (4.16) yield*

$$[x_2] =]-\infty, 0] \cap \sin(\mathbb{R}) = [-1, 0],$$

$$[x_1] =]-\infty, 0] \cap \mathbb{R} =]-\infty, 0].$$

■

Intervalization of Gauss elimination. An important class of CSPs for which intervalization of finite subsolvers can be employed is that of *square linear systems of interval equations.* The problem is to compute a box containing the solution set of the CSP

$$\mathcal{H} : \begin{pmatrix} \mathbf{A} \in [\mathbf{A}]\,, \mathbf{b} \in [\mathbf{b}]\,, \mathbf{p} \in [\mathbf{p}] \\ \mathbf{A}\mathbf{p} - \mathbf{b} = \mathbf{0} \end{pmatrix}, \tag{4.18}$$

with

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n_{\mathrm{p}}} \\ \vdots & & & \vdots \\ a_{n_{\mathrm{p}}1} & a_{n_{\mathrm{p}}2} & \dots & a_{n_{\mathrm{p}}n_{\mathrm{p}}} \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_{n_{\mathrm{p}}} \end{pmatrix}. \tag{4.19}$$

The variables of $\mathcal{H}$ form the vector

$$\mathbf{x} = (a_{11}, \dots, a_{n_{\mathrm{p}}n_{\mathrm{p}}}, p_1, \dots, p_{n_{\mathrm{p}}}, b_1, \dots, b_{n_{\mathrm{p}}})^{\mathrm{T}}. \tag{4.20}$$

Remark 4.3 *Even if the expression* linear interval equations *is classical in the interval community, the relations between the variables are bilinear, since the entries of* $\mathbf{A}$ *and* $\mathbf{p}$ *belong to* $\mathbf{x}$. *We shall keep to the classical expression* linear interval equations *to be consistent with the literature, but shall talk of* linear CSPs *only when all the relations between the variables involved are linear. This is the case if the domain* $[\mathbf{A}]$ *is a punctual matrix, which implies that the vector of all variables boils down to* $\mathbf{x} = (p_1, \ldots, p_{n_p}, b_1, \ldots, b_{n_p})^T$ *and thus that the products of variables disappear.* ■

The classical Gauss elimination procedure can be used as a finite subsolver for (4.18). It makes it possible to compute $\mathbf{p}$ when the vector

$$\mathbf{u} = (a_{11}, a_{12}, \ldots, a_{n_\mathrm{p} n_\mathrm{p}}, b_1, \ldots, b_{n_\mathrm{p}})^\mathrm{T} \tag{4.21}$$

is known. A simple implementation of this procedure is given in Table 4.2, but more efficient implementations could be considered. An inclusion function for this finite subsolver is given in Table 4.3. For the intervalization of $\boldsymbol{\phi}$, the natural inclusion function was used. One may of course also use the centred form or any other type of inclusion function.

Table 4.2. Gauss elimination

Algorithm ϕ(in: $a_{11}, a_{12}, \ldots, a_{n_\mathrm{p} n_\mathrm{p}}, b_1, \ldots, b_{n_\mathrm{p}}$; out: $p_1, \ldots, p_{n_\mathrm{p}}$)	
1	for $i := 1$ to $n_\mathrm{p} - 1$
2	if $a_{ii} := 0$, then $\left(p_1, \ldots, p_{n_\mathrm{p}}\right) := \mathbb{R}^{n_\mathrm{p}}$; return;
3	for $j := i + 1$ to n_p
4	$\alpha_j := a_{ji}/a_{ii}$; $b_j := b_j - \alpha_j * b_i$;
5	for $k := i + 1$ to n_p
6	$a_{jk} := a_{jk} - \alpha_j * a_{ik}$;
7	for $i := n_\mathrm{p}$ down to 1
8	$p_i := \left(b_i - \sum_{j=i+1}^{n} a_{ij} * p_j\right) / a_{ii}$.

From Theorem 4.1, the operator

$$\mathcal{C}_{\mathrm{GE}}([\mathbf{A}], [\mathbf{b}], [\mathbf{p}]) \mapsto ([\mathbf{A}], [\mathbf{b}], [\mathbf{p}] \cap [\boldsymbol{\phi}]([\mathbf{A}], [\mathbf{p}], [\mathbf{b}])), \tag{4.22}$$

where $[\boldsymbol{\phi}](.)$ is given by Table 4.3 and GE stands for Gauss elimination, is a contractor for $\mathcal{H}$. Because the condition $0 \in [a_{ii}]$ is frequently satisfied for some i, $\mathcal{C}_{\mathrm{GE}}$ often fails to contract (4.18). $\mathcal{C}_{\mathrm{GE}}$ is efficient, for instance, when the interval matrix $[\mathbf{A}]$ is close to the identity matrix.

Note that the domain $[\mathbf{b}]$ for $\mathbf{b}$ can be contracted by using the subsolver

$$\boldsymbol{\phi}(\mathbf{A}, \mathbf{p}) = \mathbf{A}\mathbf{p}. \tag{4.23}$$

Table 4.3. Intervalization of Gauss elimination

Algorithm $[\phi]$ (in: $[a_{11}], [a_{12}], \ldots, \left[a_{n_\mathrm{p} n_\mathrm{p}}\right], [b_1], \ldots, \left[b_{n_\mathrm{p}}\right]$; out: $[p_1], \ldots, [p_{n_\mathrm{p}}]$)	
1	for $i := 1$ to $n_\mathrm{p} - 1$
2	if $0 \in [a_{ii}]$
3	$\left([p_1], \ldots, [p_{n_\mathrm{p}}]\right) := \mathbb{R}^{n_\mathrm{p}}$; return;
4	for $j := i + 1$ to n_p
5	$[\alpha_j] := [a_{ji}] / [a_{ii}]$; $[b_j] := [b_j] - [\alpha_j] * [b_i]$;
6	for $k := i + 1$ to n_p
7	$[a_{jk}] := [a_{jk}] - [\alpha_j] * [a_{ik}]$;
8	for $i := n_\mathrm{p}$ down to 1
9	$[p_i] := \left([b_i] - \sum_{j=i+1}^{n} [a_{ij}] * [p_j]\right) / [a_{ii}]$.

Example 4.4 *For*

$$[\mathbf{A}] = \begin{pmatrix} [4,5] & [-1,1] & [1.5,2.5] \\ [-0.5,0.5] & [-7,-5] & [1,2] \\ [-1.5,-0.5] & [-0.7,-0.5] & [2,3] \end{pmatrix},$$

$$[\mathbf{b}] = \begin{pmatrix} [3,4] \\ [0,2] \\ [3,4] \end{pmatrix} \text{ and } [\mathbf{p}] = \begin{pmatrix} [-\infty,\infty] \\ [-\infty,\infty] \\ [-\infty,\infty] \end{pmatrix}, \tag{4.24}$$

$\mathcal{C}_{\mathrm{GE}}([\mathbf{A}], [\mathbf{b}], [\mathbf{p}])$ *yields:*

$$[\mathbf{p}] = \begin{pmatrix} [-1.81928, 1.16873] \\ [-0.414071, 1.72523] \\ [0.700233, 3.42076] \end{pmatrix}. \tag{4.25}$$

This example is treated in Exercise 11.25, page 333. It can be checked that if $\mathcal{C}_{\mathrm{GE}}$ *is run again, the domain for* $\mathbf{p}$ *is not contracted any more.* $\mathcal{C}_{\mathrm{GE}}$ *is said to be* idempotent. *A contractor associated with the subsolver (4.23) would make it possible to contract* $[\mathbf{b}]$*, which might allow a new contraction of* $[\mathbf{p}]$ *by* $\mathcal{C}_{\mathrm{GE}}$. ■

4.2.3 Fixed-point methods

A *fixed-point subsolver* for the CSP $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ is an algorithm $\boldsymbol{\psi}$ such that

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \Leftrightarrow \mathbf{x} = \boldsymbol{\psi}(\mathbf{x}). \tag{4.26}$$

If the sequence

$$\mathbf{x}_{k+1} = \boldsymbol{\psi}(\mathbf{x}_k) \tag{4.27}$$

converges to a point $\mathbf{x}_\infty$ for a given initial value $\mathbf{x}_0$, then $\mathbf{x}_\infty$ is a solution of $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. For instance, since for any $a \neq 0$

$$x = a\left(x^2 - 2\right) + x \Leftrightarrow x^2 - 2 = 0, \tag{4.28}$$

a fixed-point subsolver for the CSP $\left(x^2 - 2 = 0, x \in \mathbb{R}\right)$ is

$$\psi(x) = a\left(x^2 - 2\right) + x. \tag{4.29}$$

A fixed-point subsolver provides an iterative procedure that may converge towards one of the solutions of $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. For instance, when $n_\mathrm{f} = n_\mathrm{x}$, a possible fixed-point subsolver for $\mathcal{H}$ is

$$\boldsymbol{\psi}(\mathbf{x}) = \mathbf{x} - \mathbf{M}\mathbf{f}(\mathbf{x}), \tag{4.30}$$

where $\mathbf{M}$ is any given invertible matrix, which may depend on $\mathbf{x}$.

Theorem 4.2 *Let $\boldsymbol{\psi} : \mathbb{R}^{n_x} \to \mathbb{R}^{n_x}$ be a fixed-point subsolver for $\mathcal{H}$, and $[\boldsymbol{\psi}] : \mathbb{IR}^{n_x} \to \mathbb{IR}^{n_x}$ be an inclusion function for $\boldsymbol{\psi}$. A contractor for $\mathcal{H}$ is obtained by replacing $[\mathbf{x}]$ in $\mathcal{H}$ by*

$$[\mathbf{x}] \cap [\boldsymbol{\psi}]([\mathbf{x}]). \tag{4.31}$$

This contractor will be called the fixed-point contractor *associated with $\boldsymbol{\psi}$.* ■

Proof. Let $\mathbb{S}$ be the solution set of $\mathcal{H}$. For any $\mathbf{x} \in \mathbb{S}$,

$$\begin{array}{lcl} \mathbf{f}(\mathbf{x}) = \mathbf{0} \text{ and } \mathbf{x} \in [\mathbf{x}] & \overset{(4.26)}{\Leftrightarrow} & \mathbf{x} \in [\mathbf{x}] \text{ and } \mathbf{x} = \boldsymbol{\psi}(\mathbf{x}) \\ & \Rightarrow & \mathbf{x} \in [\mathbf{x}] \text{ and } \mathbf{x} \in \boldsymbol{\psi}([\mathbf{x}]) \\ & \Rightarrow & \mathbf{x} \in [\mathbf{x}] \cap [\boldsymbol{\psi}]([\mathbf{x}]). \end{array}$$

Therefore, $\mathbb{S} \subset [\mathbf{x}] \cap [\boldsymbol{\psi}]([\mathbf{x}])$. ■

To illustrate the approach, three fixed-point contractors are now presented, namely the interval Gauss–Seidel, Krawczyk and interval Newton contractors.

Gauss–Seidel contractor. Consider again the CSP

$$\mathcal{H} : \begin{pmatrix} \mathbf{A} \in [\mathbf{A}], \mathbf{b} \in [\mathbf{b}], \mathbf{p} \in [\mathbf{p}] \\ \mathbf{A}\mathbf{p} - \mathbf{b} = \mathbf{0} \end{pmatrix}, \tag{4.32}$$

where the matrix $\mathbf{A}$ is assumed to be square. $\mathbf{A}$ can be decomposed as the sum of a diagonal matrix and a matrix with zeros on its diagonal:

$$\mathbf{A} = \mathrm{diag}\left(\mathbf{A}\right) + \mathrm{extdiag}\left(\mathbf{A}\right). \tag{4.33}$$

Now, $\mathbf{A}\mathbf{p} - \mathbf{b} = \mathbf{0}$ is equivalent to

$$\mathrm{diag}\left(\mathbf{A}\right)\mathbf{p} + \mathrm{extdiag}\left(\mathbf{A}\right)\mathbf{p} = \mathbf{b}. \tag{4.34}$$

Provided that $\mathrm{diag}(\mathbf{A})$ is invertible (*i.e.*, $\mathbf{A}$ has no zero entry on its diagonal), this equation can be rewritten as

$$\mathbf{p} = (\mathrm{diag}\,(\mathbf{A}))^{-1}\,(\mathbf{b} - \ \mathrm{extdiag}(\mathbf{A})\,\mathbf{p}). \tag{4.35}$$

A fixed-point subsolver for $\mathcal{H}$ is thus

$$\boldsymbol{\psi}\begin{pmatrix}\mathbf{A}\\ \mathbf{b}\\ \mathbf{p}\end{pmatrix} = \begin{pmatrix}\mathbf{A}\\ \mathbf{b}\\ (\mathrm{diag}\,(\mathbf{A}))^{-1}\,(\mathbf{b} - \ \mathrm{extdiag}(\mathbf{A})\,\mathbf{p})\end{pmatrix}. \tag{4.36}$$

An inclusion function for $\boldsymbol{\psi}$ is

$$[\boldsymbol{\psi}]\begin{pmatrix}[\mathbf{A}]\\ [\mathbf{b}]\\ [\mathbf{p}]\end{pmatrix} = \begin{pmatrix}[\mathbf{A}]\\ [\mathbf{b}]\\ (\mathrm{diag}\,([\mathbf{A}]))^{-1}\,([\mathbf{b}] - \mathrm{extdiag}([\mathbf{A}])\,[\mathbf{p}])\end{pmatrix}. \tag{4.37}$$

From Theorem 4.2, a contractor is given by

$$\mathcal{C}_{\mathrm{GS}} : \begin{pmatrix}[\mathbf{A}]\\ [\mathbf{b}]\\ [\mathbf{p}]\end{pmatrix} \mapsto \begin{pmatrix}[\mathbf{A}]\\ [\mathbf{b}]\\ [\mathbf{p}] \cap (\mathrm{diag}\,([\mathbf{A}]))^{-1}\,([\mathbf{b}] - \mathrm{extdiag}([\mathbf{A}])\,[\mathbf{p}])\end{pmatrix}. \tag{4.38}$$

$\mathcal{C}_{\mathrm{GS}}$ is the *Gauss–Seidel contractor*. As $\mathcal{C}_{\mathrm{GE}}$, it is efficient, for example, when $[\mathbf{A}]$ is close to the identity matrix.

Example 4.5 *Consider again the situation of Example 4.4, where*

$$[\mathbf{A}] = \begin{pmatrix}[4,5] & [-1,1] & [1.5,2.5]\\ [-0.5,0.5] & [-7,-5] & [1,2]\\ [-1.5,-0.5] & [-0.7,-0.5] & [2,3]\end{pmatrix}, \tag{4.39}$$

$$[\mathbf{b}] = \begin{pmatrix}[3,4]\\ [0,2]\\ [3,4]\end{pmatrix} \ \textit{and} \ [\mathbf{p}] = \begin{pmatrix}[-10,10]\\ [-10,10]\\ [-10,10]\end{pmatrix}. \tag{4.40}$$

Then

$$(\mathrm{diag}\,([\mathbf{A}]))^{-1} = \begin{pmatrix}[0.2,0.25] & [0,0] & [0,0]\\ [0,0] & [-0.2,-0.1429] & [0,0]\\ [0,0] & [0,0] & [0.3333,0.5]\end{pmatrix} \tag{4.41}$$

and

$$\text{extdiag}\,([\mathbf{A}]) = \begin{pmatrix} [0,0] & [-1,1] & [1.5,2.5] \\ [-0.5,0.5] & [0,0] & [1,2] \\ [-1.5,-0.5] & [-0.7,-0.5] & [0,0] \end{pmatrix}. \tag{4.42}$$

$\mathcal{C}_{\text{GS}} : ([\mathbf{A}], [\mathbf{b}], [\mathbf{p}])$ *yields*

$$[\mathbf{p}] = \begin{pmatrix} [-8, 9.75] \\ [-5.4001, 5.0001] \\ [-9.5, 10] \end{pmatrix}. \tag{4.43}$$

Table 4.4. Iterations of the Gauss–Seidel contractor

k	$[p_1]\,(k)$	$[p_2]\,(k)$	$[p_3]\,(k)$
0	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$
1	$[-8, 9.75]$	$[-5.40001, 5.00001]$	$[-9.5, 10]$
2	$[-6.85001, 8.28751]$	$[-5.17501, 4.97501]$	$[-6.39001, 10]$
5	$[-5.66909, 5.24052]$	$[-3.03602, 4.03031]$	$[-4.65079, 7.84124]$
10	$[-3.82831, 3.40254]$	$[-1.86306, 2.85556]$	$[-2.27608, 5.79364]$
20	$[-2.48786, 2.03998]$	$[-0.994123, 2.00077]$	$[-0.775045, 4.29151]$
100	$[-2.08452, 1.63673]$	$[-0.736983, 1.74357]$	$[-0.321329, 3.83781]$

Let us iterate contraction by $\mathcal{C}_{\text{GS}}$. *The results obtained for some values of* k *are given in Table 4.4, where* $[\mathbf{p}]\,(k)$ *is the box obtained for* $[\mathbf{p}]$ *at iteration* k. *For this example, the results obtained are less accurate than those obtained by* $\mathcal{C}_{\text{GE}}$ *in Example 4.4, page 72, but this is not always the case. This example is treated in Exercise 11.26, page 335.* ■

Krawczyk contractor. Consider the CSP $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$, where $n_{\text{f}} = n_{\text{x}}$ and $\mathbf{f}$ is assumed to be differentiable. Since, for any invertible matrix $\mathbf{M}$, $\mathbf{f}(\mathbf{x}) = \mathbf{0} \Leftrightarrow \mathbf{x} - \mathbf{M}\mathbf{f}(\mathbf{x}) = \mathbf{x}$, the function $\boldsymbol{\psi}(\mathbf{x}) = \mathbf{x} - \mathbf{M}\mathbf{f}(\mathbf{x})$ is a fixed-point subsolver for $\mathcal{H}$. The centred inclusion function for $\boldsymbol{\psi}$ is

$$[\boldsymbol{\psi}]([\mathbf{x}]) = \boldsymbol{\psi}(\mathbf{x}_0) + [\mathbf{J}_{\boldsymbol{\psi}}]([\mathbf{x}]) * ([\mathbf{x}] - \mathbf{x}_0), \tag{4.44}$$

where $[\mathbf{J}_{\boldsymbol{\psi}}]$ is an inclusion function for the Jacobian matrix of $\boldsymbol{\psi}$ and $\mathbf{x}_0 = \text{mid}([\mathbf{x}])$ (recall that $\text{mid}([\mathbf{x}])$ denotes the centre of $[\mathbf{x}]$). From Theorem 4.2, the following fixed-point contractor is obtained, classically called the *Krawczyk contractor* (Neumaier, 1990):

$$\mathcal{C}_{\text{K}} : [\mathbf{x}] \mapsto [\mathbf{x}] \cap (\boldsymbol{\psi}(\mathbf{x}_0) + [\mathbf{J}_{\boldsymbol{\psi}}]\,([\mathbf{x}]) * ([\mathbf{x}] - \mathbf{x}_0))\,. \tag{4.45}$$

Replace $\boldsymbol{\psi}(\mathbf{x})$ by $\mathbf{x} - \mathbf{M}\mathbf{f}(\mathbf{x})$ in (4.45) to get

$$\mathcal{C}_{\text{K}} : [\mathbf{x}] \mapsto [\mathbf{x}] \cap (\mathbf{x}_0 - \mathbf{M}\mathbf{f}\,(\mathbf{x}_0) + (\mathbf{I} - \mathbf{M}[\mathbf{J}_{\mathbf{f}}]([\mathbf{x}])) * ([\mathbf{x}] - \mathbf{x}_0))\,, \tag{4.46}$$

where $\mathbf{I}$ is the identity matrix and $[\mathbf{J_f}]$ is an inclusion function for the Jacobian matrix of $\mathbf{f}$. The matrix $\mathbf{M}$ is often taken to be the inverse $\mathbf{J_f}^{-1}(\mathbf{x}_0)$ of the Jacobian matrix of $\mathbf{f}$, computed at $\mathbf{x}_0$. It can be viewed as a preconditioning matrix (see Section 4.3.2, page 84). An algorithm implementing $\mathcal{C}_{\mathrm{K}}$ is shown in Table 4.5.

Table 4.5. Krawczyk contractor

Algorithm $\mathrm{C_K}$(in: $\mathbf{f}$; inout: $[\mathbf{x}]$)	
1	$\mathbf{x}_0 := \mathrm{mid}([\mathbf{x}])$;
2	$\mathbf{M} := \mathbf{J_f}^{-1}(\mathbf{x}_0)$;
3	$[\mathbf{J}_\psi] := \mathbf{I} - \mathbf{M}[\mathbf{J_f}]([\mathbf{x}])$;
4	$[\mathbf{r}] := \mathbf{x}_0 - \mathbf{M}\mathbf{f}(\mathbf{x}_0) + [\mathbf{J}_\psi] * ([\mathbf{x}] - \mathbf{x}_0)$;
5	$[\mathbf{x}] := [\mathbf{x}] \cap [\mathbf{r}]$.

Example 4.6 *Consider the CSP* $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$*, described by*

$$\mathcal{H} : \begin{pmatrix} f_1(x_1, x_2) = x_1^2 - 4x_2 \\ f_2(x_1, x_2) = x_2^2 - 2x_1 + 4x_2 \\ [\mathbf{x}] = [-0.1, 0.1] \times [-0.1, 0.3] \end{pmatrix}. \tag{4.47}$$

$\mathcal{H}$ *has a unique solution* $\mathbf{x} = (0, 0)^T$*. The Jacobian matrix for* $\mathbf{f}$ *is*

$$\mathbf{J_f} = \begin{pmatrix} 2x_1 & -4 \\ -2 & 2x_2 + 4 \end{pmatrix}, \tag{4.48}$$

and the preconditioning matrix $\mathbf{M}$ *is given by*

$$\mathbf{M} = \mathbf{J_f}^{-1}(0, 0.1) = \begin{pmatrix} -0.525 & -0.5 \\ -0.25 & 0 \end{pmatrix}. \tag{4.49}$$

The Krawczyk contractor yields:

$$\begin{aligned} \mathrm{C_K}([\mathbf{x}]) &= \begin{pmatrix} [-0.0555, 0.0455] \\ [-0.005, 0.005] \end{pmatrix}, \\ \mathrm{C_K}(\mathrm{C_K}([\mathbf{x}])) &= \begin{pmatrix} [-0.00258, 0.00255] \\ [-0.00128, 0.00127] \end{pmatrix}, \\ \mathrm{C_K}(\mathrm{C_K}(\mathrm{C_K}([\mathbf{x}]))) &= \begin{pmatrix} [-0.00000818, 0.00000817] \\ [-0.00000329, 0.00000329] \end{pmatrix}. \end{aligned} \tag{4.50}$$

which converges to the unique solution. This example is treated in Exercise 11.27, page 335. ■

Newton contractor. Consider again the CSP $\mathcal{H}$: $(\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$, where $n_\mathrm{f} = n_\mathrm{x}$ and the fixed-point subsolver given by $\boldsymbol{\psi}(\mathbf{x}) = \mathbf{x} - \mathbf{M}\mathbf{f}(\mathbf{x})$, see (4.30). If $\mathbf{f}(\mathbf{x})$ is affine, say $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$, then the fixed-point subsolver becomes $\boldsymbol{\psi}(\mathbf{x}) = \mathbf{x} - \mathbf{M}(\mathbf{A}\mathbf{x} + \mathbf{b})$. For $\mathbf{M} = \mathbf{A}^{-1}$, the sequence $\mathbf{x}_{k+1} = \boldsymbol{\psi}(\mathbf{x}_k)$ converges towards the solution $\mathbf{x}^* = -\mathbf{A}^{-1}\mathbf{b}$ in one step. Now, if $\mathbf{f}$ is non-linear but differentiable, it can be approximated by its first-order Taylor expansion to get an approximate fixed-point subsolver $\boldsymbol{\psi}(\mathbf{x}) = \mathbf{x} - \mathbf{J}_\mathbf{f}^{-1}(\mathbf{x}) * \mathbf{f}(\mathbf{x})$. The sequence (4.27) then leads to the Newton method. The inclusion function

$$[\boldsymbol{\psi}]([\mathbf{x}]) = [\mathbf{x}] - [\mathbf{J}_\mathbf{f}]^{-1}([\mathbf{x}]) * [\mathbf{f}]([\mathbf{x}]) \tag{4.51}$$

yields the Newton contractor (see Theorem 4.2)

$$\mathcal{C}_\mathrm{N}\colon\ [\mathbf{x}] \mapsto [\mathbf{x}] \cap \Big([\mathbf{x}] - [\mathbf{J}_\mathbf{f}]^{-1}([\mathbf{x}]) * [\mathbf{f}]([\mathbf{x}])\Big)\,. \tag{4.52}$$

Other inclusion functions, such as the centred form, could of course be used. Classically, a more efficient version of this contractor is used, as presented in Section 4.3.3, page 86.

4.2.4 Forward–backward propagation

The forward–backward contractor $\mathcal{C}_{\downarrow\uparrow}$ (Benhamou et al., 1999; Jaulin, 2000b) is based on constraint propagation (Waltz, 1975; Cleary, 1987; Davis, 1987). This contractor makes it possible to contract the domains of the CSP $\mathcal{H}$: $(\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ by taking into account any one of the n_f constraints in isolation, say $f_i(x_1, \ldots, x_{n_\mathrm{x}}) = 0$. Note that n_f is no longer necessarily equal to n_x. The next example illustrates how a given constraint can be used to contract domains.

Example 4.7 *Consider the constraint $x_3 = x_1 x_2$ and the box $[\mathbf{x}] = [1,4] \times [1,4] \times [8,40]$. This constraint can be rewritten in three ways:*

$$\begin{aligned} x_1 &= x_3/x_2,\\ x_2 &= x_3/x_1,\\ x_3 &= x_1 x_2. \end{aligned} \tag{4.53}$$

Each of these equations has been obtained by isolating one of the variables in the initial constraint. Three finite subsolvers are thus obtained:

$$\begin{aligned} &\phi_1(\textit{in: } x_2, x_3\textit{; out: } x_1)\ \{x_1 := x_3/x_2 \textit{ if } x_2 \neq 0,\ \mathbb{R} \textit{ otherwise}\},\\ &\phi_2(\textit{in: } x_1, x_3\textit{; out: } x_2)\ \{x_2 := x_3/x_1 \textit{ if } x_1 \neq 0,\ \mathbb{R} \textit{ otherwise}\},\\ &\phi_3(\textit{in: } x_1, x_2\textit{; out: } x_3)\ \{x_3 := x_1 x_2\}. \end{aligned} \tag{4.54}$$

An intervalization of these finite subsolvers leads to the following contractions

$$\begin{array}{lcl} ([x_3] \,/\, [x_2]) \cap [x_1] & = & \dfrac{[8,40]}{[1,4]} \cap [1,4] = [2,4], \\ ([x_3] \,/\, [x_1]) \cap [x_2] & = & [2,4], \\ ([x_1] * [x_2]) \cap [x_3] & = & ([1,4] * [1,4]) \cap [8,40] = [8,16]. \end{array} \tag{4.55}$$

The new domain is thus $[2,4] \times [2,4] \times [8,16]$. ■

Assume that each constraint has the form $f_i(x_1, \ldots, x_{n_x}) = 0$, where f_i can be decomposed into a sequence of operations involving elementary operators and functions such as $+, -, *, /$, sin, cos... It is then possible to decompose this constraint into *primitive constraints* (Lhomme, 1993). Roughly speaking, a primitive constraint is a constraint involving a single operator (such as $+, -, *$ or $/$) or a single function (such as sin, cos or exp). For instance the constraint $x_1 \exp(x_2) + \sin(x_3) = 0$ can be decomposed into the following set of primitive constraints

$$\begin{cases} a_1 & = \exp(x_2), \\ a_2 & = x_1 a_1, \\ a_3 & = \sin(x_3), \\ a_2 + a_3 & = 0. \end{cases} \tag{4.56}$$

The domains associated with all intermediate variables (here a_1, a_2 and a_3) are $]-\infty, \infty[$. A method for contracting $\mathcal{H}$ with respect to the constraint $x_1 \exp(x_2) + \sin(x_3) = 0$ is to contract each of the primitive constraints in (4.56) until the contractors become inefficient. This is the principle of *constraint propagation* (Waltz, 1975), initially employed without the help of interval analysis.

Forward–backward propagation selects the primitive constraints to be used for contractions in an optimal order in the sense of the size of the domains finally obtained (Benhamou et al., 1999). This is illustrated by the following example.

Example 4.8 *Consider the equation*

$$f(\mathbf{x}) = 0 \tag{4.57}$$

where

$$f(\mathbf{x}) = x_1 \exp(x_2) + \sin(x_3). \tag{4.58}$$

The domains for the variables x_1, x_2 *and* x_3 *are denoted by* $[x_1], [x_2]$ *and* $[x_3]$. *To obtain an algorithm contracting these domains, first write an algorithm that computes* $y = f(\mathbf{x})$, *by a finite sequence of elementary operations, such as the one suggested by (4.56)*

$$\begin{array}{l} a_1 := \exp(x_2); \\ a_2 := x_1 a_1; \\ a_3 := \sin(x_3); \\ y \;\; := a_2 + a_3. \end{array} \tag{4.59}$$

Then write an interval counterpart to this algorithm:

$$\begin{array}{ll} 1 & [a_1] := \exp([x_2]); \\ 2 & [a_2] := [x_1] * [a_1]; \\ 3 & [a_3] := \sin([x_3]); \\ 4 & [y] := [a_2] + [a_3]. \end{array} \tag{4.60}$$

Since $f(\mathbf{x}) = 0$, the domain for y should be taken equal to the singleton $\{0\}$. One can thus add the step

$$5 \quad [y] := [y] \cap \{0\}. \tag{4.61}$$

If $[y]$ as computed at Step 5 turns out to be empty, then we know that the CSP has no solution. Else, $[y]$ is replaced by $\{0\}$. Finally, a backward propagation is performed, updating the domains associated with all the variables to get

$$\begin{array}{ll} 6 & [a_2] := ([y] - [a_3]) \cap [a_2]; \ \textit{// see Step 4} \\ 7 & [a_3] := ([y] - [a_2]) \cap [a_3]; \ \textit{// see Step 4} \\ 8 & [x_3] := \sin^{-1}([a_3]) \cap [x_3]; \ \textit{// see Step 3} \\ 9 & [a_1] := ([a_2]/[x_1]) \cap [a_1]; \ \textit{// see Step 2} \\ 10 & [x_1] := ([a_2]/[a_1]) \cap [x_1]; \ \textit{// see Step 2} \\ 11 & [x_2] := \log([a_1]) \cap [x_2]. \quad \textit{// see Step 1} \end{array} \tag{4.62}$$

At Step 8, $\sin^{-1}([a_3]) \cap [x_3]$ returns the smallest interval containing $\{x_3 \in [x_3] \mid \sin(x_3) \in [a_3]\}$. The associated contractor is given in Table 4.6. ■

Table 4.6. Forward–backward contractor

Algorithm $\mathcal{C}_{\downarrow\uparrow}$(inout: $[\mathbf{x}]$)	
1	$[a_1] := \exp([x_2])$;
2	$[a_2] := [x_1] * [a_1]$;
3	$[a_3] := \sin([x_3])$;
4	$[y] := [a_2] + [a_3]$;
5	$[y] := [y] \cap \{0\}$;
6	$[a_2] := ([y] - [a_3]) \cap [a_2]$;
7	$[a_3] := ([y] - [a_2]) \cap [a_3]$;
8	$[x_3] := \sin^{-1}([a_3]) \cap [x_3]$;
9	$[a_1] := ([a_2]/[x_1]) \cap [a_1]$;
10	$[x_1] := ([a_2]/[a_1]) \cap [x_1]$;
11	$[x_2] := \log([a_1]) \cap [x_2]$.

An example of the design of a forward–backward contractor for a function defined by an algorithm containing loops will be presented in Section 6.4.3, page 171, in the context of state estimation. The next example illustrates the performances of $\mathcal{C}_{\downarrow\uparrow}$ on a linear CSP. See also Exercise 11.11, page 318.

Example 4.9 *Consider the CSP*

$$\mathcal{H}: \left(\begin{array}{l} x_1 + 2x_2 - x_3 = 0 \\ x_1 - x_2 - x_4 = 0 \\ [\mathbf{x}] \in [-10, 10] \times [-10, 10] \times [-1, 1] \times [-1, 1] \end{array} \right). \tag{4.63}$$

Using $\mathcal{C}_{\downarrow\uparrow}$, the constraint $x_1 + 2x_2 - x_3 = 0$ yields

$$[x_1] = [-10, 10], \tag{4.64}$$

$$[x_2] = [-\tfrac{11}{2}, \tfrac{11}{2}], \tag{4.65}$$

$$[x_3] = [-1, 1], \tag{4.66}$$

and the constraint $x_1 - x_2 - x_4 = 0$ yields

$$[x_1] = [-\tfrac{13}{2}, \tfrac{13}{2}], \tag{4.67}$$

$$[x_2] = [-\tfrac{11}{2}, \tfrac{11}{2}], \tag{4.68}$$

$$[x_4] = [-1, 1]. \tag{4.69}$$

Iterating this procedure, one would like the sequence of boxes $[\mathbf{x}](k)$ to converge towards the smallest possible domain. Unfortunately, this is not so. The results of the contractions for some values of the iteration counter k are presented in Table 4.7.

Table 4.7. Iterations of the forward–backward contractor

k	$[x_1](k)$	$[x_2](k)$	$[x_3](k)$	$[x_4](k)$
0	$[-10, 10]$	$[-10, 10]$	$[-1, 1]$	$[-1, 1]$
1	$[-6.5, 6.5]$	$[-5.5, 5.5]$	$[-1, 1]$	$[-1, 1]$
2	$[-4.75, 4.75]$	$[-3.75, 3.75]$	$[-1, 1]$	$[-1, 1]$
5	$[-3.21875, 3.21875]$	$[-2.21875, 2.21875]$	$[-1, 1]$	$[-1, 1]$
10	$[-3.00684, 3.00684]$	$[-2.00684, 2.00684]$	$[-1, 1]$	$[-1, 1]$
∞	$[-3, 3]$	$[-2, 2]$	$[-1, 1]$	$[-1, 1]$

$\mathcal{C}_{\downarrow\uparrow}$ comes to a deadlock. A possible way out is to bisect the search box into two boxes:

$$\begin{cases} [\mathbf{x}](1) = [-10, 0] \times [-10, 10] \times [-1, 1] \times [-1, 1], \\ [\mathbf{x}](2) = [0, 10] \times [-10, 10] \times [-1, 1] \times [-1, 1], \end{cases} \tag{4.70}$$

and to apply $\mathcal{C}_{\downarrow\uparrow}$ to each of these boxes in turn. For $[\mathbf{x}](1)$ *one gets*

k	$[x_1](k)$	$[x_2](k)$	$[x_3](k)$	$[x_4](k)$
0	$[-10,0]$	$[-10,10]$	$[-1,1]$	$[-1,1]$
1	$[-1.5,0]$	$[-0.5,1]$	$[-1,1]$	$[-1,1]$
∞	$[-1.5,0]$	$[-0.5,1]$	$[-1,1]$	$[-1,1]$

(4.71)

and for $[\mathbf{x}](2)$

k	$[x_1](k)$	$[x_2](k)$	$[x_3](k)$	$[x_4](k)$
0	$[0,10]$	$[-10,10]$	$[-1,1]$	$[-1,1]$
1	$[0,1.5]$	$[-1,0.5]$	$[-1,1]$	$[-1,1]$
∞	$[0,1.5]$	$[-1,0.5]$	$[-1,1]$	$[-1,1]$

(4.72)

The reunification of the domains obtained for $[\mathbf{x}](1)$ *and* $[\mathbf{x}](2)$ *provides the domain*

$$[\mathbf{x}] = [-1.5,1.5] \times [-1,1] \times [-1,1] \times [-1,1], \tag{4.73}$$

which corresponds to the optimal contraction of $\mathcal{H}$ (Jaulin, 2000b), i.e., $[\mathbf{x}]$ *is the interval hull of the solution set.* ■

Remark 4.4 *On the CSP (4.32), $\mathcal{C}_{\downarrow\uparrow}$ applied for each equation of the system* $\mathbf{Ap} = \mathbf{b}$ *leads to the same contraction for* $[\mathbf{p}]$ *as* $\mathcal{C}_{\rm GS}$*, but it has the advantage over* $\mathcal{C}_{\rm GS}$ *of also providing contractions for* $[\mathbf{A}]$ *and* $[\mathbf{b}]$. ■

4.2.5 Linear programming approach

Consider again the problem of contracting the domains of the CSP

$$\mathcal{H} : \begin{pmatrix} \mathbf{Ap} - \mathbf{b} = \mathbf{0} \\ \mathbf{A} \in [\mathbf{A}], \mathbf{b} \in [\mathbf{b}], \mathbf{p} \in [\mathbf{p}] \end{pmatrix}, \tag{4.74}$$

where now $\mathbf{A}$ is not necessarily square. Finding the smallest box $[\mathbf{p}]$ that contains all the vectors $\mathbf{p}$ consistent with $\mathcal{H}$ is known to be NP-hard in the general case (Rohn, 1994). We shall consider two special cases for which this problem can be transformed into $2n_{\rm p}$ linear programming problems, so that efficient linear programming techniques can be used. The resulting contractor will be denoted by $\mathcal{C}_{\rm LP}$.

Case 1: *All components of* $\mathbf{p}$ *are assumed to be positive* ($\mathbf{p} > \mathbf{0}$). The vector $\mathbf{p} \in [\mathbf{p}]$ is then consistent with $\mathcal{H}$ if and only if

$$\begin{aligned} & \exists \mathbf{A} \in [\mathbf{A}], \exists \mathbf{b} \in [\mathbf{b}] \mid \mathbf{Ap} - \mathbf{b} = \mathbf{0} \\ \Leftrightarrow\ & \exists \mathbf{A} \in [\mathbf{A}] \mid \mathbf{Ap} \in [\mathbf{b}] \\ \Leftrightarrow\ & \exists \mathbf{A} \in [\mathbf{A}] \mid \mathbf{Ap} \geqslant \underline{\mathbf{b}} \text{ and } \mathbf{Ap} \leqslant \overline{\mathbf{b}}. \end{aligned} \tag{4.75}$$

Now, since $\mathbf{p} > \mathbf{0}$, the following equivalences hold true

$$(\exists \mathbf{A} \in [\mathbf{A}] \mid \mathbf{Ap} \geqslant \underline{\mathbf{b}}) \Leftrightarrow \overline{\mathbf{A}}\mathbf{p} \geqslant \underline{\mathbf{b}}, \tag{4.76}$$

$$\left(\exists \mathbf{A} \in [\mathbf{A}] \mid \mathbf{Ap} \leqslant \overline{\mathbf{b}}\right) \Leftrightarrow \underline{\mathbf{A}}\mathbf{p} \leqslant \overline{\mathbf{b}}. \tag{4.77}$$

Remark 4.5 *If at least one component of* $\mathbf{p}$ *is not positive, the last two equivalences are no longer true. For instance, assume that* $\mathbf{A}$ *is scalar and equal to the real number* a*, and take (4.76) with* $[a] = [-4, -1]$*,* $\underline{b} = 4$ *and* $p = -2$*. The proposition* $(\exists a \in [-4, -1] \mid ap \geqslant \underline{b})$ *is true (take* $a = -3$*), whereas the proposition* $(\overline{a}p \geqslant \underline{b})$ *is false (*$-1 * -2 \geqslant 4$ *is false).* ■

Therefore, $\mathbf{p} \in [\mathbf{p}]$, where $[p_i] \subset \mathbb{R}^+$ $(i = 1, \ldots, n_{\mathrm{p}})$, is consistent with $\mathcal{H}$ if and only if

$$\overline{\mathbf{A}}\mathbf{p} \geqslant \underline{\mathbf{b}} \text{ and } \underline{\mathbf{A}}\mathbf{p} \leqslant \overline{\mathbf{b}}. \tag{4.78}$$

The smallest box $[\mathbf{q}]$ containing all the vectors $\mathbf{p}$ that are consistent with $\mathcal{H}$ can therefore be computed by solving the following $2n_{\mathrm{p}}$ linear programming problems:

$$\begin{cases} \text{opt } p_i,\ i = 1, \ldots, n_{\mathrm{p}}, \\ \begin{pmatrix} -\overline{\mathbf{A}} \\ \underline{\mathbf{A}} \end{pmatrix} \mathbf{p} \leqslant \begin{pmatrix} -\underline{\mathbf{b}} \\ \overline{\mathbf{b}} \end{pmatrix}, \\ \mathbf{p} \in [\mathbf{p}], \end{cases} \tag{4.79}$$

where opt is alternatively min and max to obtain the coordinates of $\underline{\mathbf{q}}$ and $\overline{\mathbf{q}}$.

Case 2: *The domain* $[\mathbf{A}]$ *is assumed to be punctual* (*i.e.,* $\underline{\mathbf{A}} = \overline{\mathbf{A}}$). The vector $\mathbf{p} \in [\mathbf{p}]$ is consistent with $\mathcal{H}$ if and only if there exists $\mathbf{b} \in [\mathbf{b}]$ such that $\mathbf{Ap} - \mathbf{b} = \mathbf{0}$, *i.e.*,

$$\exists \mathbf{b} \in [\mathbf{b}] \mid \mathbf{Ap} - \mathbf{b} = \mathbf{0} \Leftrightarrow \mathbf{Ap} \in [\mathbf{b}] \Leftrightarrow \mathbf{Ap} \geqslant \underline{\mathbf{b}} \text{ and } \mathbf{Ap} \leqslant \overline{\mathbf{b}}. \tag{4.80}$$

The smallest box $[\mathbf{q}]$ containing all the vectors $\mathbf{p}$ that are consistent with $\mathcal{H}$ can therefore be computed by solving the following $2n_{\mathrm{p}}$ linear programming problems:

$$\begin{cases} \text{opt } p_i,\ i = 1, \ldots, n_{\mathrm{p}}, \\ \begin{pmatrix} -\mathbf{A} \\ \mathbf{A} \end{pmatrix} \mathbf{p} \leqslant \begin{pmatrix} -\underline{\mathbf{b}} \\ \overline{\mathbf{b}} \end{pmatrix}, \\ \mathbf{p} \in [\mathbf{p}], \end{cases} \tag{4.81}$$

where opt is again alternatively min and max.

4.3 External Approximation

Contractors based on the basic contractors of Section 4.2 and able to contract a much larger class of CSPs will now be presented.

4.3.1 Principle

A subvector $\mathbf{p}$ of $\mathbf{x}$ is *consistent* with $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ if $\mathbf{p}$ can be supplemented with another subvector to form a solution vector $\mathbf{x}$ (*i.e.*, $\mathbf{x} \in \mathbb{S}$). Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be two CSPs and $\mathbf{x}$ be the vector of all variables of $\mathcal{H}_1$.

We shall write $\mathcal{H}_1 \Rightarrow \mathcal{H}_2$, if

- all the variables of $\mathcal{H}_1$ are also variables of $\mathcal{H}_2$ and
- if $\mathbf{x}$ is a solution vector of $\mathcal{H}_1$ then it is consistent with $\mathcal{H}_2$.

The notation $\mathcal{H}_1 \Rightarrow \mathcal{H}_2$ indicates that the CSP $\mathcal{H}_2$ can be deduced from the CSP $\mathcal{H}_1$ (for instance by introducing additional variables taking the values of some expressions in constraints of $\mathcal{H}_1$). We shall say that $\mathcal{H}_2$ is an *external approximation* of $\mathcal{H}_1$, because the solution set of $\mathcal{H}_2$ is guaranteed to contain that of $\mathcal{H}_1$.

Example 4.10 *Consider the three CSPs*

$$\mathcal{H}_1 : \begin{pmatrix} 3x_1 - \exp(x_1) = 0 \\ x_1 \in [0, 2] \end{pmatrix},$$

$$\mathcal{H}_2 : \begin{pmatrix} 3x_1 - x_2 = 0 \\ x_2 - \exp(x_1) = 0 \\ x_1 \in [0, 2],\ x_2 \in [-\infty, +\infty] \end{pmatrix}, \tag{4.82}$$

$$\mathcal{H}_3 : \begin{pmatrix} 3x_1 - \exp(1) - \exp(\xi)(x_1 - 1) = 0 \\ x_1 \in [0, 2],\ \xi \in [0, 2] \end{pmatrix}.$$

Check that $\mathcal{H}_1 \Rightarrow \mathcal{H}_2$ and $\mathcal{H}_1 \Rightarrow \mathcal{H}_3$. The fact that $\mathcal{H}_1 \Rightarrow \mathcal{H}_3$ is a direct consequence of the mean-value theorem, which implies that for any $x_1 \in [0, 2]$, there exists $\xi \in [0, 2]$ such that $\exp(x_1) = \exp(1) + \exp(\xi)(x_1 - 1)$. ■

Contracting the domains of a CSP $\mathcal{H}$ via an external approximation consists of three steps:

- finding an external approximation $\mathcal{H}_1$ of $\mathcal{H}$ for which basic subsolvers are efficient,
- contracting $\mathcal{H}_1$
- updating the domains for $\mathcal{H}$.

The approach will first be illustrated through two classical methods, namely the preconditioning of linear interval systems in Section 4.3.2 and the interval Newton iteration (or Newton contractor) in Section 4.3.3, before presenting some alternative approaches in Sections 4.3.4 and 4.3.5.

4.3.2 Preconditioning

Consider the CSP $\mathcal{H} : (\mathbf{Ap} - \mathbf{b} = \mathbf{0}, \mathbf{A} \in [\mathbf{A}], \mathbf{p} \in [\mathbf{p}], \mathbf{b} \in [\mathbf{b}])$, and assume that $[\mathbf{A}]$ is such that the Gauss–Seidel contractor presented in Section 4.2.3, page 73, and the Gauss elimination contractor presented in Section 4.2.2, page 70, are not efficient. If $\mathbf{A}_0$ is some invertible matrix in $[\mathbf{A}]$, then

$$\begin{pmatrix} \mathbf{Ap} - \mathbf{b} = \mathbf{0} \\ \mathbf{p} \in [\mathbf{p}],\ \mathbf{A} \in [\mathbf{A}],\ \mathbf{b} \in [\mathbf{b}] \end{pmatrix} \Rightarrow \begin{pmatrix} (i) & \mathbf{A}' = \mathbf{A}_0^{-1}\mathbf{A} \\ (ii) & \mathbf{b}' = \mathbf{A}_0^{-1}\mathbf{b} \\ (iii) & \mathbf{A}'\mathbf{p} - \mathbf{b}' = \mathbf{0} \\ (iv) & \mathbf{p} \in [\mathbf{p}], \mathbf{A} \in [\mathbf{A}],\ \mathbf{b} \in [\mathbf{b}] \end{pmatrix}. \tag{4.83}$$

A basic contraction over the constraints (i) and (ii) provides domains $[\mathbf{A}']$ and $[\mathbf{b}']$ for $\mathbf{A}'$ and $\mathbf{b}'$. Provided that $w([\mathbf{A}])$ is small enough and that $\mathbf{A}_0$ is well conditioned, the interval matrix $[\mathbf{A}']$ is close to the identity matrix. Contractors such as $\mathcal{C}_{\rm GS}$ or $\mathcal{C}_{\rm GE}$ are then likely to be efficient on the constraint (iii). This is illustrated by the contractor $\mathcal{C}_{\rm GSP}$ (GSP for Gauss–Seidel with preconditioning) of Table 4.8.

Table 4.8. Gauss–Seidel contractor with preconditioning

Algorithm $\mathcal{C}_{\rm GSP}$(inout: $[\mathbf{A}], [\mathbf{p}], [\mathbf{b}]$)	
1	$\mathbf{A}_0 := \text{mid}([\mathbf{A}])$;
2	$[\mathbf{A}'] := \mathbf{A}_0^{-1}\,[\mathbf{A}]$;
3	$[\mathbf{b}'] := \mathbf{A}_0^{-1}\,[\mathbf{b}]$;
4	$\mathcal{C}_{\rm GS}(\mathbf{A}'\mathbf{p} - \mathbf{b}' = \mathbf{0}, [\mathbf{A}'], [\mathbf{p}], [\mathbf{b}'])$;
5	$[\mathbf{b}] := \mathbf{A}_0\,[\mathbf{b}'] \cap [\mathbf{b}]$;
6	$[\mathbf{A}] := \mathbf{A}_0[\mathbf{A}'] \cap [\mathbf{A}]$.

Remark 4.6 *If $\mathcal{C}_{\rm GE}$ was used at Step 4 instead of $\mathcal{C}_{\rm GS}$, the contractor of Table 4.8 would become $\mathcal{C}_{\rm GEP}$ (*GEP *for Gauss elimination with preconditioning).* ■

Example 4.11 *Consider again the situation of Example 4.4, page 72, where*

$$[\mathbf{A}] = \begin{pmatrix} [4,5] & [-1,1] & [1.5,2.5] \\ [-0.5,0.5] & [-7,-5] & [1,2] \\ [-1.5,-0.5] & [-0.7,-0.5] & [2,3] \end{pmatrix} \text{ and } [\mathbf{b}] = \begin{pmatrix} [3,4] \\ [0,2] \\ [3,4] \end{pmatrix}, \tag{4.84}$$

but assume that

$$[\mathbf{p}] = \begin{pmatrix} [-10, 10] \\ [-10, 10] \\ [-10, 10] \end{pmatrix}. \tag{4.85}$$

At Step 2, $[\mathbf{A}'] = (\mathrm{mid}\,([\mathbf{A}]))^{-1} * [\mathbf{A}]$ *is given by*

$$\begin{pmatrix} [0.81909, 1.18092] & [-0.21869, 0.21869] & [-0.18092, 0.18092] \\ [-0.14248, 0.14248] & [0.79556, 1.20445] & [-0.14248, 0.14248] \\ [-0.23659, 0.23659] & [-0.15110, 0.15110] & [0.76342, 1.23659] \end{pmatrix}, \tag{4.86}$$

and at Step 3,

$$[\mathbf{b}'] = (\mathrm{mid}\,([\mathbf{A}]))^{-1} * [\mathbf{b}] = \begin{pmatrix} [-0.0755468, 0.302187] \\ [-0.0231942, 0.437376] \\ [1.24056, 1.74951] \end{pmatrix}. \tag{4.87}$$

$\mathcal{C}_{\mathrm{GS}}(\mathbf{A}'\mathbf{p} - \mathbf{b}' = \mathbf{0}, [\mathbf{A}'], [\mathbf{p}], [\mathbf{b}'])$ *finally yields*

$$[\mathbf{p}] = \begin{pmatrix} [-4.97088, 5.24758] \\ [-3.611, 4.13162] \\ [-3.4532, 7.3698] \end{pmatrix}. \tag{4.88}$$

Table 4.9. Iterations of Gauss–Seidel contractor with preconditioning

k	$[p_1]\,(k)$	$[p_2]\,(k)$	$[p_3]\,(k)$
0	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$
1	$[-4.97088, 5.24758]$	$[-3.61101, 4.13162]$	$[-3.45313, 7.36981]$
2	$[-2.82313, 3.09983]$	$[-2.28883, 2.80945]$	$[-0.818916, 4.73559]$
5	$[-1.26437, 1.54107]$	$[-0.939061, 1.45968]$	$[0.474056, 3.14881]$
10	$[-1.10986, 1.38656]$	$[-0.813738, 1.33436]$	$[0.573876, 2.98711]$
20	$[-1.10698, 1.38368]$	$[-0.811386, 1.33201]$	$[0.575741, 2.98409]$

The results of Table 4.9 are obtained by iterating the application of $\mathcal{C}_{\mathrm{GSP}}$. *They are better than those obtained without preconditioning in Example 4.5, page 74. This example is treated in Exercise 11.28, page 336.* ■

Example 4.12 *If now* $[\mathbf{p}]$ *is given, as in Example 4.4, by*

$$[\mathbf{p}] = \begin{pmatrix} [-\infty, \infty] \\ [-\infty, \infty] \\ [-\infty, \infty] \end{pmatrix}, \tag{4.89}$$

$\mathcal{C}_{\rm GSP}$ cannot contract $[\mathbf{p}]$. *On the other hand, by replacing $\mathcal{C}_{\rm GS}$ by $\mathcal{C}_{\rm GE}$ in Step 4 of Table 4.8, one obtains the contractor $\mathcal{C}_{\rm GEP}$, which contracts* $[\mathbf{p}]$ *in one iteration into*

$$[\mathbf{p}] = \begin{pmatrix} [-1.10698, 1.38367] \\ [-0.785241, 1.33201] \\ [0.758351, 2.98409] \end{pmatrix}. \tag{4.90}$$

This result is slightly better than that obtained by $\mathcal{C}_{\rm GS}$ in Example 4.4. No further contraction can be performed on $[\mathbf{p}]$ *by iterating $\mathcal{C}_{\rm GE}$. The results obtained by $\mathcal{C}_{\rm GEP}$ in one interation for this example are better than those obtained by $\mathcal{C}_{\rm GSP}$ in* 20 *iterations (see Table 4.9). On the other hand, for many examples, $\mathcal{C}_{\rm GEP}$ is unable to contract* $[\mathbf{p}]$ *whereas $\mathcal{C}_{\rm GSP}$ is efficient.* ■

4.3.3 Newton contractor

This contractor applies to the CSP

$$\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}]), \tag{4.91}$$

with $n_{\rm f} = n_{\rm x}$ (Moore, 1979; Hansen, 1992a). It is not a direct intervalization of the Newton fixed-point method, but can be interpreted as an improved version of the contractor presented in Section 4.2.3. It is very efficient when $\mathbf{f}$ is smooth over the domains of the CSP and when these domains are small. Let $\mathbf{x}_0$ be any vector in $[\mathbf{x}]$ (its centre, for instance), then a componentwise application of the mean-value theorem to (4.91) yields, with obvious notation,

$$\begin{pmatrix} \mathbf{f}(\mathbf{x}_0) + \mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi}_1 \dots \boldsymbol{\xi}_{n_{\rm f}})(\mathbf{x} - \mathbf{x}_0) = \mathbf{0} \\ \mathbf{x} \in [\mathbf{x}], \boldsymbol{\xi}_1 \in [\mathbf{x}] \dots \boldsymbol{\xi}_{n_{\rm f}} \in [\mathbf{x}] \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} \mathbf{A}\mathbf{p} + \mathbf{f}(\mathbf{x}_0) = \mathbf{0} \\ \mathbf{p} = \mathbf{x} - \mathbf{x}_0 \\ \mathbf{A} = \mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi}_1 \dots \boldsymbol{\xi}_{n_{\rm f}}) \\ \mathbf{x} \in [\mathbf{x}], \boldsymbol{\xi}_1 \in [\mathbf{x}] \dots \boldsymbol{\xi}_{n_{\rm f}} \in [\mathbf{x}] \end{pmatrix} \tag{4.92}$$

This yields the contractor of Table 4.10, denoted by $\mathcal{C}_{\rm N}$ (for Newton contractor).

Remark 4.7 *The order in which the constraints are written, which was arbitrary in (4.92), is significant in the implementation of $\mathcal{C}_{\rm N}$.* ■

Remark 4.8 *From a geometrical point of view, the principle of this contractor is to enclose the graph of each coordinate function $f_i(\mathbf{x})$ over* $[\mathbf{x}]$ *between two hyperplanes and to solve the associated interval linear system. This can be viewed as a linearization where non-linearity is transformed into uncertainty.* ■

Table 4.10. Newton contractor

Algorithm $\mathcal{C}_{\mathrm{N}}$ (in: $\mathbf{f}$; inout: $[\mathbf{x}]$)	
1	$\mathbf{x}_0 := \mathrm{mid}([\mathbf{x}])$;
2	$[\mathbf{A}] := [\mathbf{J}_{\mathbf{f}}]([\mathbf{x}])$;
3	$[\mathbf{p}] := [\mathbf{x}] - \mathbf{x}_0$;
4	$\mathcal{C}_{\mathrm{GSP}}$ $(\mathbf{A}\mathbf{p} + \mathbf{f}(\mathbf{x}_0) = \mathbf{0}, \mathbf{A} \in [\mathbf{A}], \mathbf{p} \in [\mathbf{p}])$;
5	$[\mathbf{x}] := [\mathbf{x}] \cap ([\mathbf{p}] + \mathbf{x}_0)$.

4.3.4 Parallel linearization

In Section 4.3.3, the number n_{f} of constraints was assumed to be equal to the dimension n_{x} of $\mathbf{x}$. Assume now that n_{f} and n_{x} may differ. The contractor to be presented handles all the constraints simultaneously, using a *parallel linearization* approach (Kolev, 1998; Jaulin, 2001b). The principle is to enclose each $f_i(\mathbf{x})$ over $[\mathbf{x}]$ between two parallel hyperplanes (contrary to the interval Newton method, where these hyperplanes are not parallel). The function $\mathbf{f}$ is then bracketed over $[\mathbf{x}]$ according to

$$\mathbf{A}\mathbf{x} + \underline{\mathbf{b}} \leqslant \mathbf{f}(\mathbf{x}) \leqslant \mathbf{A}\mathbf{x} + \overline{\mathbf{b}}. \tag{4.93}$$

This bracketing can again be found by using the mean-value theorem, provided that $\mathbf{f}$ is differentiable. Assume that $\mathbf{x}_0$ is a point in $[\mathbf{x}]$, for instance its centre. Then there exists $\boldsymbol{\xi} \in [\mathbf{x}]$ such that

$$\begin{aligned} & \mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi})(\mathbf{x} - \mathbf{x}_0) \\ \Leftrightarrow\ & \mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi})(\mathbf{x} - \mathbf{x}_0) - \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) \\ \Leftrightarrow\ & \mathbf{f}(\mathbf{x}) = \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0)\mathbf{x} + \mathbf{f}(\mathbf{x}_0) - \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0)\mathbf{x}_0 + (\mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi}) - \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0))(\mathbf{x} - \mathbf{x}_0). \end{aligned} \tag{4.94}$$

Equivalently, one may write $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$, with

$$\mathbf{A} = \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0) \tag{4.95}$$

and

$$\mathbf{b} = \mathbf{f}(\mathbf{x}_0) - \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0)\mathbf{x}_0 + (\mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi}) - \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0))(\mathbf{x} - \mathbf{x}_0). \tag{4.96}$$

Based on (4.94), an external approximation of $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ is thus

$$\begin{pmatrix} \mathbf{A}\mathbf{x} + \mathbf{b} = \mathbf{0} \\ \mathbf{A} = \mathbf{J}_{\mathbf{f}}(\mathbf{x}_0) \\ \mathbf{b} = \mathbf{f}(\mathbf{x}_0) - \mathbf{A}\mathbf{x}_0 + (\mathbf{J}_{\mathbf{f}}(\boldsymbol{\xi}) - \mathbf{A})(\mathbf{x} - \mathbf{x}_0) \\ \mathbf{x} \in [\mathbf{x}], \boldsymbol{\xi} \in [\mathbf{x}] \end{pmatrix}. \tag{4.97}$$

The resulting contractor by parallel linearization is denoted by $\mathcal{C}_{\|}$. An implementation of $\mathcal{C}_{\|}$ is given in Table 4.11. The contractor $\mathcal{C}_{\mathrm{LP}}$ called at Step 4 by

Table 4.11. Contractor by parallel linearization

Algorithm $\mathcal{C}_{\|}$ (in: $\mathbf{f}$; inout: $[\mathbf{p}]$)	
1	$\mathbf{x}_0 := \text{mid}([\mathbf{x}])$;
2	$\mathbf{A} := \mathbf{J_f}(\mathbf{x}_0)$;
3	$[\mathbf{b}] := \mathbf{f}(\mathbf{x}_0) - \mathbf{A}\mathbf{x}_0 + ([\mathbf{J_f}]([\mathbf{x}]) - \mathbf{A})([\mathbf{x}] - \mathbf{x}_0)$;
4	$\mathcal{C}_{\text{LP}}(\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}, \mathbf{x} \in [\mathbf{x}], \mathbf{b} \in [\mathbf{b}])$.

$\mathcal{C}_{\|}$ performs a contraction of the domain for $\mathbf{x}$ via linear programming (see Section 4.2.5, Case 2).

To quantify the quality of the linear bracketing (4.93), let us compute the ratio $w([\mathbf{b}])/w([\mathbf{x}])$, where $[\mathbf{b}]$ is the box computed at Step 3 of $\mathcal{C}_{\|}$. This ratio is given by

$$\frac{w([\mathbf{b}])}{w([\mathbf{x}])} = \frac{w(([\mathbf{J_f}]([\mathbf{x}]) - \mathbf{J_f}(\mathbf{x}_0)) * ([\mathbf{x}] - \mathbf{x}_0))}{w([\mathbf{x}])}, \tag{4.98}$$

so $w([\mathbf{b}])/w([\mathbf{x}])$ tends to 0 with $w([\mathbf{x}])$, which means that the bracketing becomes more and more accurate when $[\mathbf{x}]$ converges to a point. Figure 4.2 illustrates the parallel linearization $ax + [b] = \frac{1}{4}x + [\frac{1}{3}, 1]$ of a function f over $[x] = [-2, 2]$.

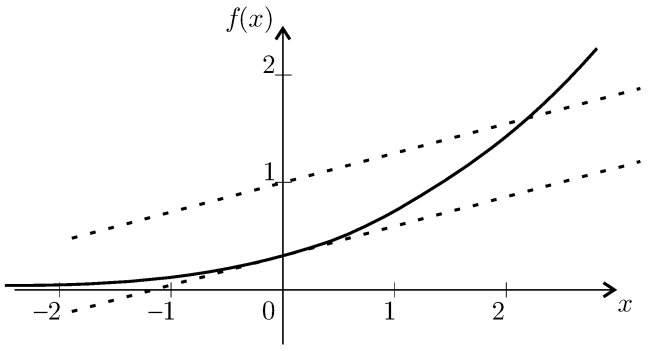


Fig. 4.2. Parallel linearization in the scalar case

4.3.5 Using formal transformations

Algebraic manipulation of the existing constraints makes it possible to build a CSP $\mathcal{H}_1$ that is an external approximation of $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$, *i.e.*, such that $\mathcal{H} \Rightarrow \mathcal{H}_1$. Except in the polynomial case (Marti and Rueher, 1995; Benhamou and Granvilliers, 1997), no general formal method seems to have been developed to build such external approximations, but it is good

practice to design $\mathcal{H}_1$ in such a way that it involves few variables to facilitate processing by bisection if it turns out to be needed. The idea is illustrated in the following example.

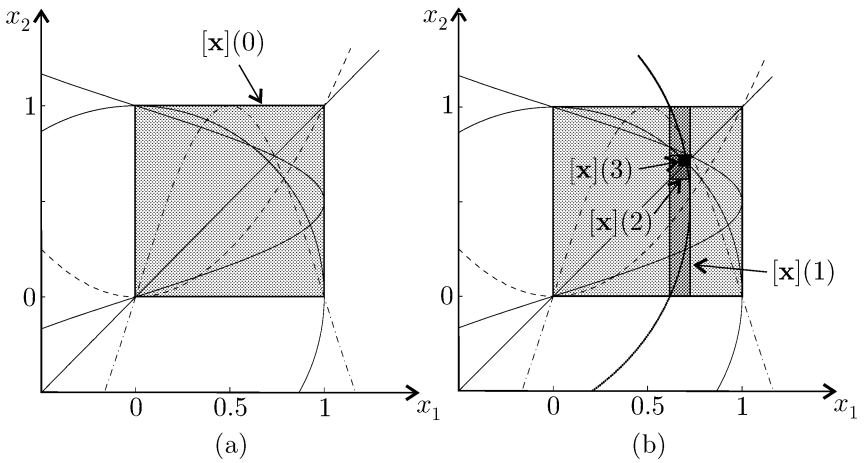


Fig. 4.3. The CSP of Example 4.13 has an empty solution set; (a) contractors such as $\mathcal{C}_{\|}$, $\mathcal{C}_{\mathrm{N}}$ or $\mathcal{C}_{\downarrow\uparrow}$ fail to contract $[\mathbf{x}](0)$; (b) a formal transformation makes it possible to bypass the deadlock

Example 4.13 *Consider the CSP*

$$\mathcal{H}: \left(\begin{array}{ll} x_1 - x_2 & = 0 \\ x_1^2 + x_2^2 - 1 & = 0 \\ x_2 - \sin(\pi x_1) & = 0 \\ x_1 - \sin(\pi x_2) & = 0 \\ x_2 - x_1^2 & = 0 \\ x_1 \in [0,1], x_2 \in [0,1] & \end{array} \right). \tag{4.99}$$

Contractors based on linearization, such as $\mathcal{C}_{\|}$ or $\mathcal{C}_{\mathrm{N}}$ are not efficient on $\mathcal{H}$ because the domains are too large. $\mathcal{C}_{\downarrow\uparrow}$ is not efficient either, because each of the five constraints is consistent with the box $[\mathbf{x}](0) = [0,1] \times [0,1]$ (see Figure 4.3a). Now, by summing the first two constraints, one gets the new constraint $x_1^2 + x_2^2 - 1 + x_1 - x_2 = 0$. Therefore, the CSP

$$\mathcal{H}_1: \left(\begin{array}{l} \left(x_1 + \frac{1}{2}\right)^2 + \left(x_2 - \frac{1}{2}\right)^2 - \frac{3}{2} = 0 \\ x_1 \in [0,1], x_2 \in [0,1] \end{array} \right). \tag{4.100}$$

satisfies $\mathcal{H} \Rightarrow \mathcal{H}_1$. Constraint propagation can be used to contract $[x_1]$ and $[x_2]$ in $\mathcal{H}_1$. The primitive constraints associated with the constraint of $\mathcal{H}_1$ are

$$\begin{cases} a_1 &= x_1 + \frac{1}{2}, \\ a_2 &= x_2 - \frac{1}{2}, \\ a_3 &= a_1^2, \\ a_4 &= a_2^2, \\ a_3 + a_4 &= \frac{3}{2}. \end{cases}$$

They are used to contract the domains for x_1, x_2, a_1, a_2, a_3 and a_4 as detailled below

$$\begin{array}{ll}
a_1 = x_1 + \frac{1}{2} \text{ and } x_1 \in [0,1] & \Rightarrow a_1 \in \left[\frac{1}{2}, \frac{3}{2}\right], \\
a_2 = x_2 - \frac{1}{2} \text{ and } x_2 \in [0,1] & \Rightarrow a_2 \in \left[-\frac{1}{2}, \frac{1}{2}\right], \\
a_3 = a_1^2 \text{ and } a_1 \in \left[\frac{1}{2}, \frac{3}{2}\right] & \Rightarrow a_3 \in \left[\frac{1}{4}, \frac{9}{4}\right], \\
a_4 = a_2^2 \text{ and } a_2 \in \left[-\frac{1}{2}, \frac{1}{2}\right] & \Rightarrow a_4 \in \left[0, \frac{1}{4}\right], \\
a_3 = \frac{3}{2} - a_4,\ a_4 \in \left[0, \frac{1}{4}\right] \text{ and } a_3 \in [\frac{1}{4}, \frac{9}{4}] & \Rightarrow a_3 \in \left[\frac{5}{4}, \frac{3}{2}\right], \\
a_4 = \frac{3}{2} - a_3,\ a_3 \in [\frac{5}{4}, \frac{3}{2}] \text{ and } a_4 \in [0, \frac{1}{4}] & \Rightarrow a_4 \in \left[0, \frac{1}{4}\right], \\
a_2^2 = a_4,\ a_2 \in \left[-\frac{1}{2}, \frac{1}{2}\right] \text{ and } a_4 \in \left[0, \frac{1}{4}\right] & \Rightarrow a_2 \in \left[-\frac{1}{2}, \frac{1}{2}\right], \\
a_1^2 = a_3,\ a_1 \in \left[\frac{1}{2}, \frac{3}{2}\right] \text{ and } a_3 \in \left[\frac{5}{4}, \frac{3}{2}\right] & \Rightarrow a_1 \in \left[\sqrt{\frac{5}{4}}, \sqrt{\frac{3}{2}}\right], \\
x_2 = a_2 + \frac{1}{2} \text{ and } a_2 \in \left[-\frac{1}{2}, \frac{1}{2}\right] & \Rightarrow x_2 \in [0,1], \\
x_1 = a_1 - \frac{1}{2} \text{ and } a_1 \in \left[\sqrt{\frac{5}{4}}, \sqrt{\frac{3}{2}}\right] & \Rightarrow x_1 \in \left[\sqrt{\frac{5}{4}} - \frac{1}{2}, \sqrt{\frac{3}{2}} - \frac{1}{2}\right].
\end{array}$$

No more contraction can be performed. The resulting contracted box is

$$[\mathbf{x}](1) = \left[\sqrt{\tfrac{5}{4}} - \tfrac{1}{2}, \sqrt{\tfrac{3}{2}} - \tfrac{1}{2}\right] \times [0,1]$$

The contraction is optimal, as illustrated by the box $[\mathbf{x}](1)$, in dark grey in Figure 4.3b. This is because x_1 and x_2 occur only once in the constraint of $\mathcal{H}_1$. The contracted domain $[\mathbf{x}](1)$ can then be transmitted back to $\mathcal{H}$ to bypass the deadlock. The first constraint of $\mathcal{H}$ makes it possible to contract $[\mathbf{x}](1)$ down to $[\mathbf{x}](2)$ and the second constraint to contract $[\mathbf{x}](2)$ down to $[\mathbf{x}](3)$, see Figure 4.3b. Few more iterations are necessary to get empty intervals. ■

4.4 Collaboration Between Contractors

4.4.1 Principle

Several contractors for CSPs of the form $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ have been presented. None of them can claim to be universally better than the others. What is important is their complementarity. A good contractor is one that is able to contract the CSP, especially when the others are not. Among the contractors presented up to now, $\mathcal{C}_{\text{GE}}$, $\mathcal{C}_{\text{N}}$, $\mathcal{C}_{\text{GSP}}$ and $\mathcal{C}_{\|}$ are efficient only if

the size of $[\mathbf{x}]$ is small, whereas $\mathcal{C}_{\downarrow\uparrow}$ is more efficient if $[\mathbf{x}]$ is large. $\mathcal{C}_{\rm GE}$, $\mathcal{C}_{\rm GSP}$ and $\mathcal{C}_{\rm LP}$ consider constraints only of the form $\mathbf{Ap} = \mathbf{b}$, whereas $\mathcal{C}_{\|}$, $\mathcal{C}_{\downarrow\uparrow}$ and $\mathcal{C}_{\rm N}$ can deal with much more general non-linear constraints.

In this section, the available contractors will be made to collaborate in order to build a more efficient contractor, inheriting the qualities of its constituents. Any new contractor with desirable features can be appended to the resulting contractor.

The basic idea is taken from *interval constraint propagation* (ICP), an extension to intervals of *constraint propagation* as initially developed by Waltz (1975), which is akin to a relaxation method. This extension was independently proposed by Cleary (1987) and Davis (1987). The idea is to contract the domains of the variables of the CSP by using all the available contractors successively. The order in which the contractors are selected forms the *strategy* (Montanari and Rossi, 1991).

Let us first recall some definitions (Benhamou and Granvilliers, 1997). A contractor $\mathcal{C}$ satisfies

$$\begin{array}{ll} \forall [\mathbf{x}],\ \mathcal{C}([\mathbf{x}]) \subset [\mathbf{x}] & (\textit{contractance}), \\ \forall [\mathbf{x}],\ [\mathbf{x}] \cap \mathbb{S} \subset \mathcal{C}([\mathbf{x}]) & (\textit{correctness}), \end{array} \tag{4.101}$$

where $\mathbb{S}$ is the solution set of $\mathcal{H}$. Moreover, a contractor is *monotonic* if

$$[\mathbf{x}] \subset [\mathbf{y}] \Rightarrow \mathcal{C}([\mathbf{x}]) \subset \mathcal{C}([\mathbf{y}]). \tag{4.102}$$

All the contractors presented in this chapter are monotonic (Granvilliers, 1998). A contractor $\mathcal{C}$ is *idempotent* if

$$\mathcal{C} \circ \mathcal{C}([\mathbf{x}]) \triangleq \mathcal{C}(\mathcal{C}([\mathbf{x}])) = \mathcal{C}([\mathbf{x}]). \tag{4.103}$$

A *fixed point* of a contractor $\mathcal{C}$ is a box $[\mathbf{x}]$ that satisfies

$$\mathcal{C}([\mathbf{x}]) = [\mathbf{x}]. \tag{4.104}$$

If $\mathcal{C}_1$ and $\mathcal{C}_2$ are monotonic, then the contractor $\mathcal{C}_{1,2} \triangleq \mathcal{C}_1 \circ \mathcal{C}_2([\mathbf{x}]))$ is also monotonic. Even if $\mathcal{C}_1$ and $\mathcal{C}_2$ are idempotent, $\mathcal{C}_{1,2}$ may not be idempotent. A *store* is a set of contractors and a strategy is a sequence of contractors belonging to the store. Consider, for instance, a store $\mathcal{L}$ consisting of four contractors $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ and $\mathcal{C}_4$:

$$\mathcal{L} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}.$$

A *cyclic* strategy associated with $\mathcal{L}$ corresponds to the sequence

$$\mathcal{S} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \ldots\}. \tag{4.105}$$

A strategy $\mathcal{S}$ is *fair* if, for any $k \geqslant 1$ and any contractor $\mathcal{C}$ in the store, there exists $k_1 \geqslant k$ such that $\mathcal{C}$ is at rank k_1.

The contractor $\mathcal{C}_\infty$ of Table 4.12 corresponds to the application of all the contractors of the list $\mathcal{L}$ according to the strategy $\mathcal{S}$. For a store of monotonic contractors and a fair strategy, it is trivial to prove that $\mathcal{C}_\infty$ is monotonic and idempotent.

Table 4.12. Contractor combining all the contractors of the store $\mathcal{L}$ according to a strategy $\mathcal{S}$

Algorithm $\mathcal{C}_\infty$ (inout: $[\mathbf{x}]$)	
1	$k = 0; [\mathbf{x}](0) = [\mathbf{x}];$
2	repeat
3	$k := k + 1;$
4	choose the contractor $\mathcal{C}$ in $\mathcal{L}$ according to the strategy $\mathcal{S}$;
5	$[\mathbf{x}](k) := \mathcal{C}([\mathbf{x}](k-1));$
6	until $[\mathbf{x}](k)$ is a fixed point of all contractors in $\mathcal{L}$;
7	$[\mathbf{x}] := [\mathbf{x}](k).$

Theorem 4.3 *The algorithm $\mathcal{C}_\infty([\mathbf{x}])$ converges to the largest box $[\mathbf{z}]$ included in $[\mathbf{x}]$ such that $\forall \mathcal{C}$ in $\mathcal{L}$, $\mathcal{C}([\mathbf{z}]) = [\mathbf{z}]$, provided that all sets (or constraints) involved are closed.* ■

Proof. Denote by $[\mathbf{x}](k)$ the box at iteration k. Let us first prove that $\mathcal{C}_\infty([\mathbf{x}])$ contains $[\mathbf{z}]$. Since $[\mathbf{z}] \subset [\mathbf{x}](0)$ and since all contractors of the store are monotonic,

$$[\mathbf{z}] \subset [\mathbf{x}](k) \Rightarrow \mathcal{C}([\mathbf{z}]) \subset \mathcal{C}([\mathbf{x}](k)) \Rightarrow [\mathbf{z}] \subset [\mathbf{x}](k+1), \tag{4.106}$$

for any contractor $\mathcal{C}$ of $\mathcal{L}$, then $[\mathbf{z}] \subset [\mathbf{x}](\infty) = \mathcal{C}_\infty([\mathbf{x}])$. Let us now prove that $\mathcal{C}_\infty([\mathbf{x}]) = [\mathbf{z}]$. For all $\mathcal{C}$ in $\mathcal{L}$, since the strategy is fair, $\mathcal{C}(\mathcal{C}_\infty([\mathbf{x}])) = \mathcal{C}_\infty([\mathbf{x}])$, *i.e.*, $\mathcal{C}_\infty([\mathbf{x}])$ is a fixed point of all contractors in $\mathcal{L}$. Therefore $[\mathbf{z}] = \mathcal{C}_\infty([\mathbf{x}])$. ■

This result has been established by Montanari and Rossi (1991) for CSPs with finite domains, see also Arsouze et al. (2000). Here, it has been extended to continuous domains. It shows that the result obtained by $\mathcal{C}_\infty$ is independent of the strategy used, provided that it is fair. In our implementation, a cyclic strategy has been chosen because it is simple to implement as it does not require any bookkeeping on dynamical structures involving the use of pointers. More efficient strategies can be found in Montanari and Rossi (1991). In practice, the loop in the algorithm of Table 4.12 is stopped when the contractions are deemed too small.

The algorithm of Table 4.12 will now be illustrated on an example where all constraints have been chosen linear to facilitate calculation by hand. For simplicity, the store contains only contractors based on forward–backward propagation. There are only two variables to allow a visual presentation of the contractions and to illustrate the deadlock effect.

Example 4.14 *Consider the two following CSPs, both to be contracted with respect to their two equality constraints with $\mathcal{C}_{\downarrow\uparrow}$.*

$$\mathcal{H}_1 : \begin{pmatrix} x_1 + x_2 \ = 0 \\ x_1 - 2x_2 = 0 \\ x_1 \in [-10, 10] \\ x_2 \in [-10, 10] \end{pmatrix} \text{ and } \mathcal{H}_2 : \begin{pmatrix} x_1 + x_2 = 0 \\ x_1 - x_2 = 0 \\ x_1 \in [-10, 10] \\ x_2 \in [-10, 10] \end{pmatrix}. \tag{4.107}$$

The store contains two contractors, namely $\mathcal{C}_{\downarrow\uparrow}$ applied to the first equation (contractor $\mathcal{C}_1$) and $\mathcal{C}_{\downarrow\uparrow}$ applied to the second equation (contractor $\mathcal{C}_2$). A cyclic strategy is chosen. For $\mathcal{H}_1$, $\mathcal{C}_\infty$ converges to the solution $\mathbf{x} = \mathbf{0}$ (Figure 4.4a), but $\mathcal{C}_\infty$ fails to contract $\mathcal{H}_2$. The reason for this failure is that the box $[-10, 10] \times [-10, 10]$ is consistent with each constraint taken independently. For a visual interpretation of this deadlock, consider the sets associated with the two constraints of $\mathcal{H}_2$:

$$\begin{aligned} \mathbb{E}_1 &= \{(x_1, x_2) \mid x_1 + x_2 = 0\}, \\ \mathbb{E}_2 &= \{(x_1, x_2) \mid x_1 - x_2 = 0\}. \end{aligned} \tag{4.108}$$

*$\mathcal{C}_1([\mathbf{x}])$ returns the smallest box that contains $\mathbb{E}_1 \cap [\mathbf{x}]$. But since $\mathbb{E}_1$ intersects all faces of $[\mathbf{x}]$ (see Figure 4.4b), this smallest box is $[\mathbf{x}]$ itself (*i.e., *$\mathcal{C}_1([\mathbf{x}]) = [\mathbf{x}]$). The situation is similar with $\mathbb{E}_2$,* i.e., *$\mathcal{C}_2([\mathbf{x}]) = [\mathbf{x}]$. As illustrated by Figure 4.4a, the deadlock cannot occur with $\mathcal{H}_1$. Note that the contractors $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_\infty$ are all idempotent for $\mathcal{H}_1$ and $\mathcal{H}_2$. The composition $\mathcal{C}_1 \circ \mathcal{C}_2$ is idempotent for $\mathcal{H}_2$ but not for $\mathcal{H}_1$.* ■

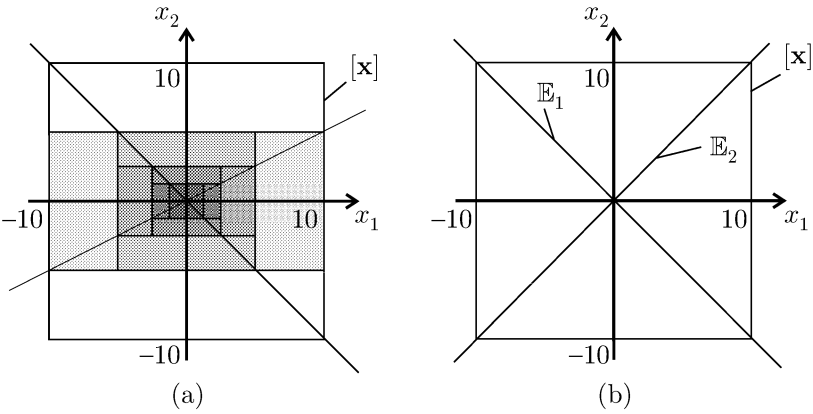


Fig. 4.4. Interpretation of forward–backward propagation; (a) $\mathcal{C}_\infty$ contracts $[\mathbf{x}]$ down to the singleton $\mathbf{0}$; (b) the contractor is at a deadlock, *i.e.*, $\mathcal{C}_1([\mathbf{x}]) = [\mathbf{x}]$ and $\mathcal{C}_2([\mathbf{x}]) = [\mathbf{x}]$

Remark 4.9 *Example 4.14 suggests three conjectures that are true if $n_x = 2$ but false if $n_x \geqslant 3$. The first one is that if $\mathbb{E}_1$ or $\mathbb{E}_2$ in Figure 4.4b is*

moved, the deadlock disappears, and thus that the failure of $\mathcal{C}_{\downarrow\uparrow}$ is atypical. The second conjecture is that if $\mathcal{C}_{\downarrow\uparrow}$ fails in the linear case, then the centre of $[\mathbf{x}]$ is a solution of $\mathcal{H}$ (see Figure 4.4b). The last conjecture is that if the constraints are monotonous with respect to all the variables with the same type of monotonicity, then $\mathcal{C}_{\downarrow\uparrow}$ cannot fail (see Figure 4.5). Figure 4.6 provides a counterexample to these three conjectures, by showing a situation with two linear constraints with the same type of monotonicity, with an empty solution set and for which $\mathcal{C}_{\downarrow\uparrow}$ fails. The two constraints are represented by the two parallel planes. Both planes touch all faces of $[\mathbf{x}]$ and thus $\mathcal{C}_{\downarrow\uparrow}$ cannot contract $[\mathbf{x}]$. ■

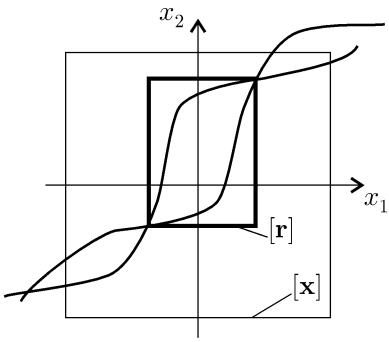


Fig. 4.5. With $\mathcal{C}_{\downarrow\uparrow}$, the box $[\mathbf{x}]$ will converge to the box $[\mathbf{r}]$, which is the smallest one consistent with the two constraints

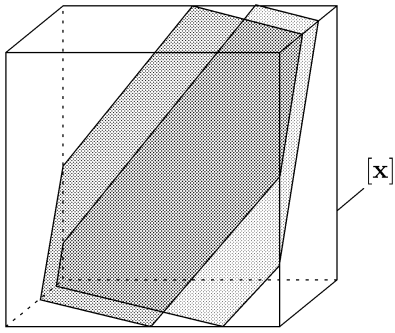


Fig. 4.6. Counterexample to three conjectures of Remark 4.9

4.4.2 Contractors and inclusion functions

Contractors may employ inclusion functions. These inclusion functions are not necessarily minimal, so contractors may be used to improve their accuracy. The complexity of the resulting contractors remains polynomial. This idea is the foundation of *box-consistency*, developed in the context of constraint propagation (Benhamou et al., 1999). Using contractors to improve inclusion functions is especially helpful when dealing with high-dimensional problems (typically more than ten variables), and when there are multiple occurrences of variables in the formal expression of the function f.

An upper bound $\overline{y}$ for

$$\overline{f} = \max_{\mathbf{x}\in[\mathbf{x}]} f(\mathbf{x}), \tag{4.109}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, can be computed by the algorithm UUB (for upper upper bound) of Table 4.13. UUB uses a contractor $\mathcal{C}$ for contracting the CSP

$$\mathcal{H} : \left(f(\mathbf{x}) = y,\ \mathbf{x} \in [\mathbf{x}],\ y \in [\underline{y}, \overline{y}]\right), \tag{4.110}$$

where $[\underline{y}, \overline{y}]$ is an interval guaranteed to contain $\overline{f}$, initially equal to $[f]([\mathbf{x}])$. The complexity of the computation can be kept polynomial provided that $\mathcal{C}$ has polynomial complexity.

Table 4.13. Computing an upper bound for an inclusion function with a contractor

Algorithm UUB(in: f, $[\mathbf{x}]$; out: $\overline{y}$)	
1	repeat
2	$[\underline{y}, \overline{y}] := [f]([\mathbf{x}])$; // where $[f]$ is a classical inclusion function
3	$\underline{y} := f(\text{mid}([\mathbf{x}]))$;
4	$([\mathbf{x}], [y]) := \mathcal{C}\left(f(\mathbf{x}) = y,\ \mathbf{x} \in [\mathbf{x}],\ y \in [\underline{y}, \overline{y}]\right)$;
5	until the improvement on $\underline{y}$ and $\overline{y}$ is deemed too small.

The interval computed at Step 2 of UUB contains $\overline{f}$, as defined by (4.109). At Step 3, $\underline{y}$ is necessarily a lower bound for $\overline{f}$, but this statement could be made more efficient by using a local maximization algorithm, such as a punctual Newton method. The contractor of Step 4 eliminates parts of $[\mathbf{x}]$ that contain $\mathbf{x}$ such that $f(\mathbf{x}) < \underline{y}$.

Example 4.15 *Let us search for an upper bound of the function* $f(x) = x^2 - \frac{3}{2}x$ *over the interval* $[x] = [0, 4]$. *With the natural inclusion function, at Step 2,* UUB *yields*

$$[f]([x]) = [x]^2 - \frac{3}{2}[x] = [0, 16] - \frac{3}{2}[0, 4] = [-6, 16]. \tag{4.111}$$

The pessimism due to the two occurrences of x *in the formal expression of* f *can be illustrated by decomposing the constraint* $f(x) = x^2 - \frac{3}{2}x$ *into the*

primitive constraints $a = x^2$ *and* $y = a - \frac{3}{2}x$. *In the* (x, a) *space (*i.e., *in the* $\left(x, x^2\right)$ *space), the relation* $y = -\frac{3}{2}x + a$ *corresponds to the orthogonal projection of the point* (x, a) *onto a straight line with direction vector* $\mathbf{v} = (-\frac{3}{2}, 1)^T$. *As illustrated by Figure 4.7a,* $[f]([x]) = [-6, 16]$ *is much larger than* $f([x]) = [-0.5625, 12]$, *because it includes the projections of the pairs* $(x = 4, x^2 = 0)$ *and* $(x = 0, x^2 = 16)$, *which are unfeasible. The role of* UUB *is to enclose the unknown upper bound* $\overline{f} = 12$ *for* $f([x])$ *inside a smaller interval. At Step 3,* UUB *computes* $\underline{y} = f(\text{mid}([x])) = f(2) = 4 - 3 = 1$. *At this stage, the best available enclosure for* $\overline{f}$ *is* $[1, 16]$. *At Step 4,* UUB *calls a contractor for the CSP*

$$(x^2 - \frac{3}{2}x = y, x \in [0, 4], y \in [1, 16]). \tag{4.112}$$

Assume for simplicity that the only available contractor is $\mathcal{C}_{\downarrow\uparrow}$. *Contracting the domains of the CSP (4.112) corresponds to the following operations*

$$\begin{aligned}
&[x] = [0, 4]\,,\\
&[a] = [x]^2 = [0, 16],\\
&[y] = \left([a] - \tfrac{3}{2}\,[x]\right) \cap [1, 16] = [1, 16],\\
&[a] = \left([y] + \tfrac{3}{2}\,[x]\right) \cap [a] = ([1, 16] + [0, 6]) \cap [0, 16] = [1, 16],\\
&[x] = \sqrt{[a]} \cap [0, 4] = [1, 4].
\end{aligned} \tag{4.113}$$

This is summarized by Figure 4.7b. The box $[1, 4] \times [1, 16]$ *containing the pair* $\left(x, x^2\right)$ *is represented in grey. During the second iteration of the loop, a better lower bound for* $\overline{f}$ *is obtained as* $\underline{y} = f\left(\text{mid}\left([x]\right)\right) = f(2.5) = 2.5$. *The best known enclosure for* $\overline{f}$ *is now* $[2.5, 16]$. *Figures 4.7c and 4.7d describe the results of the second and third iterations of the loop. The procedure converges to the actual value* $\overline{f} = 12$. ■

Table 4.14. Inclusion function evaluation with a contractor

Algorithm IFEC(in: f, $[\mathbf{x}]$; out: $[\underline{y}, \overline{y}]$)	
1	$\underline{y} := -\text{UUB}(-f, [\mathbf{x}])$;
2	$\overline{y} := \text{UUB}(f, [\mathbf{x}])$.

The same algorithm can also be used to compute a lower bound $\underline{y}$ for the lower bound of $f\left([\mathbf{x}]\right)$. It suffices to compute $-\text{UUB}(-f, [\mathbf{x}])$. An inclusion function $[f]$ for f can thus be obtained by running the algorithm IFEC presented in Table 4.14. Employing this algorithm for the evaluation of inclusion functions used by a contractor preserves the polynomial complexity of this contractor.

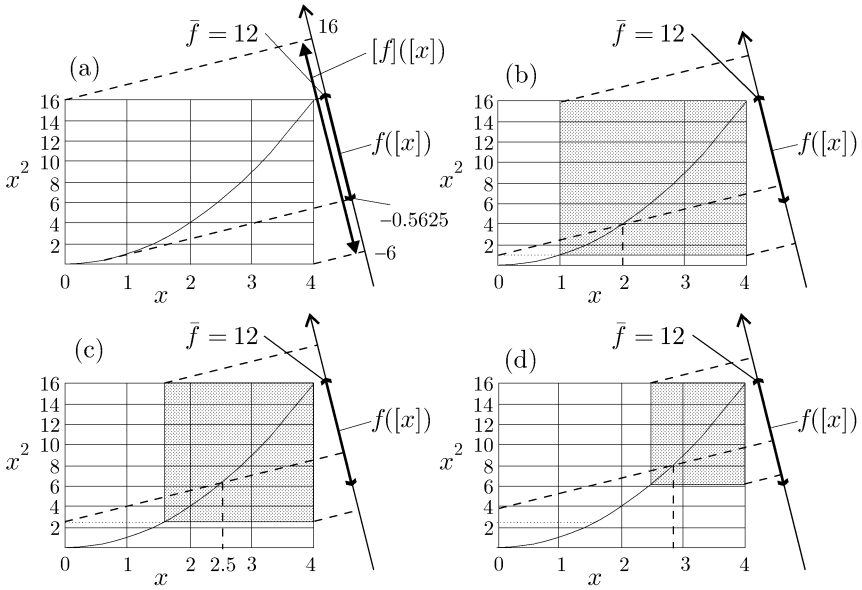


Fig. 4.7. Contracting procedure to compute an upper bound for $f([x])$; as the scales on the two axes differ, orthogonal projection does not actually correspond to the directions indicated in dotted lines, which should only be interpreted symbolically

4.5 Contractors for Sets

4.5.1 Definitions

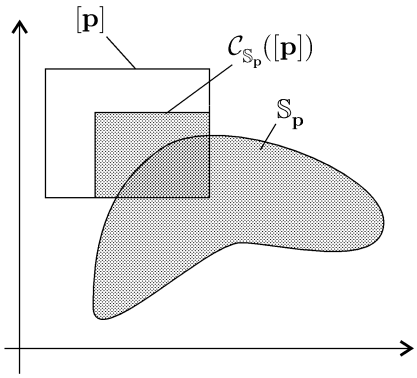


Fig. 4.8. Contractor for sets

The operator $\mathcal{C}_{\mathbb{S}_\mathrm{p}} : \mathbb{IR}^{n_\mathrm{p}} \to \mathbb{IR}^{n_\mathrm{p}}$ is a contractor for a set $\mathbb{S}_\mathrm{p}$ of $\mathbb{R}^{n_\mathrm{p}}$ if it satisfies

$$\forall [\mathbf{p}] \in \mathbb{IR}^{n_\mathrm{p}}, \begin{cases} \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) \subset [\mathbf{p}] & \text{(contractance)}, \\ \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) \cap \mathbb{S}_\mathrm{p} = [\mathbf{p}] \cap \mathbb{S}_\mathrm{p} & \text{(correctness)}, \end{cases} \tag{4.114}$$

as illustrated by Figure 4.8. Even if the notions of contractors for sets and for CSPs defining these sets can be considered as equivalent, contractors for sets will often be preferred in later chapters. They simplify the presentation of algorithms and make it possible to avoid the use of the terminology of CSPs.

Properties of contractors for sets are presented in Table 4.15.

Table 4.15. Properties of contractors for sets

$\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ is *monotonic* iff	$[\mathbf{p}] \subset [\mathbf{q}] \Rightarrow \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) \subset \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{q}])$
$\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ is *minimal* iff	$\forall [\mathbf{p}] \in \mathbb{IR}^{n_\mathrm{p}},\ \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) = [[\mathbf{p}] \cap \mathbb{S}_\mathrm{p}]$
$\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ is *thin* iff	$\forall \mathbf{p} \in \mathbb{R}^{n_\mathrm{p}},\ \mathcal{C}_{\mathbb{S}_\mathrm{p}}(\mathbf{p}) = \{\mathbf{p}\} \cap \mathbb{S}_\mathrm{p}$
$\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ is *idempotent* iff	$\forall [\mathbf{p}] \in \mathbb{IR}^{n_\mathrm{p}}, \mathcal{C}_{\mathbb{S}_\mathrm{p}}\left(\mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}])\right) = \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}])$
$\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ is *more contracting* than $\mathcal{C}'_{\mathbb{S}_\mathrm{p}}$ iff	$\forall [\mathbf{p}] \in \mathbb{IR}^{n_\mathrm{p}}, \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) \subset \mathcal{C}'_{\mathbb{S}_\mathrm{p}}([\mathbf{p}])$

Let $\mathcal{C}_{\mathbb{S}^1_\mathrm{p}}$ and $\mathcal{C}_{\mathbb{S}^2_\mathrm{p}}$ be two monotonic contractors for $\mathbb{S}^1_\mathrm{p}$ and $\mathbb{S}^2_\mathrm{p}$ and define

$$\mathcal{C}_{\mathbb{S}^1_\mathrm{p}} \cap \mathcal{C}_{\mathbb{S}^2_\mathrm{p}}([\mathbf{p}]) \triangleq \mathcal{C}_{\mathbb{S}^1_\mathrm{p}}([\mathbf{p}]) \cap \mathcal{C}_{\mathbb{S}^2_\mathrm{p}}([\mathbf{p}]), \tag{4.115}$$

$$\mathcal{C}_{\mathbb{S}^1_\mathrm{p}} \sqcup \mathcal{C}_{\mathbb{S}^2_\mathrm{p}}([\mathbf{p}]) \triangleq \mathcal{C}_{\mathbb{S}^1_\mathrm{p}}([\mathbf{p}]) \sqcup \mathcal{C}_{\mathbb{S}^2_\mathrm{p}}([\mathbf{p}])\,. \tag{4.116}$$

It is trivial to show that the following properties hold true:

$$\begin{array}{ll} (i) & \mathbb{S}^1_\mathrm{p} \subset \mathbb{S}^2_\mathrm{p} \Rightarrow \mathcal{C}_{\mathbb{S}^2_\mathrm{p}} \text{ is also a contractor for } \mathbb{S}^1_\mathrm{p}, \\ (ii) & \mathcal{C}_{\mathbb{S}^1_\mathrm{p}} \cap \mathcal{C}_{\mathbb{S}^2_\mathrm{p}} \text{ is a contractor for } \mathbb{S}^1_\mathrm{p} \cap \mathbb{S}^2_\mathrm{p}, \\ (iii) & \mathcal{C}_{\mathbb{S}^1_\mathrm{p}} \sqcup \mathcal{C}_{\mathbb{S}^2_\mathrm{p}} \text{ is a contractor for } \mathbb{S}^1_\mathrm{p} \sqcup \mathbb{S}^2_\mathrm{p}. \end{array} \tag{4.117}$$

The property (i) will be used to contract domains for optimization problems and (iii) is useful to develop contractors for problems involving a disjunction of constraints (*i.e.*, the Boolean operator `OR` is involved).

Example 4.16 *Consider an inclusion test* $[t_{\mathbb{S}_\mathrm{p}}]$ *for the set* $\mathbb{S}_\mathrm{p}$*. A contractor* $\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ *for* $\mathbb{S}_\mathrm{p}$ *is given by*

$$\begin{array}{l} \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) = \varnothing \text{ if } [t_{\mathbb{S}_\mathrm{p}}]([\mathbf{p}]) = 0, \\ \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}]) = [\mathbf{p}] \text{ otherwise.} \end{array} \tag{4.118}$$

This contractor is thin if and only if $[t_{\mathbb{S}_\mathrm{p}}]$ *is thin.* ■

4.5.2 Sets defined by equality and inequality constraints

The contractor $\mathcal{C}_\infty$ developed in Section 4.4 for CSPs of the form $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ can be used to build efficient contractors for more general classes of sets. Consider, for instance, a set $\mathbb{S}_\mathrm{p}$ defined by equality and inequality constraints:

$$\mathbb{S}_\mathrm{p} = \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid \mathbf{g}(\mathbf{p}) \leqslant \mathbf{0}, \mathbf{h}(\mathbf{p}) = \mathbf{0}\}, \tag{4.119}$$

where $\mathbf{g}$ and $\mathbf{h}$ are non-linear vector functions and the inequality is to be understood component-wise. Set

$$\mathbf{x} = \begin{pmatrix} \mathbf{p} \\ \mathbf{v} \end{pmatrix}, [\mathbf{x}] = [\mathbf{p}] \times [\mathbf{v}] \text{ and } \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \mathbf{g}(\mathbf{p}) + \mathbf{v} \\ \mathbf{h}(\mathbf{p}) \end{pmatrix}, \tag{4.120}$$

where $\mathbf{v}$ is a vector of slack variables with domain $[\mathbf{v}] = [0, \infty[\times \cdots \times [0, \infty[$. Since

$$\begin{pmatrix} \mathbf{f}(\mathbf{x}) = \mathbf{0} \\ \mathbf{x} \in [\mathbf{x}] \end{pmatrix} \Leftrightarrow \begin{pmatrix} \mathbf{g}(\mathbf{p}) + \mathbf{v} = \mathbf{0} \\ \mathbf{h}(\mathbf{p}) = \mathbf{0} \\ \mathbf{p} \in [\mathbf{p}] \\ \mathbf{v} \in [0, \infty[\times \cdots \times [0, \infty[\end{pmatrix} \Leftrightarrow \begin{pmatrix} \mathbf{g}(\mathbf{p}) \leqslant \mathbf{0} \\ \mathbf{h}(\mathbf{p}) = \mathbf{0} \\ \mathbf{p} \in [\mathbf{p}] \end{pmatrix}, \tag{4.121}$$

a contractor $\mathcal{C}$ for the CSP $\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}])$ can be used as a contractor for the set $\mathbb{S}_\mathrm{p}$ using the algorithm described in Table 4.16. Step 1 defines the CSP associated with the set $\mathbb{S}_\mathrm{p}$, Step 2 contracts the domain for the extended box $[\mathbf{x}]$ and Step 3 computes the projection of $[\mathbf{x}]$ onto the $\mathbf{p}$-space.

Table 4.16. Contractor for a set defined by equality and inequality constraints

Algorithm $\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ (in: $\mathbf{g}, \mathbf{h}$; inout: $[\mathbf{p}]$)	
1	$\mathbf{x} := (\mathbf{p}, \mathbf{v})$; $[\mathbf{x}] := [\mathbf{p}] \times [0, \infty[^{\times n_\mathrm{g}}$; $\mathbf{f}(\mathbf{x}) := (\mathbf{g}(\mathbf{p}) + \mathbf{v}, \mathbf{h}(\mathbf{p}))$;
2	$[\mathbf{x}] := \mathcal{C}([\mathbf{x}])$;
3	$[\mathbf{p}] := \mathrm{proj}_{\mathbb{R}^{n_\mathrm{p}}}[\mathbf{x}]$. // projection of $[\mathbf{x}]$ onto $\mathbf{p}$-space

4.5.3 Improving contractors using local search

Consider a set $\mathbb{S}_\mathrm{p}$ with a non-zero volume, a contractor $\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ for $\mathbb{S}_\mathrm{p}$ and a box $[\mathbf{p}]$ to be contracted as illustrated by Figure 4.9a. Some known feasible points are represented by black dots and $[\mathbf{r}]$ denotes the smallest box containing them. The only parts of $[\mathbf{p}]$ that may be eliminated by contraction are those in $[\mathbf{p}]$ and outside $[\mathbf{r}]$. Let $[\mathbf{q}]$ be a box with a face in common with $[\mathbf{p}]$ and touching $[\mathbf{r}]$ as indicated in Figure 4.9a. A contraction of $[\mathbf{q}]$ can be extended to $[\mathbf{p}]$ as

shown in Figure 4.9b. This technique can be employed in $\mathcal{C}_{\mathbb{S}_p}$ to make it more efficient. Moreover, the fact that $[\mathbf{p}]$ and $[\mathbf{r}]$ are almost equal can be used as a stopping criterion for the contractor. To be efficient, local searches should inflate $[\mathbf{r}]$ as much as possible; see Section 5.4, page 111, for more details.

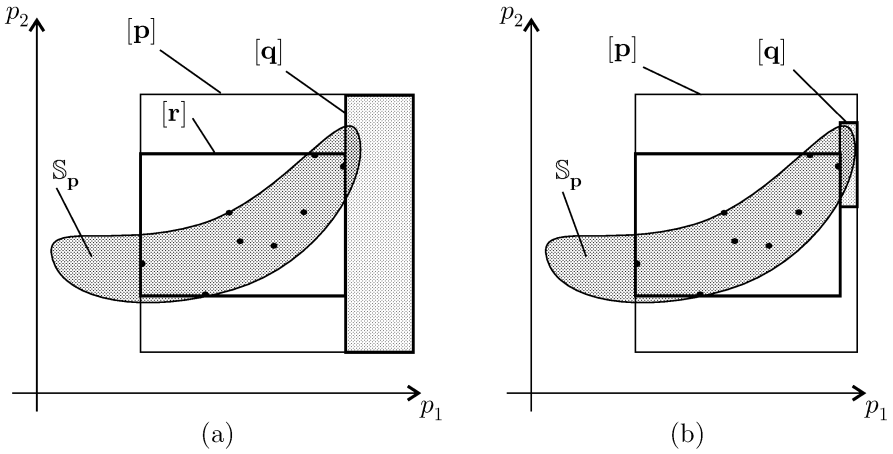


Fig. 4.9. Feasible points make it possible to improve the efficiency of contractors

4.6 Conclusions

This chapter has presented the important notion of contractor, used to downsize the search box without losing any of the solutions of the problem of interest. Contractors are basic ingredients of the solvers to be presented in the next chapter. As illustrated by Figure 4.10, solvers use inclusion functions and interval computation only through contractors. Note that even if many contractors are based on inclusion functions, some contractors use other types of tools. This is the case, for instance, for the contractors developed in the context of constraint propagation over continuous domains. The notion of contractor generalizes that of inclusion test presented in Chapter 2, in the sense that an inclusion test applied to a box $[\mathbf{x}]$ can be seen as a special contractor that returns either $[\mathbf{x}]$ itself or the empty set. As shown in Section 4.4.2, contractors may also be helpful to improve the quality of inclusion functions. This is represented by the upward arrow of Figure 4.10.

Contractors are requested to have a polynomial complexity in time and space, and thus not allowed to bisect domains. As a result, they may reach deadlocks, as illustrated by Example 4.14. Bisection will be a way out of such deadlocks. The idea is to split the box $[\mathbf{x}]$ into two subboxes, and to

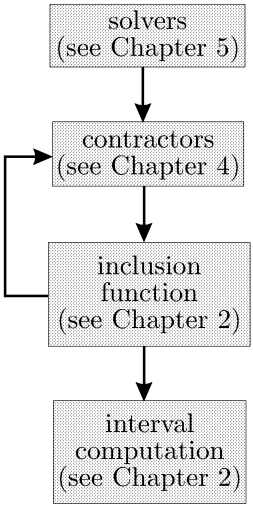


Fig. 4.10. Solvers call contractors, contractors use inclusion functions that require interval computation; the accuracy of the inclusion functions can be improved by using contractors

attempt contraction on each of them. When the dimension of $[\mathbf{x}]$ is large, this should be a last resort, because a bisection in one direction is often followed by bisections in the others, and the complexity then becomes exponential. For instance, in dimension 20, a bisection in each direction of a single box generates more than one million boxes.

Recent results have shown that if a limitation is set on the number of components of $\mathbf{x}$ allowed to be bisected, it is possible to keep the complexity of the contractor polynomial. The corresponding methods are based for instance on 3-B-consistency (Lhomme and Rueher, 1997) or on box-consistency (Benhamou et al., 1999). A result that seems even more promising is the algorithm based on (3-2)-consistency (Sam-Haroud, 1995; Lottaz, 2000), which provides a polynomial contractor that is optimal for a huge class of CSPs, in the sense that it generates the smallest box that contains the solution set. It suffices that the CSP involves constraints that are at most ternary (this can be achieved by decomposing the CSP into primitive constraints) and that these constraints satisfy some row-convexity conditions. Unfortunately, even if this contractor is polynomial, it requires computation with five-dimensional subpavings, which turns out to be extremely difficult with present-day computers.

The contractors presented in this chapter will be important ingredients of the solvers to be presented in the next.

5. Solvers

5.1 Introduction

Chapter 4 presented contractors that make it possible to contain a compact set $\mathbb{S}$ defined by non-linear equations and inequalities in a box. Although the results are guaranteed, the accuracy with which $\mathbb{S}$ is characterized is not under control. On the other hand, bisection allows accuracy to be controlled, but causes exponential complexity. Bisection should therefore be avoided as much as possible when the number of variables is high, in an attempt to escape the curse of dimensionality. This is why, in our opinion, when many variables are involved *bisection should be used as a last resort*, only when all available contractors have failed. A decision may then have to be taken as to which variable domains should be bisected.

All the solvers proposed in this chapter partition the search box into a union of boxes (*the paving*). The paving is generally built by the solver itself. On each box of this paving, contractors, inclusion tests and local optimization procedures are called. All of these procedures have a polynomial complexity. The results returned by the solvers depend only on the results obtained for each box of the paving. The precision of the solver is controlled by coefficients specifying, for example, the width ε of the smallest boxes of the paving, or the accuracy in the localization of a global optimum. For a given problem, the *accumulation set* of a solver is the set where boxes with width less than ε accumulate when ε tends to zero. The computing time of the solver increases quickly with the dimension and size of this accumulation set.

To illustrate the methodology followed to obtain efficient solvers, several problems will be considered. Section 5.2 is about solving systems of non-linear equations where the number of equations is equal to the number of variables. The characterization of a set defined by non-linear inequalities will be performed in Section 5.3 by bracketing this set between two subpavings. Section 5.4 addresses the problem of finding the smallest box containing a set defined by non-linear inequalities. Section 5.5 deals with the minimization of a cost function under equality and inequality constraints. The approach is then extended in Section 5.6 to the difficult problem of minimax optimization. Section 5.7 presents a method for characterizing level sets of a cost function.

L. Jaulin et al., *Applied Interval Analysis*

5.2 Solving Square Systems of Non-linear Equations

Consider n variables linked by n equations:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0, \\ \quad\vdots \qquad\qquad \vdots \ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0, \end{cases} \tag{5.1}$$

or equivalently, in vector form,

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}. \tag{5.2}$$

The problem to be solved is to characterize the set $\mathbb{S}_x$ of all the vectors $\mathbf{x}$ that satisfy $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ and belong to a (possibly very large) search box $[\mathbf{x}]$. The recursive algorithm of Table 5.1 computes a subpaving that contains $\mathbb{S}_x$. We called this algorithm SIVIAX because the search space corresponds to the entire vector $\mathbf{x}$ whereas in Section 5.3 the algorithm SIVIAP will deal with a subvector $\mathbf{p}$ of $\mathbf{x}$. $\mathcal{L}$ is initialized as the empty list and ε is a small positive real number. The union of all the boxes in the list $\mathcal{L}$ returned by SIVIAX contains $\mathbb{S}_x$. $\mathcal{C}_{\mathbb{S}_x}$ used at Step 1 is a contractor for $\mathbb{S}_x$, *i.e.*, it satisfies $\mathcal{C}_{\mathbb{S}_x}([\mathbf{x}]) \cap \mathbb{S}_x = [\mathbf{x}] \cap \mathbb{S}_x$ (see Section 4.5). Chapter 4 suggested a number of such contractors.

Table 5.1. Algorithm SIVIAX for solving a set of non-linear equations

Algorithm SIVIAX(in: $[\mathbf{x}], \mathcal{C}_{\mathbb{S}_x}, \varepsilon$; inout: $\mathcal{L}$)	
1	$[\mathbf{x}] := \mathcal{C}_{\mathbb{S}_x}([\mathbf{x}])$;
2	if $([\mathbf{x}] = \varnothing)$ then return;
3	if $(w([\mathbf{x}]) < \varepsilon)$ then
4	$\quad \mathcal{L} := \mathcal{L} \cup \{[\mathbf{x}]\}$; return;
5	bisect $[\mathbf{x}]$ into $[\mathbf{x}_1]$ and $[\mathbf{x}_2]$;
6	SIVIAX$([\mathbf{x}_1], \mathcal{C}_{\mathbb{S}_x}, \varepsilon, \mathcal{L})$; SIVIAX$([\mathbf{x}_2], \mathcal{C}_{\mathbb{S}_x}, \varepsilon, \mathcal{L})$.

Remark 5.1 *For some applications, it is useful to test whether there exists a unique solution of* $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ *in a given box* $[\mathbf{x}]$ *of* $\mathcal{L}$. *If* $\mathcal{C}_{\mathrm{N}}([\mathbf{x}])$, $\mathcal{C}_{\mathrm{NP}}([\mathbf{x}])$ *or* $\mathcal{C}_{\mathrm{K}}([\mathbf{x}])$ *is strictly inside* $[\mathbf{x}]$, *then there exists a unique solution of* $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ *in* $[\mathbf{x}]$ *(Hansen, 1992b).* ■

To bisect $[\mathbf{x}]$ into two boxes at Step 5, one may cut it at its centre, perpendicularly to one of its edges of maximum length. But, as we shall see now, when the problem is ill conditioned, such a bisection may be inefficient and one should find a more suitable criterion to select the direction along

which the bisection should be conducted (Ratschek and Rokne, 1995; Ratz and Csendes, 1995).

Define the ith *symmetry segment* and the ith *symmetry (hyper) plane* of $[\mathbf{x}]$ as follows:

$$\begin{aligned}\text{segm}_i([\mathbf{x}]) &\triangleq m_1 \times \ldots \times m_{i-1} \times [x_i] \times m_{i+1} \times \ldots \times m_n,\\ \text{plane}_i([\mathbf{x}]) &\triangleq [x_1] \times \ldots \times [x_{i-1}] \times m_i \times [x_{i+1}] \times \ldots \times [x_n],\end{aligned}$$

where $m_k = \text{mid}([x_k])$. Note that $\text{segm}_i([\mathbf{x}])$ and $\text{plane}_i([\mathbf{x}])$ are orthogonal. These definitions are illustrated by Figure 5.1, where $n = 3$.

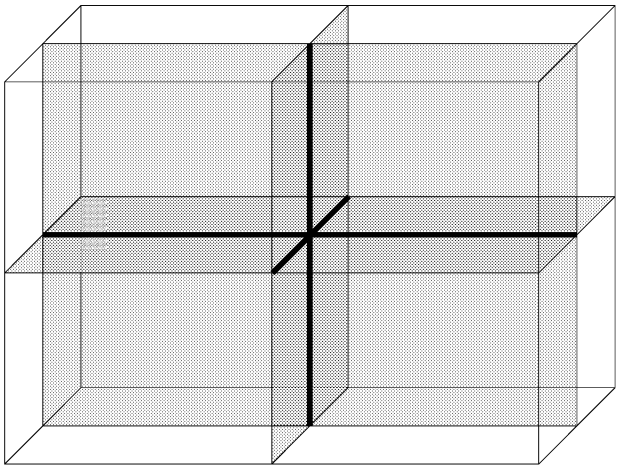


Fig. 5.1. Three-dimensional cube with its three symmetry planes and its three symmetry segments

Figures 5.2 to 5.4 show that the efficiency of bisection may strongly depend on the choice of the plane along which this bisection is performed. In these pictures, the inclusion function $[\mathbf{f}]$ for $\mathbf{f}$ is minimal, but this is not required. Figure 5.2 illustrates a situation where $\mathbf{f}$ is ill conditioned and where the box $[\mathbf{x}]$ is assumed small enough to allow a linear approximation of the behaviour of $\mathbf{f}$ over $[\mathbf{x}]$. The symmetry segments of $[\mathbf{x}]$ as well as their images are represented with thin lines. A bisection along $\text{plane}_1([\mathbf{x}])$, as in Figure 5.3, marginally improves the description of the behaviour of $\mathbf{f}$ over $[\mathbf{x}]$, contrary to a bisection along $\text{plane}_2([\mathbf{x}])$, as in Figure 5.4, which is much more efficient. It then seems rather natural to bisect $[\mathbf{x}]$ along the symmetry plane orthogonal to the symmetry segment along which $\mathbf{f}$ is the most sensitive, by choosing the index i that maximizes

$$\mu_1(i) = \max_{j\in\{1,\ldots,n\}} w\left(f_j\left(\text{segm}_i\left([\mathbf{x}]\right)\right)\right). \tag{5.3}$$

This criterion can be approximated by

$$\mu_2(i) = \max_{j\in\{1,\ldots,n\}} w([x_i]) \left| \frac{\partial f_j}{\partial x_i} \left(\text{mid}\left([\mathbf{x}]\right)\right) \right|, \tag{5.4}$$

which is much easier to compute. If the *efficiency* of the bisection of $[\mathbf{x}]$ into $[\mathbf{x}_1]$ and $[\mathbf{x}_2]$ is quantified by

$$\frac{\text{vol}([\mathbf{f}]([\mathbf{x}]))}{\text{vol}([\mathbf{f}]([\mathbf{x}_1]) \cup [\mathbf{f}]([\mathbf{x}_2]))}, \tag{5.5}$$

then the bisection of Figure 5.3 is about half as efficient as that of Figure 5.4.

Once the direction i of bisection has been chosen, it may be more interesting to translate the bisection plane by cutting along

$$\begin{aligned}\text{plane}_i(\alpha, [\mathbf{x}]) = {} & [x_1] \times \cdots \times [x_{i-1}] \\ & \times (\alpha \underline{x}_i + (1-\alpha)\bar{x}_i) \times [x_{i+1}] \times \cdots \times [x_n],\end{aligned} \tag{5.6}$$

where $\alpha \in]0,1[$, with $\alpha = 0.5$ corresponding to a symmetry plane. This is illustrated by Figures 5.5 to 5.7. In Figure 5.5 four horizontal and vertical segments of $[\mathbf{x}]$ are represented with thin lines, as well as their images by $\mathbf{f}$. The criterion (5.4) suggests that the bisection should be performed for $i = 2$. For $\alpha = 0.5$, one obtains the situation depicted in Figure 5.6. The bisection of Figure 5.7, which corresponds to $\alpha = 0.2$, leads to better efficiency in the sense of (5.5). A suitable choice for α generally reduces the number of bisections performed by SIVIAX but the improvement is significant only when the problem is very ill conditioned. For simplicity, in this book, we shall only consider bisections along symmetry planes.

5.3 Characterizing Sets Defined by Inequalities

Consider the set of non-linear inequalities

$$\begin{cases} g_1(p_1, p_2, \ldots, p_{n_\text{p}}) \in [y_1], \\ \qquad\vdots \qquad\qquad \vdots \quad \vdots \\ g_{n_\text{g}}(p_1, p_2, \ldots, p_{n_\text{p}}) \in [y_{n_\text{g}}], \end{cases} \tag{5.7}$$

i.e., in vector form,

$$\mathbf{g}(\mathbf{p}) \in [\mathbf{y}], \tag{5.8}$$

where $\mathbf{p}$ is assumed to belong to the prior search box $[\mathbf{p}]$. The problem to be treated in this section is the characterization of the set

$$\mathbb{S}_\text{p} \triangleq \{\mathbf{p} \in [\mathbf{p}] \mid \mathbf{g}(\mathbf{p}) \in [\mathbf{y}]\} = \mathbf{g}^{-1}([\mathbf{y}]) \cap [\mathbf{p}]. \tag{5.9}$$

Denote by $\mathbf{x}$ the vector obtained by appending $\mathbf{y}$ to $\mathbf{p}$, and by $[\mathbf{x}]$ the box $[\mathbf{p}] \times [\mathbf{y}]$. For some applications, it is also of interest to characterize $\mathbb{S}_\text{x}$ and $\mathbb{S}_\text{y}$ defined by

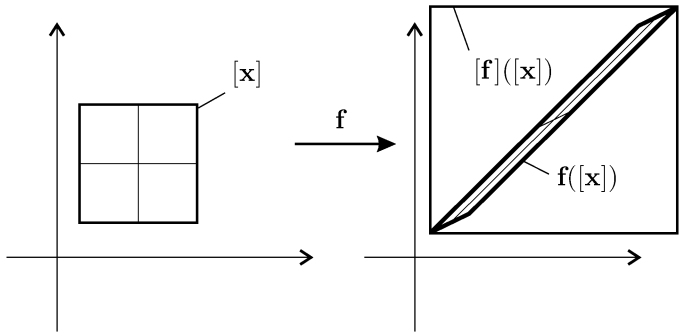


Fig. 5.2. Situation where **f** is ill conditioned

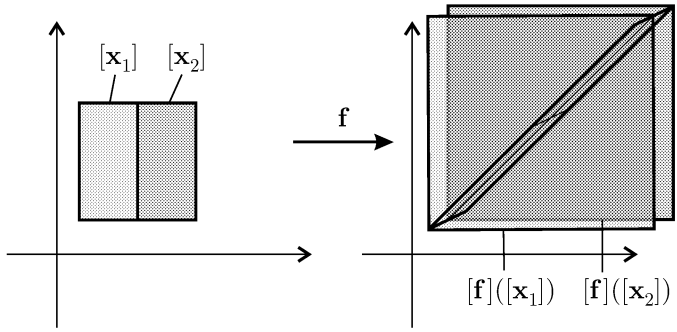


Fig. 5.3. Inefficient bisection

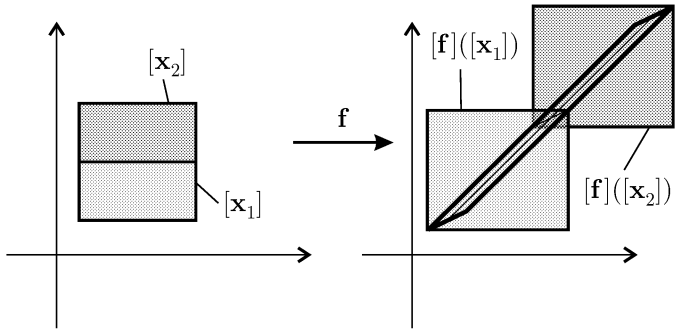


Fig. 5.4. Efficient bisection

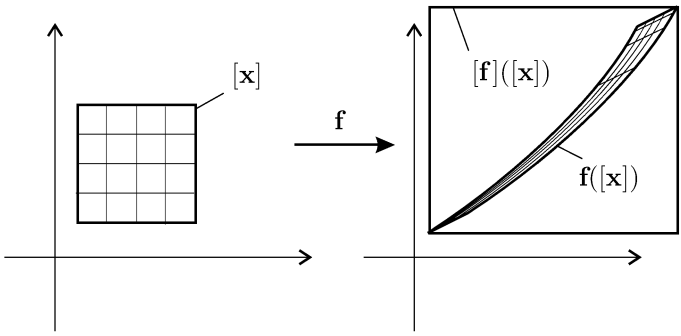


Fig. 5.5. Situation where **f** is ill conditioned

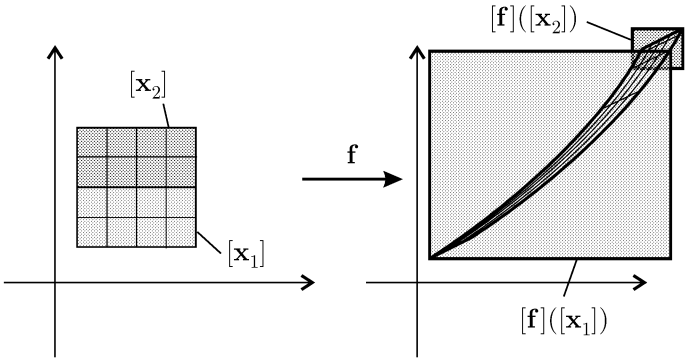


Fig. 5.6. Bisection at the centre ($\alpha = 0.5$)

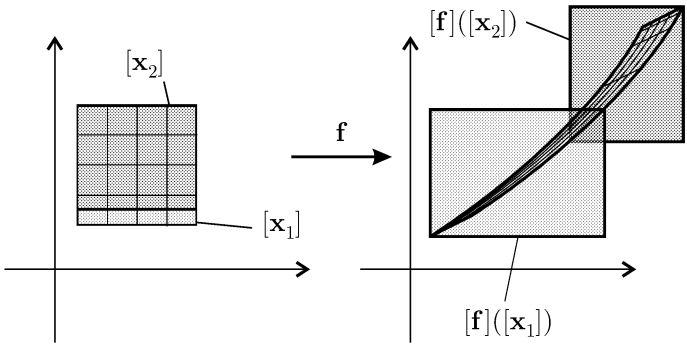


Fig. 5.7. A more efficient bisection ($\alpha = 0.2$)

$$\mathbb{S}_x \triangleq \left\{ \mathbf{x} = \begin{pmatrix} \mathbf{p} \\ \mathbf{y} \end{pmatrix} \mid \mathbf{y} = \mathbf{g}(\mathbf{p}),\ \mathbf{y} \in [\mathbf{y}],\ \mathbf{p} \in [\mathbf{p}] \right\}, \tag{5.10}$$

$$\mathbb{S}_y \triangleq \{\mathbf{g}(\mathbf{p}) \mid \mathbf{p} \in [\mathbf{p}],\ \mathbf{g}(\mathbf{p}) \in [\mathbf{y}]\} = \mathbf{g}([\mathbf{p}]) \cap [\mathbf{y}]. \tag{5.11}$$

Note that $\mathbb{S}_p$ is the orthogonal projection of $\mathbb{S}_x$ onto the $\mathbf{p}$-space and that $\mathbb{S}_y$ is its orthogonal projection onto the $\mathbf{y}$-space, as illustrated by Figure 5.8.

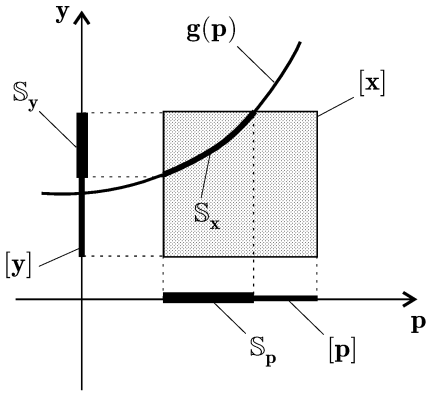


Fig. 5.8. Feasible sets $\mathbb{S}_x$, $\mathbb{S}_y$ and $\mathbb{S}_p$

Outer approximations for $\mathbb{S}_p$, $\mathbb{S}_y$ and $\mathbb{S}_x$ and an inner approximation for $\mathbb{S}_p$ can be obtained with arbitrary precision by using the recursive algorithm SIVIAPY given in Table 5.2.

Table 5.2. Algorithm SIVIAPY

Algorithm SIVIAPY(in: $[\mathbf{x}], \mathcal{C}_{\mathbb{S}_x}, \varepsilon$; inout: $\mathcal{L}$)	
1	$[\mathbf{x}] := \mathcal{C}_{\mathbb{S}_x}([\mathbf{x}])$;
2	if $([\mathbf{x}] = \varnothing)$ then return;
3	$([\mathbf{p}], [\mathbf{y}]) := [\mathbf{x}]$;
4	if $(w([\mathbf{x}]) < \varepsilon)$ then
5	$\quad \mathcal{L} := \mathcal{L} \cup \{[\mathbf{x}]\}$; return;
6	bisect $([\mathbf{p}])$ into $[\mathbf{p}_1]$ and $[\mathbf{p}_2]$;
7	$[\mathbf{x}_1] := ([\mathbf{p}_1], [\mathbf{y}])$; $[\mathbf{x}_2] := ([\mathbf{p}_2], [\mathbf{y}])$;
8	SIVIAPY$([\mathbf{x}_1], \mathcal{C}_{\mathbb{S}_x}, \varepsilon, \mathcal{L})$; SIVIAPY$([\mathbf{x}_2], \mathcal{C}_{\mathbb{S}_x}, \varepsilon, \mathcal{L})$.

In contrast to the algorithm SIVIAX of Section 5.2, where a single type of box is considered, SIVIAPY distinguishes $[\mathbf{p}]$ and $[\mathbf{y}]$, in order to allow the

characterization of $\mathbb{S}_{\mathrm{p}}$, $\mathbb{S}_{\mathrm{y}}$ and $\mathbb{S}_{\mathrm{x}}$. $\mathcal{L}$ is initialized as the empty list. After completion of SIVIAPY, $\mathcal{L}$ can be written as

$$\mathcal{L} = \{[\mathbf{x}_1],[\mathbf{x}_2],\ldots,[\mathbf{x}_{\overline{k}}]\}, \tag{5.12}$$

or equivalently as

$$\mathcal{L} = \left\{([\mathbf{p}_1],[\mathbf{y}_1]),([\mathbf{p}_2],[\mathbf{y}_2]),\ldots,\left([\mathbf{p}_{\overline{k}}],[\mathbf{y}_{\overline{k}}]\right)\right\}. \tag{5.13}$$

At Step 1, $\mathcal{C}_{\mathbb{S}_{\mathrm{x}}}$ is a contractor for the set $\mathbb{S}_{\mathrm{x}}$ (see Section 4.5, page 97). From the list $\mathcal{L}$ generated by SIVIAPY, outer approximations $\overline{\mathbb{S}}_{\mathrm{p}}$, $\overline{\mathbb{S}}_{\mathrm{y}}$ and $\overline{\mathbb{S}}_{\mathrm{x}}$ for $\mathbb{S}_{\mathrm{p}}$, $\mathbb{S}_{\mathrm{y}}$ and $\mathbb{S}_{\mathrm{x}}$ can be obtained by

$$\overline{\mathbb{S}}_{\mathrm{p}} = \bigcup_{k=1,\ldots,\overline{k}} [\mathbf{p}_k],\ \overline{\mathbb{S}}_{\mathrm{y}} = \bigcup_{k=1,\ldots,\overline{k}} [\mathbf{y}_k] \quad \text{and} \quad \overline{\mathbb{S}}_{\mathrm{x}} = \bigcup_{k=1,\ldots,\overline{k}} [\mathbf{x}_k] \tag{5.14}$$

and an inner approximation for $\mathbb{S}_{\mathrm{p}}$ is obtained as

$$\underline{\mathbb{S}}_{\mathrm{p}} = \bigcup_{k=1,\ldots,\overline{k}} \{[\mathbf{p}_k] \mid [\mathbf{g}]([\mathbf{p}_k]) \subset [\mathbf{y}]\}, \tag{5.15}$$

where $[\mathbf{g}]$ is an inclusion function for $\mathbf{g}$. Various strategies can be considered for the bisection of $[\mathbf{p}]$ at Step 6. When the problem is well conditioned, for simplicity, bisection is along a *principal plane* of $[\mathbf{p}]$, *i.e.*, along a symmetry plane orthogonal to one of the edges of maximum length. Otherwise, bisection is performed perpendicularly to the direction i that maximizes

$$\max_{j\in\{1,\ldots,n_{\mathrm{g}}\}} w([p_i]) \left|\frac{\partial g_j}{\partial p_i}(\mathrm{mid}([\mathbf{p}]))\right|. \tag{5.16}$$

The subpaving $\overline{\mathbb{S}}_{\mathrm{p}}$ generated by SIVIAPY in the $\mathbf{p}$-space accumulates on the set $\mathbb{S}_{\mathrm{p}}$, which generically has a dimension equal to n_{p}. This means that when the dimension of $\mathbf{p}$ is large (typically greater than four) and when high accuracy is required for the characterization of $\mathbb{S}_{\mathrm{p}}$, no computer will be able to complete SIVIAPY in a reasonable time. When one is interested in $\mathbb{S}_{\mathrm{p}}$ only and not in $\mathbb{S}_{\mathrm{y}}$ and $\mathbb{S}_{\mathrm{x}}$, the dimension of the accumulation subpaving can be reduced to $n_{\mathrm{p}} - 1$. It suffices to store the current box $[\mathbf{p}]$ in $\underline{\mathbb{S}}_{\mathrm{p}}$ when the condition $[\mathbf{g}]([\mathbf{p}_k]) \subset [\mathbf{y}]$ is satisfied, and to remove it from the list of the boxes still to be bisected. The corresponding recursive algorithm SIVIAP, given in Table 5.3, is similar to the SIVIA algorithm of Section 3.4.1, page 55. The main difference is that SIVIAP uses contractors for $\mathbb{S}_{\mathrm{p}}$. The two subpavings $\underline{\mathbb{S}}_{\mathrm{p}}$ and $\overline{\mathbb{S}}_{\mathrm{p}}$ are initialized as the empty set.

At Step 1, SIVIAP uses a contractor $\mathcal{C}_{\mathbb{S}_{\mathrm{p}}}$ for $\mathbb{S}_{\mathrm{p}}$. We shall assume that $\mathcal{C}_{\mathbb{S}_{\mathrm{p}}}$ is either the contractor

$$\mathcal{C}^0_{\mathbb{S}_{\mathrm{p}}} : \begin{cases} \mathbb{IR}^{n_{\mathrm{p}}} \to & \mathbb{IR}^{n_{\mathrm{p}}} \\ [\mathbf{p}] \mapsto & \begin{cases} \varnothing & \text{if } [\mathbf{g}]([\mathbf{p}]) \cap [\mathbf{y}] = \varnothing \\ [\mathbf{p}] & \text{otherwise} \end{cases} \end{cases} \tag{5.17}$$

or a more efficient contractor, which may for instance include $\mathcal{C}^0_{\mathbb{S}_p}$ in its store (see Section 4.4, page 90). If $\mathcal{C}_{\mathbb{S}_p}$ has contracted $[\mathbf{p}]$ to $\varnothing$, then $[\mathbf{p}]$ is eliminated at Step 2. At Steps 3 and 4, if $[\mathbf{p}]$ is proved to be inside $\mathbb{S}_p$, then it is stored in $\underline{\mathbb{S}}_p$ and $\overline{\mathbb{S}}_p$. Steps 5 to 8 are similar to steps of SIVIAX (page 104). After completion of SIVIAP, we have

$$\underline{\mathbb{S}}_p \subset \mathbb{S}_p \subset \overline{\mathbb{S}}_p. \tag{5.18}$$

Table 5.3. Algorithm SIVIAP

Algorithm SIVIAP(in: $[\mathbf{p}], \mathcal{C}_{\mathbb{S}_p}, \mathbf{g}, [\mathbf{y}], \varepsilon$; inout: $\underline{\mathbb{S}}_p, \overline{\mathbb{S}}_p$)
1 $[\mathbf{p}] := \mathcal{C}_{\mathbb{S}_p}([\mathbf{p}])$;
2 if $([\mathbf{p}] = \varnothing)$ then return;
3 if $[\mathbf{g}]([\mathbf{p}]) \subset [\mathbf{y}]$
4 $\quad \underline{\mathbb{S}}_p := \underline{\mathbb{S}}_p \cup [\mathbf{p}]$; $\overline{\mathbb{S}}_p := \overline{\mathbb{S}}_p \cup [\mathbf{p}]$; return;
5 if $(w([\mathbf{p}]) < \varepsilon)$ then
6 $\quad \overline{\mathbb{S}}_p := \overline{\mathbb{S}}_p \cup [\mathbf{p}]$; return;
7 bisect $([\mathbf{p}])$ into $[\mathbf{p}_1]$ and $[\mathbf{p}_2]$;
8 SIVIAP$([\mathbf{p}_1], \mathcal{C}_{\mathbb{S}_p}, \mathbf{g}, [\mathbf{y}], \varepsilon, \underline{\mathbb{S}}_p, \overline{\mathbb{S}}_p)$; SIVIAP$([\mathbf{p}_2], \mathcal{C}_{\mathbb{S}_p}, \mathbf{g}, [\mathbf{y}], \varepsilon, \underline{\mathbb{S}}_p, \overline{\mathbb{S}}_p)$.

Example 5.1 *Consider again the problem of Example 3.2, page 58, which is the characterization of the set of vectors* $\mathbf{p}$ *that satisfy*

$$\begin{cases} \exp(p_1) + \exp(p_2) & \in [10, 11], \\ \exp(2p_1) + \exp(2p_2) & \in [62, 72]. \end{cases} \tag{5.19}$$

For $[\mathbf{p}] = [0, 4] \times [0, 4]$*, and* $\varepsilon = 0.001$*,* SIVIAP *generates a subpaving similar to that of Figure 3.9, page 58, in* 3.8 *s on a* PENTIUM 133*. With the same value of* ε*,* SIVIA *as presented in Chapter 3 would take* 6 *s. The improvement brought by* SIVIAP *increases with* n_p *and decreases when the size of* $\mathbb{S}_p$ *increases.* ■

The following section deals with the problem of finding the smallest box that contains $\mathbb{S}_p$.

5.4 Interval Hull of a Set Defined by Inequalities

Characterizing a (full) compact set $\mathbb{S}_p$ may turn out to be too costly when the dimension of $\mathbf{p}$ is high and when $\mathbb{S}_p$ is large, because the paving of all the boxes generated by SIVIA or SIVIAP accumulates on the boundary of $\mathbb{S}_p$. In the hope of computing less if less if asked for, consider now the problem of finding the *interval hull* $[\mathbb{S}_p]$ of $\mathbb{S}_p$ (the smallest box that contains it) instead of requesting a detailed characterization of $\mathbb{S}_p$.

This simplified characterization is important for many practical problems such as parameter estimation, where the interval components of the interval hull correspond to the parameter uncertainty intervals (see Chapter 6).

5.4.1 First approach

A first approach to solving this problem in a non-linear context is to decompose it into the $2n_{\mathrm{p}}$ optimization problems

$$\min_{\mathbf{p}\in\mathbb{S}_{\mathrm{p}}} p_i, \quad \max_{\mathbf{p}\in\mathbb{S}_{\mathrm{p}}} p_i, \quad i = 1, \ldots, n_{\mathrm{p}}. \tag{5.20}$$

Optimization may be based on signomial programming (Milanese and Vicino, 1991) or on interval analysis (Jaulin, 1994). For each of the optimization problems in (5.20), the paving generated by an interval technique for global optimization (such as the one to be presented in Section 5.5) accumulates on $\partial\mathbb{S}_{\mathrm{p}} \cap \partial\,[\mathbb{S}_{\mathrm{p}}]$, *i.e.*, on the part of the boundary of $\mathbb{S}_{\mathrm{p}}$ that belongs to the boundary of the interval hull of $\mathbb{S}_{\mathrm{p}}$. This is a drastic simplification compared to SIVIAP, which accumulates on the boundary of $\mathbb{S}_{\mathrm{p}}$; see Figure 5.9.

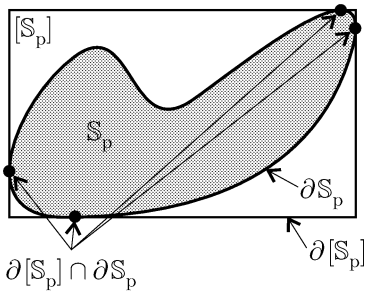
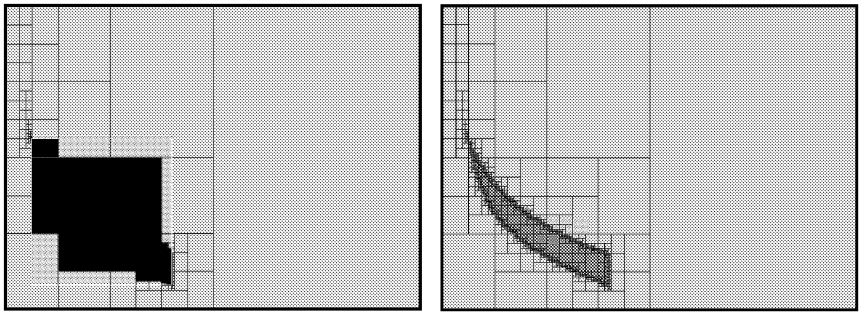


Fig. 5.9. The part of the boundary of $\mathbb{S}_{\mathrm{p}}$ that also belongs to the boundary of $[\mathbb{S}_{\mathrm{p}}]$ is indicated by black dots

Example 5.2 *Consider the problem (Jaulin, 1994) of characterizing the set*

$$\mathbb{S}_{\mathrm{p}} = \left\{(p_1, p_2) \in [0,5]^2 \mid \forall t \in [0,1],\ |t^2 + 2t + 1 - p_1 e^{p_2 t}| \leqslant 1\right\}.$$

The subpavings obtained when performing the four optimizations of (5.20) *are presented in Figure 5.10a. Compare with Figure 5.10b, which presents the subpavings generated by* SIVIAP *when solving the same example.* ■



(a) (b)

Fig. 5.10. Pavings generated when processing Example 5.2; (a) when evaluating the interval hull; (b) when using SIVIAP; the number of bisections is much smaller in (a) than in (b)

5.4.2 Second approach

A second approach based on interval analysis is to use the algorithm HULL (Jaulin, 2000a), which brackets $[\mathbb{S}_{\mathrm{p}}]$ between two boxes $[\mathbf{p}_{\mathrm{in}}]$ and $[\mathbf{p}_{\mathrm{out}}]$, as follows:

$$[\mathbf{p}_{\mathrm{in}}] \subset [\mathbb{S}_{\mathrm{p}}] \subset [\mathbf{p}_{\mathrm{out}}]. \tag{5.21}$$

The only assumptions are that a contractor is available for $\mathbb{S}_{\mathrm{p}}$, that it is possible to check whether a given point $\mathbf{p}$ belongs to $\mathbb{S}_{\mathrm{p}}$ and that a (possibly very large) box $[\mathbf{p}]$ containing $\mathbb{S}_{\mathrm{p}}$ is available. Instead of solving $2n_{\mathrm{p}}$ optimization problems, HULL generates two sequences of boxes $[\mathbf{p}_{\mathrm{in}}](k)$ and $[\mathbf{p}_{\mathrm{out}}](k)$ and a sequence of subpavings $\mathbb{L}(k)$ that satisfy

$$\begin{cases} [\mathbf{p}_{\mathrm{in}}](k) \subset [\mathbb{S}_{\mathrm{p}}], \\ \mathbb{S}_{\mathrm{p}} \subset \mathbb{L}(k) \cup [\mathbf{p}_{\mathrm{out}}](k), \\ [\mathbf{p}_{\mathrm{in}}](k) \subset [\mathbf{p}_{\mathrm{out}}](k). \end{cases} \tag{5.22}$$

The principle of HULL is illustrated by Figure 5.11. As suggested in Remark 3.1, page 51, $\mathbb{L}(k)$ will be denoted by $\mathcal{L}(k)$ when it is considered as a list of boxes.

HULL empties $\mathcal{L}(k)$ by increasing $[\mathbf{p}_{\mathrm{in}}](k)$ as much as possible and $[\mathbf{p}_{\mathrm{out}}](k)$ as little as possible, while satisfying the three conditions (5.22). The basic transformations employed to perform this task are described below. After each transformation, k is increased by 1. For simplicity, the dependency of $[\mathbf{p}_{\mathrm{in}}], [\mathbf{p}_{\mathrm{out}}]$ and $\mathcal{L}$ in k will be omitted.

1. **Inner inflation**: If a point $\tilde{\mathbf{p}} \in \mathbb{S}_{\mathrm{p}}$ is found outside $[\mathbf{p}_{\mathrm{in}}]$ by any local search (this is the case of the point represented by the black dot in

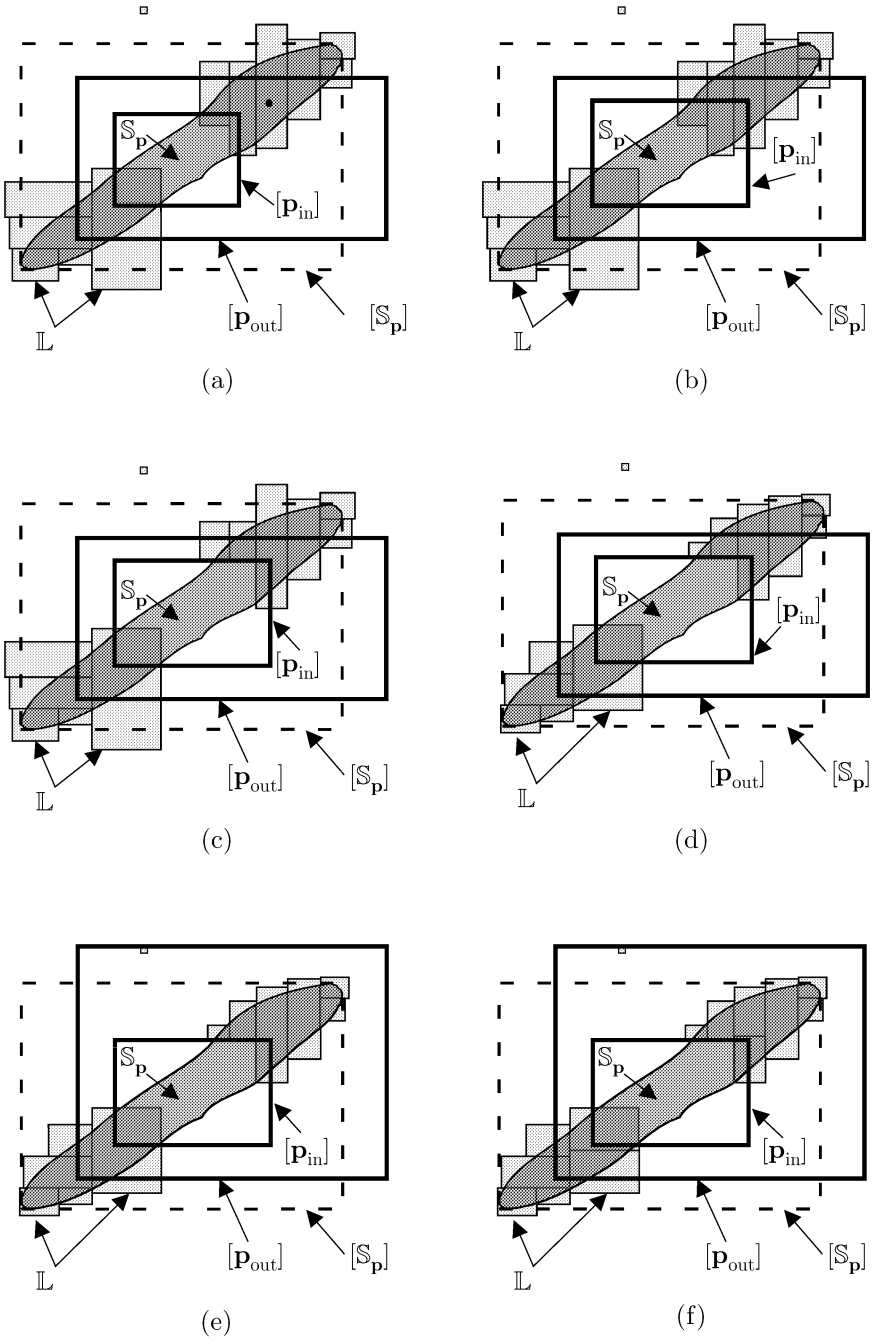


Fig. 5.11. Principle of HULL; (a) initial configuration, (b) inner inflation, (c) amputation, (d) contraction, (e) outer inflation, (f) bisection

Figure 5.11a), then set $[\mathbf{p}_{\text{in}}] := [\mathbf{p}_{\text{in}}] \sqcup \{\tilde{\mathbf{p}}\}$ and $[\mathbf{p}_{\text{out}}] := [\mathbf{p}_{\text{out}}] \sqcup \{\tilde{\mathbf{p}}\}$. Recall that the interval-union operator $\sqcup$ computes the smallest box that contains the union of its arguments, *i.e.*, $\mathbb{A} \sqcup \mathbb{B} = [\mathbb{A} \cup \mathbb{B}]$. Inner inflation is illustrated by Figure 5.11b.

2. **Amputation**: For any given $[\mathbf{p}]$ listed in $\mathcal{L}$, set $[\mathbf{p}] := [[\mathbf{p}] \setminus [\mathbf{p}_{\text{in}}]]$, where
$$[\mathbf{p}]\backslash[\mathbf{p}_{\text{in}}] \triangleq \{\mathbf{p} \in [\mathbf{p}] \mid \mathbf{p} \notin [\mathbf{p}_{\text{in}}]\}. \tag{5.23}$$
If, for instance, $[\mathbf{p}] \subset [\mathbf{p}_{\text{in}}]$ then $[\mathbf{p}]\backslash[\mathbf{p}_{\text{in}}] = \varnothing$ and the amputation amounts to removing $[\mathbf{p}]$ from $\mathcal{L}$. Note that the amputation is inefficient if $[\mathbf{p}]$ contains a corner of $[\mathbf{p}_{\text{in}}]$, since the box $[[\mathbf{p}]\backslash[\mathbf{p}_{\text{in}}]]$ is then equal to $[\mathbf{p}]$. In Figure 5.11c, the amputation has been efficient only for two boxes of $\mathcal{L}$.
3. **Contraction**: For any given $[\mathbf{p}]$ listed in $\mathcal{L}$, set $[\mathbf{p}] := \mathcal{C}_{\mathbb{S}_{\text{p}}}([\mathbf{p}])$, where $\mathcal{C}_{\mathbb{S}_{\text{p}}}$ is the available contractor for $\mathbb{S}_{\text{p}}$. See Figure 5.11d for an illustration.
4. **Outer inflation**: If the width of a box $[\mathbf{p}]$ listed in $\mathcal{L}$ is smaller than ε or if only a very small part of $[\mathbf{p}]$ is outside $[\mathbf{p}_{\text{out}}]$ (*i.e.*, $h^0_\infty([\mathbf{p}], [\mathbf{p}_{\text{out}}]) < \varepsilon$), then remove $[\mathbf{p}]$ from $\mathcal{L}$ and set $[\mathbf{p}_{\text{out}}] := [\mathbf{p}_{\text{out}}] \sqcup [\mathbf{p}]$. The small grey box at the top of Figure 5.11e is deemed too small for a bisection to be considered; $[\mathbf{p}_{\text{out}}]$ has thus been inflated in order to enclose it, even if $[\mathbf{p}]$ does not intersect $\mathbb{S}_{\text{p}}$.
5. **Bisection**: If the width of a box $[\mathbf{p}]$ listed in $\mathcal{L}$ is larger than ε, then it is bisected into two subboxes $[\mathbf{p}_1]$ and $[\mathbf{p}_2]$. In $\mathcal{L}$, $[\mathbf{p}]$ is then replaced by these two subboxes. Since bisection makes the complexity of HULL exponential with respect to n_{p}, it should only be performed as a last resort. This is illustrated on Figure 5.11f.

It is trivial to show that the three properties (5.22) remain satisfied after each transformation. For simplicity we have chosen a *first-in-first-out* structure for the list $\mathcal{L}$, even if other structures might be more suitable. The algorithm HULL is summarized in Table 5.4.
The following properties hold true (Jaulin, 2000a):

$$\exists \overline{k} > 0 \text{ such that } \mathbb{L}\left(\overline{k}\right) = \varnothing, \tag{5.24}$$

$$[\mathbf{p}_{\text{in}}](\overline{k}) \subset [\mathbb{S}_{\text{p}}] \subset [\mathbf{p}_{\text{out}}](\overline{k}), \tag{5.25}$$

where $\overline{k}$ is the value of k after completion of HULL. This means that HULL terminates and provides a guaranteed bracketing of $[\mathbb{S}_{\text{p}}]$. The analysis of the convergence of $[\mathbf{p}_{\text{in}}](\overline{k})$ and $[\mathbf{p}_{\text{out}}](\overline{k})$ towards $[\mathbb{S}_{\text{p}}]$ when ε tends to 0 remains to be carried out.

Example 5.3 *For the problem of Example 5.1,* HULL *finds in* 0.055 *s on a* PENTIUM 133 *the smallest box enclosing* $\mathbb{S}_{\text{p}}$*, with an accuracy of six digits. Figure 5.12 illustrates the technique. During the first iteration, at Steps 4, 5 and 6,* HULL *succeeds in scanning only the leftmost connected component of* $\mathbb{S}_{\text{p}}$*. It is thus able to inflate* $[\mathbf{p}_{\text{in}}]$ *and* $[\mathbf{p}_{\text{out}}]$ *so that they are almost equal to the*

Table 5.4. Algorithm for characterizing the interval hull of a set defined by nonlinear inequalities

Algorithm HULL(in: $[\mathbf{p}], \mathcal{C}_{\mathbb{S}_\mathrm{p}}, \varepsilon$; out: $[\mathbf{p}_\mathrm{in}]$, $[\mathbf{p}_\mathrm{out}]$)	
1	$[\mathbf{p}_\mathrm{in}] := \varnothing$; $[\mathbf{p}_\mathrm{out}] := \varnothing$; $\mathcal{L} := \{[\mathbf{p}]\}$;
2	repeat
3	pop first box out of $\mathcal{L}$ into $[\mathbf{p}]$;
4	with a local seach, initialized at mid($[\mathbf{p}]$), search for feasible points $\tilde{\mathbf{p}}$ outside $[\mathbf{p}_\mathrm{in}]$
5	for each $\tilde{\mathbf{p}}$,
6	$[\mathbf{p}_\mathrm{in}] := [\mathbf{p}_\mathrm{in}] \sqcup \{\tilde{\mathbf{p}}\}$; $[\mathbf{p}_\mathrm{out}] := [\mathbf{p}_\mathrm{out}] \sqcup \{\tilde{\mathbf{p}}\}$; (*inner inflation*);
7	$[\mathbf{p}] := [[\mathbf{p}] \backslash [\mathbf{p}_\mathrm{in}]]$; (*amputation*)
8	if $[\mathbf{p}] \neq \varnothing$; $[\mathbf{p}] := \mathcal{C}_{\mathbb{S}_\mathrm{p}}([\mathbf{p}])$; (*contraction*)
9	if $[\mathbf{p}] \neq \varnothing$;
10	if $(w([\mathbf{p}]) < \varepsilon)$ or $\left(h^0_\infty([\mathbf{p}], [\mathbf{p}_\mathrm{out}]) < \varepsilon\right)$
11	$[\mathbf{p}_\mathrm{out}] := [\mathbf{p}_\mathrm{out}] \sqcup [\mathbf{p}]$; (*outer inflation*)
12	else
13	bisect $[\mathbf{p}]$ and put the resulting boxes at the end of $\mathcal{L}$;
14	until $\mathcal{L} = \varnothing$.

interval hull of the left component of $\mathbb{S}_\mathrm{p}$ *(see Figure 5.12a). The local search of Step 4 is performed by the algorithm* CROSS*, shown at page 152 (Jaulin, 2000a). At Step 7,* $[\mathbf{p}]$ *cannot be amputated, but at Step 8,* $\mathcal{C}_{\mathbb{S}_\mathrm{p}}$ *contracts* $[\mathbf{p}]$ *directly to the solution box* $[\mathbb{S}_\mathrm{p}]$*. At this stage,* HULL *cannot conclude that* $[\mathbf{p}]$ *is equal to* $[\mathbb{S}_\mathrm{p}]$*. This is why* $[\mathbf{p}]$ *is bisected into two subboxes.* $\mathcal{L}$ *now contains the two boxes represented in Figure 5.12b. Note that the conditions (5.22) are satisfied. Then* HULL *takes the left box, fails in its local search and contracts it at Step 8 until it becomes almost equal to* $[\mathbf{p}_\mathrm{in}]$ *and* $[\mathbf{p}_\mathrm{out}]$*. Since at Step 10,* $h^0_\infty([\mathbf{p}], [\mathbf{p}_\mathrm{out}]) < \varepsilon$*,* $[\mathbf{p}_\mathrm{out}]$ *is slightly inflated. Then* HULL *goes to Step 2. Again the conditions (5.22) are satisfied.* HULL *takes the last box of* $\mathcal{L}$ *and succeeds in scanning the rightmost connected component of* $\mathbb{S}_\mathrm{p}$*. The boxes* $[\mathbf{p}_\mathrm{in}]$ *and* $[\mathbf{p}_\mathrm{out}]$ *are thus inflated and become almost equal to* $[\mathbb{S}_\mathrm{p}]$*. At Step 11,* $[\mathbf{p}_\mathrm{out}]$ *is slightly inflated by* $[\mathbf{p}]$ *and* HULL *terminates because* $\mathcal{L}$ *is empty.* ■

Remark 5.2 *When the volume of* $\mathbb{S}_\mathrm{p}$ *is too small and* $\mathbb{S}_\mathrm{p}$ *is elongated, it may become very difficult to find feasible points that allow an inner inflation. As a result,* HULL *may bisect boxes that are inside* $[\mathbb{S}_\mathrm{p}]$*. To avoid this effect, which slows down the algorithm, one could use the main feature of the approach of Section 5.4.1, which bisects only boxes with a part outside* $[\mathbb{S}_\mathrm{p}]$*. Thus, at each iteration* HULL *should compute the smallest box* $[\mathbf{p}_\mathrm{ext}]$ *that contains* $[\mathbf{p}_\mathrm{out}]$ *and the boxes of* $\mathcal{L}$*. It will then bisect only boxes of* $\mathcal{L}$ *that touch the boundary of* $[\mathbf{p}_\mathrm{ext}]$*.* ■

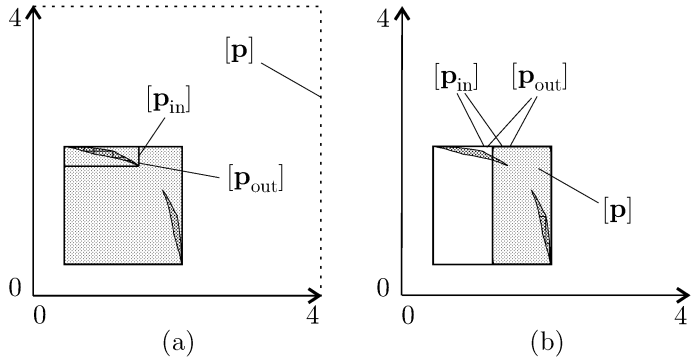


Fig. 5.12. (a) $[\mathbf{p}]$ is contracted into the grey box; local search scans the left connected component of $\mathbb{S}_{\mathrm{p}}$ only; (b) after bisection, HULL becomes able to scan the right connected component of $\mathbb{S}_{\mathrm{p}}$; when HULL terminates, $[\mathbf{p}_{\mathrm{in}}]$and $[\mathbf{p}_{\mathrm{out}}]$ are indistinguishable

5.5 Global Optimization

The problem to be considered now is the minimization of a cost function $c(\mathbf{p})$ over a compact set $\mathbb{S}_{\mathrm{p}}^{\infty} \subset \mathbb{R}^{n_{\mathrm{p}}}$:

$$\min_{\mathbf{p} \in \mathbb{S}_{\mathrm{p}}^{\infty}} c\,(\mathbf{p})\,. \tag{5.26}$$

For unconstrained minimization, $\mathbb{S}_{\mathrm{p}}^{\infty}$ will be taken as a possibly very large box $[\mathbf{p}]$ of $\mathbb{R}^{n_{\mathrm{p}}}$. For constrained minimization, the definition of $\mathbb{S}_{\mathrm{p}}^{\infty}$ will also involve equality or inequality constraints. For instance, $\mathbb{S}_{\mathrm{p}}^{\infty}$ may be defined as

$$\mathbb{S}_{\mathrm{p}}^{\infty} \triangleq \{\mathbf{p} \in \mathbb{R}^{n_{\mathrm{p}}} \mid \mathbf{h}(\mathbf{p}) \leqslant \mathbf{0} \text{ and } \mathbf{p} \in [\mathbf{p}]\}. \tag{5.27}$$

The global minimum will be denoted by $\widehat{c}$, and the set of all the corresponding global minimizers by $\widehat{\mathbb{S}}_{\mathrm{p}}$. It is always possible to transform a maximization problem into a minimization problem, for instance by multiplying $c\,(\mathbf{p})$ by -1. The most efficient interval-based optimization algorithms combine the use of contractors for $\widehat{\mathbb{S}}_{\mathrm{p}}$ and classical local search with branching algorithms. We shall explain how to get a contractor for $\widehat{\mathbb{S}}_{\mathrm{p}}$, before describing a branching algorithm that performs the minimization.

The tools developed in Chapter 4 cannot be applied directly to build a contractor $\mathcal{C}_{\widehat{\mathbb{S}}_{\mathrm{p}}}$ for $\widehat{\mathbb{S}}_{\mathrm{p}}$, because the set of all global minimizers is usually not described by non-linear equations or inequalities. Now, if $\overline{c}$ is an upper bound for the global minimum $\widehat{c}$ (for instance obtained by local minimization), then

$$\widehat{\mathbb{S}}_{\mathrm{p}} \subset \mathbb{S}_{\mathrm{p}}(\overline{c}), \tag{5.28}$$

where

$$\mathbb{S}_{\mathrm{p}}(\overline{c}) \triangleq \mathbb{P}(\overline{c}) \cap \mathbb{S}_{\mathrm{p}}^{\infty}, \tag{5.29}$$

with

$$\mathbb{P}(\overline{c}) \triangleq \{\mathbf{p} \in \mathbb{R}^{n_{\mathrm{P}}} \mid c(\mathbf{p}) \leqslant \overline{c}\}. \tag{5.30}$$

Thus, the contractor for $\mathbb{S}_{\mathrm{p}}(\overline{c})$ defined by

$$\mathcal{C}_{\mathbb{S}_{\mathrm{p}}(\overline{c})} = \mathcal{C}_{\mathbb{S}_{\mathrm{p}}^{\infty}} \cap \mathcal{C}_{\mathbb{P}(\overline{c})} \tag{5.31}$$

is also a contractor for $\widehat{\mathbb{S}}_{\mathrm{p}}$, see (4.115) and (4.117), page 98. This contractor is not thin in general, and its efficiency strongly depends on the value of $\overline{c}$. Note that if $\widetilde{c}$ is the smallest $\overline{c}$ such that $\mathbb{S}_{\mathrm{p}}(\overline{c}) \neq \varnothing$ then $\mathbb{S}_{\mathrm{p}}(\widetilde{c}) = \mathbb{S}_{\mathrm{p}}(\widehat{c}) = \widehat{\mathbb{S}}_{\mathrm{p}}$.

Remark 5.3 *Additional information could be used to increase the efficiency of the contractor for $\widehat{\mathbb{S}}_p$. If, for instance, the cost function c to be minimized is twice differentiable with respect to $\mathbf{p}$ and the minimization is unconstrained, then the set*

$$\mathbb{O} \triangleq \left\{\mathbf{p} \in \mathbb{R}^{n_p} \mid \frac{dc}{d\mathbf{p}}(\mathbf{p}) = \mathbf{0}, \frac{d^2c}{d\mathbf{p}^2}(\mathbf{p}) \geqslant \mathbf{0}\right\} \tag{5.32}$$

contains $\widehat{\mathbb{S}}_p$. A better contractor for $\widehat{\mathbb{S}}_p$ is

$$\mathcal{C}_{\widehat{\mathbb{S}}_p} \triangleq \mathcal{C}_{\mathbb{S}_p(\overline{c})} \cap \mathcal{C}_{\mathbb{O}}. \tag{5.33}$$

■

Table 5.5 presents the branching algorithm OPTIMIZE that performs the minimization. $\mathcal{Q}$ is a working list of boxes ordered by increasing value of the associated lower bound for the cost. OPTIMIZE fills a list $\mathcal{L}$ of boxes. Upon completion of the algorithm, the set $\mathbb{L}$ associated with this list contains all the global minimizers of the cost function $c(.)$ over $\mathbb{S}_{\mathrm{p}}^{\infty}$, and the interval $[\widehat{c}]$ contains the global minimum $\widehat{c}$. $[c](.)$ is an inclusion function for the cost function $c(.)$. The box of $\mathcal{Q}$ selected at Step 3 is the one associated with the smallest lower bound of the cost, which corresponds to selecting the most promising box. At Step 4, some unspecified local minimization procedure GODOWN is used to decrease the upper bound $\overline{c}$. The real number $\varepsilon > 0$ is the width beyond which boxes listed in $\mathcal{Q}$ will not be bisected. The interval $[\widehat{c}]$ bracketing the global minimum is computed at Steps 15 and 16 by interval evaluation of c over all the boxes of $\mathcal{L}$.

As an illustration, consider the situation of Figure 5.13a. An upper bound c_1 is available for $\widehat{c}$, but the contractor $\mathcal{C}_{\mathbb{S}_{\mathrm{p}}(c_1)}$ leaves $[\mathbf{p}]$ unchanged. A local search can then provide a smaller upper bound c_2 for $\widehat{c}$ (Figure 5.13b). The contractor $\mathcal{C}_{\mathbb{S}_{\mathrm{p}}(c_2)}$ can now be used to contract $[\mathbf{p}]$ with an increased efficiency, as shown in Figure 5.13c. Since $\mathcal{C}_{\mathbb{S}_{\mathrm{p}}(c_2)}$ is also a contractor for $\widehat{\mathbb{S}}_{\mathrm{p}}$, no global solution of the minimization problem can be lost.

Table 5.5. Algorithm for reliable minimization

Algorithm OPTIMIZE(in: $[\mathbf{p}], c(.), \varepsilon$; out $[\widehat{c}]$, $\mathcal{L}$)

```
1  Q := {([p], ∞)}; c̄ := ∞; [ĉ] := ∅; L := ∅;
2  repeat
3     pop first box out of Q into [p];
4     c̄ := GoDown(mid([p]), c(.));
5     remove from Q any pair ([p_i], c_i) such that c_i > c̄;
6     [p] := C_{S_p}([p]);
7     if [p] ≠ ∅ then
8        if (w([p]) < ε) then
9           put ([p], lb([c]([p]))) into L;
10       else
11          bisect [p] into [p_1] and [p_2];
12          put ([p_1], lb([c]([p_1]))) and ([p_2], lb([c]([p_2])))) into Q;
13 until Q = ∅;
14 remove from L any pair ([p_i], c_i) such that c_i > c̄;
15 for all [p] in L, [ĉ] := [ĉ] ∪ [c]([p]);
16 [ĉ] := [ĉ] ∩ ]−∞, c̄].
```

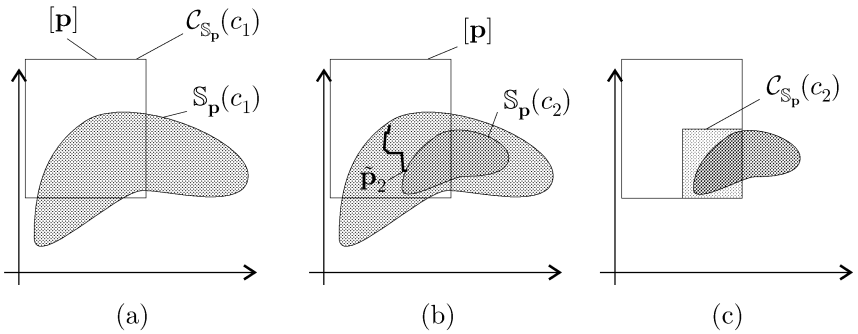


Fig. 5.13. OPTIMIZE algorithm; (a) initial configuration, (b) local minimization, (c) contraction

Remark 5.4 *Experiments (Ratschek and Rokne, 1995; Ratz and Csendes, 1995) have shown that an efficient choice for the bisection is to cut along* $\text{plane}_i([\mathbf{p}])$*, a symmetry plane such that*

$$w\left(\left[\frac{\partial c}{\partial p_i}\right]([\mathbf{p}])\right) * w([p_i]) \geqslant w\left(\left[\frac{\partial c}{\partial p_j}\right]([\mathbf{p}])\right) * w([p_j]),$$

$$\forall j \in \{1, \ldots, \dim \mathbf{p}\}.$$

■

5.5.1 The Moore–Skelboe algorithm

A simplified version of OPTIMIZE has been proposed by Skelboe (1974) and improved by Moore (1976). The resulting algorithm does not use any contractor and does not perform any local search. The next two examples (Walster et al., 1985; Moore and Ratschek, 1988; Jansson and Knüppel, 1995) illustrate the efficiency of the Moore–Skelboe algorithm and the influence of the dimension of the problem.

Example 5.4 *The Branin function*

$$c(\mathbf{p}) = (p_2 - \frac{5.1}{4\pi^2}p_1^2 + \frac{5}{\pi}p_1 - 6)^2 + 10(1 - \frac{1}{8\pi})\cos p_1 + 10 \tag{5.34}$$

admits three global minimizers over the box $[\mathbf{p}] = [-5, 10] \times [0, 15]$, *namely*

$$\widehat{\mathbf{p}}_1 = (-\pi, 12.275)^T, \quad \widehat{\mathbf{p}}_2 = (\pi, 2.275)^T, \quad \widehat{\mathbf{p}}_3 = (3\pi, 2.475)^T. \tag{5.35}$$

The corresponding global minimum is $\widehat{c} \simeq 0.397887$. *For* $\varepsilon = 10^{-5}$, *after* 422 *bisections performed in* 0.4 *s on a* PENTIUM 90, *the Moore–Skelboe algorithm finds* 18 *boxes, the union of which contains all the global minimizers;* $\widehat{c}$ *is also computed, with an accuracy of* 10 *digits. The* 18 *solution boxes can be decomposed into three groups. The interval hulls of these groups are*

$$\begin{aligned} [\widehat{\mathbf{p}}]_1 &= [-3.141594, -3.141586] \times [12.274982, 12.275012], \\ [\widehat{\mathbf{p}}]_2 &= \quad [3.141591, 3.141599] \quad \times \quad [2.274982, 2.275014], \\ [\widehat{\mathbf{p}}]_3 &= \quad [9.424769, 9.424785] \quad \times \quad [2.474977, 2.475021]. \end{aligned} \tag{5.36}$$

■

Example 5.5 *Consider the Levy 13 family of functions*

$$\begin{aligned} c_n(\mathbf{p}) = \sin^2 3\pi p_1 &+ \sum_{i=1}^{n-1}(p_i - 1)^2(1 + \sin^2 3\pi p_{i+1}) \\ &+ (p_n - 1)^2(1 + \sin^2 2\pi p_n), \end{aligned} \tag{5.37}$$

with $n \geqslant 1$. *The search box* $[\mathbf{p}]$ *is* $[-10, 10]^{\times n}$ *if* $n \leqslant 4$, *and* $[-5, 5]^{\times n}$ *if* $n > 4$. *Each function* c_n *admits the global minimum* $\widehat{c} = 0$ *and only one global minimizer* $\widehat{\mathbf{p}}$ *with all of its entries equal to one. The number of local minimizers of* c_n *grows exponentially with* n *(*900 *for* $n = 2$ *and* 10^5 *for* $n = 5$*). The performances of the Moore–Skelboe algorithm for six values of* n *with* $\varepsilon = 10^{-5}$ *are given in Table 5.6. The times indicated are for a* PENTIUM 90. ■

Table 5.6. Computing time and number of solution boxes as a function of dimension n

n	1	...	5	6	7	8	9
Computing time (s)	0.05	...	0.71	1.43	2.97	6.87	17.02
Number of solution boxes	1	...	66	147	294	547	955

5.5.2 Hansen's algorithm

Another variation around OPTIMIZE is Hansen's algorithm. This section presents some of the contractors involved (Hansen and Sengupta, 1980; Hansen, 1992b). All of them can be cast into the framework of Chapter 4. Note that this algorithm does not use any contractor based on interval constraint propagation such as $\mathcal{C}_{\downarrow\uparrow}$. Some of its special features will be presented; for more detail the reader is urged to consult Hansen (1992b), entirely devoted to the subject.

Upper-bound contractors: Assume that an upper bound $\overline{c}$ for $\widehat{c}$ is available. Contracting a box under the constraint $c(\mathbf{p}) \leqslant \overline{c}$ amounts to contracting the CSP

$$\mathcal{H} : (c(\mathbf{p}) = z,\ \mathbf{p} \in [\mathbf{p}],\ z \in]-\infty, \overline{c}]) . \tag{5.38}$$

Provided that $c(.)$ is differentiable, an external approximation of $\mathcal{H}$ (see Section 4.3, page 82) is given by

$$\mathcal{H}_1 : \begin{pmatrix} c(\mathbf{m}) + \sum_{i=1}^{n} (p_i - m_i) g_i(\boldsymbol{\xi}) = z \\ \mathbf{p} \in [\mathbf{p}],\ \boldsymbol{\xi} \in [\mathbf{p}],\ z \in]-\infty, \overline{c}] \end{pmatrix}, \tag{5.39}$$

where $g_i(\boldsymbol{\xi})$ is the ith component of the gradient of c at $\boldsymbol{\xi}$ and $\mathbf{m} = \text{mid}([\mathbf{p}])$. A subsolver for $\mathcal{H}_1$ is given by

$$p_k = m_k + \frac{1}{g_i(\boldsymbol{\xi})} \left(z - c(\mathbf{m}) - \sum_{i=1, i \neq k}^{n} (p_i - m_i)\, g_i(\boldsymbol{\xi}) \right), \tag{5.40}$$

which is used by Hansen to contract the domain for the variable p_k. Another external approximation of $\mathcal{H}$ based on the second-order Taylor expansion of c is

$$\mathcal{H}_2 : \begin{pmatrix} c(\mathbf{m}) + (\mathbf{p} - \mathbf{m})^{\mathrm{T}} \mathbf{g}(\mathbf{m}) + \dfrac{(\mathbf{p} - \mathbf{m})^{\mathrm{T}} \mathbf{H}(\boldsymbol{\xi})(\mathbf{p} - \mathbf{m})}{2} = z \\ \mathbf{p} \in [\mathbf{p}], \boldsymbol{\xi} \in [\mathbf{p}], z \in]-\infty, \overline{c}] \end{pmatrix}, \tag{5.41}$$

where $\mathbf{m} = \text{mid}([\mathbf{p}])$, $\mathbf{g}$ is the gradient vector of c, and $\mathbf{H}(\boldsymbol{\xi})$ is its Hessian matrix at $\boldsymbol{\xi}$. The first line of (5.41) is equivalent to

$$\begin{array}{l} c(\mathbf{m}) + (p_k - m_k) g_k(\mathbf{m}) \\ \quad + \sum_{i=1, i\neq k}^{n} (p_i - m_i) g_i(\mathbf{m}) + \frac{1}{2}(p_k - m_k)^2 h_{kk}(\boldsymbol{\xi}) \\ \quad + \frac{1}{2}(p_k - m_k) \sum_{i=1, i\neq k}^{n} (p_i - m_i) h_{ik}(\boldsymbol{\xi}) \\ \quad + \frac{1}{2} \sum_{i=1, i\neq k}^{n} (p_i - m_i) \sum_{j=1, j\neq k}^{i} (p_j - m_j) h_{ij}(\boldsymbol{\xi}) = z. \end{array} \tag{5.42}$$

$\mathcal{H}_2$ can thus be rewritten as

$$\mathcal{H}_2 : \left(\begin{array}{l} 1 : \alpha_k + \beta_k t_k + \gamma_k t_k^2 = 0 \\ 2 : t_k = p_k - m_k \\ 3 : \alpha_k = -z + c(\mathbf{m}) + \sum_{i=1, i\neq k}^{n} (p_i - m_i) g_i(\mathbf{m}) \\ \qquad + \frac{1}{2} \sum_{i=1, i\neq k}^{n} (p_i - m_i) \sum_{j=1, j\neq k}^{i} (p_j - m_j) h_{ij}(\boldsymbol{\xi}) \\ 4 : \beta_k = g_k(\mathbf{m}) + \frac{1}{2} \sum_{i=1, i\neq k}^{n} (p_i - m_i) h_{ik}(\boldsymbol{\xi}) \\ 5 : \gamma_k = \frac{1}{2} h_{kk}(\boldsymbol{\xi}) \\ 6 : \mathbf{p} \in [\mathbf{p}],\ \boldsymbol{\xi} \in [\mathbf{p}],\ z \in]-\infty, \overline{c}] \end{array} \right). \tag{5.43}$$

Domains for α_k, β_k and γ_k are easily obtained from Constraints 3 to 6 of (5.43). Constraint 1 can then be used to get a domain for t_k. Since this constraint is quadratic and involves only t_k, a special algorithm can be developed to get the smallest domain for t_k consistent with it (Hansen, 1992b). Constraint 2 can then be used to contract $[p_k]$.

Concavity and gradient contractors: If no constraint is involved in the minimization problem and if c is differentiable, then all the global minimizers $\widehat{\mathbf{p}}$ should satisfy $\mathbf{g}(\widehat{\mathbf{p}}) = \mathbf{0}$, where $\mathbf{g}(\mathbf{p})$ is the value of the gradient of c at $\mathbf{p}$. Moreover, if c is twice differentiable, it should be convex in all directions, including the axes of parameter space. Therefore the constraints

$$g_1(\mathbf{p}) = 0, \ldots, g_n(\mathbf{p}) = 0 \tag{5.44}$$

$$\frac{\partial^2 c}{\partial p_1^2}(\mathbf{p}) \geqslant 0, \ldots, \frac{\partial^2 c}{\partial p_n^2}(\mathbf{p}) \geqslant 0 \tag{5.45}$$

can be used to contract the current box $[\mathbf{p}]$ at Step 6 of Optimize for an unconstrained minimization. For a maximization, the sign of the inequalities in (5.45) should be reversed.

Stopping criterion:Because of Step 8, Optimize does not bisect any box that satisfies $w([\mathbf{p}]) < \varepsilon$. In Hansen's variant of Optimize, the boxes that satisfy both conditions

$$w([\mathbf{p}]) < \varepsilon_{\mathrm{p}} \quad \text{and} \quad w([c]([\mathbf{p}])) < \varepsilon_{\mathrm{c}}, \tag{5.46}$$

are never bisected. The accuracy coefficients ε_{p} and ε_{c} are chosen by the user.

Uniqueness condition: Assume again that no constraint is involved in the minimization problem. Let $[\mathbf{p}]'$ be the box obtained after one iteration of the Newton contractor applied to the box $[\mathbf{p}]$ over the gradient constraint

$\mathbf{g}(\mathbf{p}) = \mathbf{0}$. If $[\mathbf{p}]'$ is strictly included in $[\mathbf{p}]$, then there exists at most one global minimizer of c in $[\mathbf{p}]$ (Ratschek and Rokne, 1995; Wolfe, 1996). Therefore, when only one box is returned by OPTIMIZE, the uniqueness condition may prove that the global minimizer is unique. On the other hand, if several boxes are returned, it is possible that some of them do not contain any global minimizer, even if each of them satisfies the uniqueness condition.

To illustrate the efficiency of Hansen's algorithm, consider the same examples as with the Moore–Skelboe algorithm.

Example 5.6 *Consider again the Branin function (5.34) of Example 5.4. Recall that it admits three global minimizers*

$$\widehat{\mathbf{p}}_1 = (-\pi, 12.275), \quad \widehat{\mathbf{p}}_2 = (\pi, 2.275), \quad \widehat{\mathbf{p}}_3 = (3\pi, 2.475),$$

in $[\mathbf{p}]_0 = [-5, 10] \times [0, 15]$ *and that the global minimum is* $\widehat{c} \simeq 0.397887$. *For* $\varepsilon_{\mathrm{p}} = \varepsilon_{\mathrm{c}} = 10^{-5}$, *after* 32 *iterations performed in* 0.05 *s on a* PENTIUM 90, *Hansen's algorithm returns three boxes, the width of which is smaller than* 10^{-7}. *The global minimum is obtained with an accuracy of* 10^{-10}. ■

Example 5.7 *Consider again the Levy 13 family of functions of Example 5.5. For* $n \leqslant 50$ *and* $\varepsilon_{\mathrm{p}} = \varepsilon_{\mathrm{c}} = 10^{-5}$, *Hansen's algorithm produces the results of Table 5.7. Comparing Tables 5.6 and 5.7, we observe that the use of contractors makes it possible to deal with higher-dimensional problems. Note that on this example Hansen's algorithm always returns a single box that satisfies the uniqueness condition, thus proving that there exists one and only one global minimizer.* ■

Table 5.7. Computing time and number of solution boxes as a function of dimension n

n	1	...	5	...	20	50
Computing time (s)	0.05	...	0.33	...	12.5	401
Number of solution boxes	1	...	1	...	1	1

Fritz–John contractor: Assume now that there are inequality constraints of the form $\mathbf{h}(\mathbf{p}) \leqslant \mathbf{0}$ to be satisfied by the optimizers. Hansen proposes use of a contractor based on the Fritz–John conditions. Similar to the more famous Kuhn–Tucker conditions (Pardalos and Rosen, 1987), the Fritz–John conditions provide necessary conditions for a vector $\widehat{\mathbf{p}}$ to be a solution of a constrained optimization problem, as stated by the following theorem.

Theorem 5.1 *If the vector* $\widehat{\mathbf{p}} \in [\mathbf{p}] \subset \mathbb{R}^n$ *is a* local *minimizer of the cost function* c *under the constraints* $h_i(\widehat{\mathbf{p}}) \leqslant 0$ *for* $i \in \{1, \ldots, m\}$, *then there exist* $m + 1$ *real coefficients* $u_0, u_1, \ldots, u_m$ *such that*

$$\begin{aligned}
&u_0 \frac{\partial c}{\partial p_i}(\widehat{\mathbf{p}}) + u_1 \frac{\partial h_1}{\partial p_i}(\widehat{\mathbf{p}}) + \cdots + u_m \frac{\partial h_m}{\partial p_i}(\widehat{\mathbf{p}}) = 0, && i = 1, \ldots, n, \\
&u_i h_i(\widehat{\mathbf{p}}) = 0, && i = 1, \ldots, m, \\
&u_i \geqslant 0, && i = 0, \ldots, m, \\
&u_0 + u_1 + \cdots + u_m = 1.
\end{aligned} \tag{5.47}$$

The coefficients $u_0, u_1, \ldots, u_m$ *are called* Lagrange coefficients. ■

The $n + m + 1$ equations in (5.47) involve $n + m + 1$ variables (n for the components of $\mathbf{p}$ and $m + 1$ for the Lagrange coefficients). Since all the u_is should be positive and since their sum is equal to 1, the domain for each u_i can be set to $[0, 1]$. The second line of (5.47) implies that if $h_i(\widehat{\mathbf{p}}) < 0$ (which means that the constraint associated with h_i is not active at $\widehat{\mathbf{p}}$) then $u_i = 0$. The equations in (5.47) can be put in the compact form $\mathbf{f}^{\mathrm{FJ}}(\mathbf{t}) = \mathbf{0}$, where

$$\mathbf{f}^{\mathrm{FJ}}(\mathbf{t}) = \begin{pmatrix}
u_0 + u_1 + \cdots + u_m - 1 \\
u_0 \frac{\partial c}{\partial p_1}(\mathbf{p}) + u_1 \frac{\partial h_1}{\partial p_1}(\mathbf{p}) + \cdots + u_m \frac{\partial h_m}{\partial p_1}(\mathbf{p}) \\
\vdots \\
u_0 \frac{\partial c}{\partial p_n}(\mathbf{p}) + u_1 \frac{\partial h_1}{\partial p_n}(\mathbf{p}) + \cdots + u_m \frac{\partial h_m}{\partial p_n}(\mathbf{p}) \\
u_1 h_1(\mathbf{p}) \\
\vdots \\
u_m h_m(\mathbf{p})
\end{pmatrix}, \tag{5.48}$$

and

$$\mathbf{t} = \begin{pmatrix} \mathbf{p} \\ \mathbf{u} \end{pmatrix}, \tag{5.49}$$

with $\mathbf{u} = (u_0, u_1, \ldots, u_m)^{\mathrm{T}}$. $\mathbf{f}^{\mathrm{FJ}}$ is called the *Fritz–John function*. A contractor for the equation $\mathbf{f}^{\mathrm{FJ}}(\mathbf{t}) = \mathbf{0}$ can be used to contract $[\mathbf{p}]$ without losing any solution of the constrained minimization problem. The efficiency of the Fritz–John contractor, included in Hansen's algorithm, will now be illustrated with two examples.

Example 5.8 *Consider the minimization of*

$$c(\mathbf{p}) = 0.1 \left(p_1^2 + p_2^2\right) \tag{5.50}$$

over $[\mathbf{p}] = [-1, 1]^{\times 2}$ *under the constraint*

$$2 \sin(2\pi p_2) - \sin(4\pi p_1) \leqslant 0. \tag{5.51}$$

This problem is known to have 24 *local minimizers and only one global minimizer* $\widehat{\mathbf{p}} = (0, 0)^T$, *for which* $c(\widehat{\mathbf{p}}) = \widehat{c} = 0$ *(Ratschek and Rokne, 1988). With* $\varepsilon_{\mathrm{p}} = \varepsilon_{\mathrm{c}} = 10^{-5}$, *after* 25 *iterations and in* 0.11 *s on a* Pentium 90, *Hansen's algorithm returns a single box approximately given by*

$$[\mathbf{p}] = \begin{pmatrix} [-0.0000002, 0.000004] \\ [-0.0000002, 0.000004] \end{pmatrix}, \tag{5.52}$$

which contains the global minimizer. The global minimum $\widehat{c}$ *is proved to belong to* $\left[0, 10^{-10}\right]$. ■

Example 5.9 *Hansen (1992b) illustrates the performance of his algorithm on about 30 test cases, but only one of them involves constraints over* $\mathbf{p}$*, namely the minimization of*

$$c(\mathbf{p}) = 12p_1^2 - 6.3p_1^4 + p_1^6 + 6p_1p_2 + 6p_2^2 \tag{5.53}$$

over $[\mathbf{p}] = [-2, 4]^{\times 2}$ *under the constraints*

$$\begin{cases} 1 - 16p_1^2 - 25p_2^2 \leqslant 0, \\ 13p_1^3 - 145p_1 + 85p_2 - 400 \leqslant 0, \\ p_1p_2 - 4 \leqslant 0. \end{cases} \tag{5.54}$$

This problem is known to have two global minimizers:

$$\widehat{\mathbf{p}}(1) \simeq (-0.066042, 0.192895)^T \tag{5.55}$$

and

$$\widehat{\mathbf{p}}(2) \simeq (0.066042, -0.192895)^T, \tag{5.56}$$

for which $\widehat{c} \simeq 0.199035$*. For* $\varepsilon_{\mathrm{p}} = \varepsilon_{\mathrm{c}} = 10^{-5}$*, after* 44 *iterations and in* 0.22 *s on a* Pentium 90*, Hansen's algorithm generates two boxes that bracket these two global minimizers with a precision equal to* 10^{-10}*. The global minimum* $\widehat{c}$ *is bracketed with a precision of* 10^{-9}*. For the two boxes found in parameter space, the last two constraints are inactive so* $\widehat{u}_2 = \widehat{u}_3 = 0$*. The other two Lagrange coefficients are* $\widehat{u}_0 \simeq 0.834087$ *and* $\widehat{u}_1 \simeq 0.165913$*.* ■

5.5.3 Using interval constraint propagation

Optimize is also used in the literature with contractors based on interval constraint propagation (ICP) such as $\mathcal{C}_{\downarrow\uparrow}$ presented in Chapter 4 (see, for instance, Zhou, 1996 and van Hentenryck et al., 1997). Strangely enough, the contractors based on ICP never seem to have been combined with those employed by Hansen. When the optimization problem is non-linear and the cost function is not differentiable, the contractors based on linear approximations (*i.e.*, all those presented in Chapter 4, except $\mathcal{C}_{\downarrow\uparrow}$) are in general inefficient, and only contractors based on ICP are able to contract large domains. This is illustrated by the following example (Jaulin, 2001b).

Example 5.10 *Consider the minimization of*

$$c\left(\mathbf{p}\right) = \max_{k \in \{1,\ldots,10\}} \left| g\left(\mathbf{p}, k\right) \right|, \tag{5.57}$$

where

$$g(\mathbf{p},k) = p_1 \exp(\frac{p_2}{4}k^2) + p_3 \exp(\frac{p_4}{4}k^2) - 20\exp(-0.2\ k^2) + 10\exp(-0.05\ k^2) + 0.1\sin(\frac{k^2}{4}), \tag{5.58}$$

and $\mathbf{p} = (p_1, p_2, p_3, p_4)^T$ *belongs to the box*

$$[\mathbf{p}] = [-60, 60] \times [-1, 0] \times [-60, 60] \times [-1, 0]. \tag{5.59}$$

A permutation of p_1 *with* p_3 *and of* p_2 *with* p_4 *leaves* $c(\mathbf{p})$ *unchanged. The solution set is thus symmetric with respect to the plane* ($p_1 - p_3 = 0,\ p_2 - p_4 = 0$)*. This problem is ill conditioned and non-differentiable, and classical punctual local methods have difficulties finding even a local minimizer (Jaulin, 2001b). Moreover, they cannot detect that the problem has two global minimizers. Equipped with the single contractor* $\mathcal{C}_{\downarrow\uparrow}$ *associated with the constraint* $(c(\mathbf{p}) \leqslant \overline{c})$*, on a* PENTIUM 133, OPTIMIZE *finds in* 1.7 *s and after* 109 *bisections that the global minimum lies inside* $[0.0653, 0.0657]$*. The list* $\mathcal{L}$ *returned by* OPTIMIZE *consists of* 44 *tiny boxes, each of which is such that its image by the cost function is included in* $[0.0653, 0.0657]$*. These boxes can be classified into two symmetrical groups, associated with each of the two global minimizers.* ■

5.6 Minimax Optimization

Consider now the difficult problem (Du, 1995) of getting an enclosure of the real number c_n defined by the following sequence of optimization problems:

$$\begin{array}{lcl}
c_1(\mathbf{p}_2,\ldots,\mathbf{p}_n) & = & \max_{\mathbf{p}_1\in[\mathbf{p}_1]}\ c_0(\mathbf{p}_1,\ldots,\mathbf{p}_n), \\
 & & \text{subject to } (\mathbf{p}_1,\ldots,\mathbf{p}_n)\in\mathbb{S}(1), \\
c_2(\mathbf{p}_3,\ldots,\mathbf{p}_n) & = & \min_{\mathbf{p}_2\in[\mathbf{p}_2]}\ c_1(\mathbf{p}_2,\ldots,\mathbf{p}_n), \\
 & & \text{subject to } (\mathbf{p}_2,\ldots,\mathbf{p}_n)\in\mathbb{S}(2), \\
\vdots & \vdots & \vdots \\
c_i(\mathbf{p}_{i+1},\ldots,\mathbf{p}_n) & = & (-1)^i \min_{\mathbf{p}_i\in[\mathbf{p}_i]}\ (-1)^i\, c_{i-1}(\mathbf{p}_i,\ldots,\mathbf{p}_n), \\
 & & \text{subject to } (\mathbf{p}_i,\ldots,\mathbf{p}_n)\in\mathbb{S}(i), \\
\vdots & \vdots & \vdots \\
c_{n-1}(\mathbf{p}_n) & = & (-1)^{n-1} \min_{\mathbf{p}_{n-1}\in[\mathbf{p}_{n-1}]}\ (-1)^{n-1}\, c_{n-2}(\mathbf{p}_{n-1},\mathbf{p}_n), \\
 & & \text{subject to } (\mathbf{p}_{n-1},\mathbf{p}_n)\in\mathbb{S}(n-1), \\
c_n & = & (-1)^n \min_{\mathbf{p}_n\in[\mathbf{p}_n]}\ (-1)^n\, c_{n-1}(\mathbf{p}_n), \\
 & & \text{subject to } \mathbf{p}_n\in\mathbb{S}(n),
\end{array} \tag{5.60}$$

where $c_0(\mathbf{p}_1, \ldots, \mathbf{p}_n)$ is a known function of the n vectors $\mathbf{p}_1, \ldots, \mathbf{p}_n$. The set $\mathbb{S}(i)$ is associated with the constraints of the ith optimization problem. When i is even, $(-1)^i = 1$ and thus

$$c_i(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n) = \min_{\mathbf{p}_i \in [\mathbf{p}_i]} c_{i-1}(\mathbf{p}_i, \ldots, \mathbf{p}_n), \quad \text{subject to } (\mathbf{p}_i, \ldots, \mathbf{p}_n) \in \mathbb{S}(i). \tag{5.61}$$

When i is odd, $(-1)^i = -1$ and thus

$$c_i(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n) = \max_{\mathbf{p}_i \in [\mathbf{p}_i]} c_{i-1}(\mathbf{p}_i, \ldots, \mathbf{p}_n), \quad \text{subject to } (\mathbf{p}_i, \ldots, \mathbf{p}_n) \in \mathbb{S}(i). \tag{5.62}$$

An example of such a minimax problem is the guaranteed evaluation of

$$c_3 = \underbrace{\underbrace{\min_{p_3 \in [-1,2]} \max_{p_2 \in [-1,1]} \min_{p_1 \in [0,10]} p_1 (p_2 + p_3)}_{c_0(p_1, p_2, p_3)}}_{c_1(p_2, p_3)} \Bigg\}_{c_2(p_3)} .$$
$$\sin(p_3) \geqslant 0 \qquad p_3^2 + p_2 \geqslant 2 \qquad p_1^2 + p_2 p_3 \geqslant 1$$

For simplicity, it will be assumed that $c_i(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n)$ exists for all $i \in \{0, \ldots, n-1\}$ and for all $\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n$. This assumption, satisfied in the applications considered in this book, implies that for any $(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n)$, there exists at least one $\mathbf{p}_i$ such that $(\mathbf{p}_i, \ldots, \mathbf{p}_n) \in \mathbb{S}(i)$.

Although many application problems can be cast into this form, as illustrated in Chapter 8, the minimax problem has received very little attention in the interval community (Zuhe et al., 1990; Didrit, 1997; Wolfe, 1999; Jaulin, 2001b). Section 5.6.1 is devoted to the unconstrained case and shows how a convergent inclusion function for c_i can be built when a convergent inclusion function for c_{i-1} is available. The constrained case is considered in Section 5.6.2. Section 5.6.3 is devoted to the closely related problem of dealing with quantifiers.

5.6.1 Unconstrained case

Assume that $\mathbb{S}(i) = \mathbb{R}^{\dim \mathbb{S}(i)}$ for $i = 0, \ldots, n-1$. For the time being, take i to be even, so

$$c_i(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n) = \min_{\mathbf{p}_i \in [\mathbf{p}_i]} c_{i-1}(\mathbf{p}_i, \ldots, \mathbf{p}_n). \tag{5.63}$$

We shall now explain how to get an inclusion function for $c_i(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n)$ based on an inclusion function for $c_{i-1}(\mathbf{p}_i, \ldots, \mathbf{p}_n)$. Provided that an inclusion

function is available for $c_0(\mathbf{p}_1, \ldots, \mathbf{p}_n)$, we shall thus obtain an inclusion function for each function c_i, including c_n as defined by (5.60).

Denote $(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n)$ by $\mathbf{q}_{i+1}$. Equation 5.63 then becomes

$$c_i(\mathbf{q}_{i+1}) = \min_{\mathbf{p}_i \in [\mathbf{p}_i]} c_{i-1}(\mathbf{p}_i, \mathbf{q}_{i+1}). \tag{5.64}$$

The method to be proposed is based on the following theorem.

Theorem 5.2 *If* $[c_{i-1}]([\mathbf{p}_i], [\mathbf{q}_{i+1}])$ *is a convergent inclusion function for* $c_{i-1}(\mathbf{p}_i, \mathbf{q}_{i+1})$*, and if* $[\mathbf{p}_i](k)$*,* $k \in \{1, \ldots, \overline{k}\}$ *is a partition of* $[\mathbf{p}_i]$ *such that*

$$\forall k \in \{1, \ldots, \overline{k}\}, w([\mathbf{p}_i](k)) \leqslant w([\mathbf{q}_{i+1}]), \tag{5.65}$$

then

$$[c_i]([\mathbf{q}_{i+1}]) \triangleq \min_{k \in \{1,\ldots,\overline{k}\}} [c_{i-1}]([\mathbf{p}_i](k), [\mathbf{q}_{i+1}]) \tag{5.66}$$

is a convergent inclusion function for $c_i(\mathbf{q}_{i+1})$. ■

Proof. The first part of the proof establishes that $[c_i]$ as defined by (5.66) is an inclusion function for c_i. The second part proves that $[c_i]$ is convergent. Define

$$c_i^{(k)}(\mathbf{q}_{i+1}) \triangleq \min_{\mathbf{p}_i \in [\mathbf{p}_i](k)} c_{i-1}(\mathbf{p}_i, \mathbf{q}_{i+1}). \tag{5.67}$$

From (5.63), and since the $[\mathbf{p}_i](k)$s form a partition of $[\mathbf{p}_i]$,

$$c_i(\mathbf{q}_{i+1}) = \min_{k \in \{1,\ldots,\overline{k}\}} c_i^{(k)}(\mathbf{q}_{i+1}). \tag{5.68}$$

Now, from (5.67),

$$c_i^{(k)}(\mathbf{q}_{i+1}) \in c_{i-1}([\mathbf{p}_i](k), \mathbf{q}_{i+1}), \tag{5.69}$$

which is a subset of $[c_{i-1}]([\mathbf{p}_i](k), [\mathbf{q}_{i+1}])$, if $\mathbf{q}_{i+1} \in [\mathbf{q}_{i+1}]$. Therefore, from (5.68),

$$c_i([\mathbf{q}_{i+1}]) \subset \min_{k \in \{1,\ldots,\overline{k}\}} [c_{i-1}]([\mathbf{p}_i](k), [\mathbf{q}_{i+1}]). \tag{5.70}$$

To prove that $[c_i]$ is convergent, assume that the width of $[\mathbf{q}_{i+1}]$ tends to zero. From (5.65) the widths of the $[\mathbf{p}_i](k)$s also tend to zero, and so does the width of $[c_{i-1}]([\mathbf{p}_i](k), [\mathbf{q}_{i+1}])$ for all $k \in \{1, \ldots, \overline{k}\}$. Therefore, the width of $[c_i]([\mathbf{q}_{i+1}])$ as computed by (5.66) tends to zero. ■

For a given punctual vector $\mathbf{q}_{i+1}$, Figure 5.14a gives an interpretation of $c_i(\mathbf{q}_{i+1})$. Figure 5.14b depicts the interval function $c_{i-1}(\mathbf{p}_i, [\mathbf{q}_{i+1}])$. The inclusion function (5.66) obtained by partitioning $[\mathbf{p}_i]$ is illustrated by Figure 5.14c.

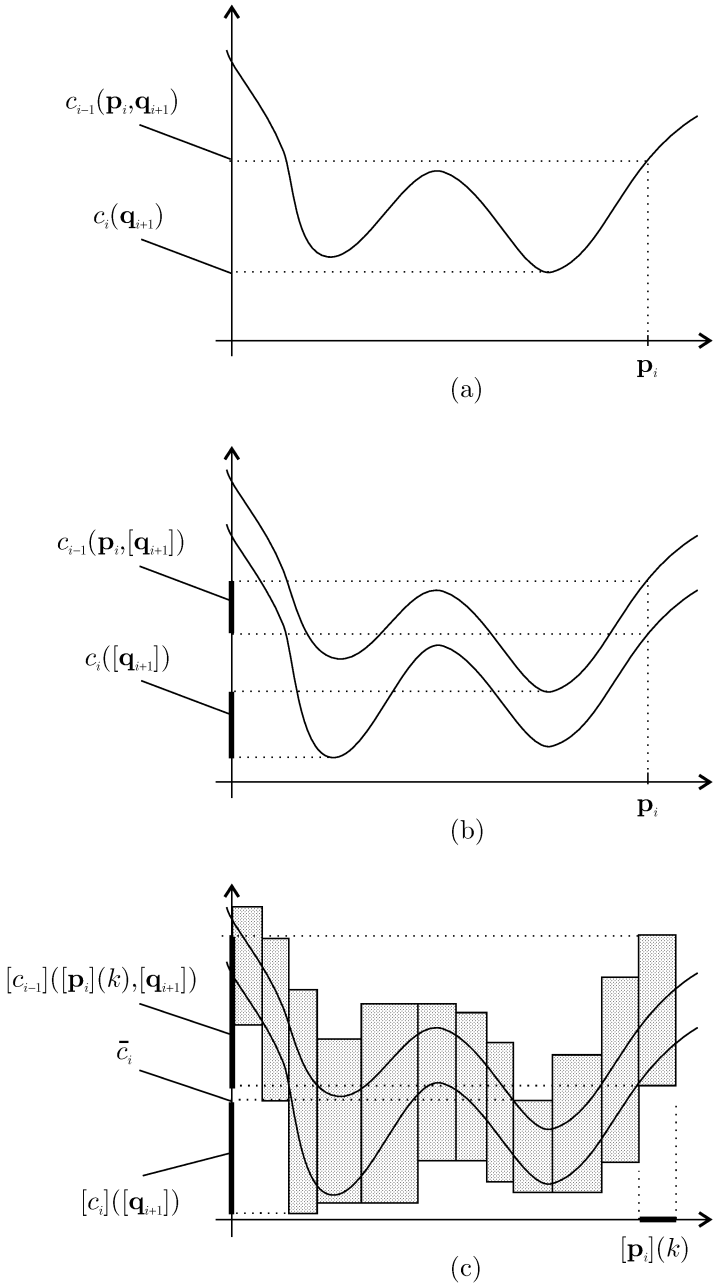


Fig. 5.14. Enclosing $c_i([\mathbf{q}_{i+1}])$ in the minimax algorithm (unconstrained case); $\mathbf{q}_{i+1}$ stands for $(\mathbf{p}_{i+1},\ldots,\mathbf{p}_n)$; (a) definition of c_i from c_{i-1} for punctual arguments; (b) evaluation of c_i from c_{i-1} when all arguments are boxes except $\mathbf{p}_i$; (c) interval enclosure of c_i and c_{i-1} when all their arguments are boxes

Remark 5.5 *The* min *operator in (5.66) should be interpreted as an interval extension of the* min *operator on real numbers. For instance*

$$\begin{aligned}&\min([1,6],[3,5],[0,10],[-1,7],[9,9])\\&\quad=[\min(1,3,0,-1,9),\min(6,5,10,7,9)]=[-1,5].\end{aligned}\tag{5.71}$$

■

Remark 5.6 *It is trivial to extend Theorem 5.2 to deal also with the case where i is odd. If a convergent inclusion function is available for c_0, a recursive application of Theorem 5.2 then yields inclusion functions for all the functions c_i.* ■

The algorithm of Table 5.8, adapted from Didrit (1997), is a recursive implementation of the inclusion function for $c_i(\mathbf{q}_{i+1})$ based on Theorem 5.2. It is assumed that an inclusion function for c_0 is available, and the presentation is first for even i. The positive coefficient ε represents the width of the smallest box that can still be bisected to update the partition of $[\mathbf{p}_i]$ so that this partition satisfies (5.65). Since i is assumed to be even, an upper bound $\overline{c}_i$ for $c_i([\mathbf{q}_{i+1}])$ is given by

$$\min_{k\in\{1,\ldots,\overline{k}\}}\left(\mathrm{ub}\left([c_{i-1}]\left([\mathbf{p}_i](k),[\mathbf{q}_{i+1}]\right)\right)\right),\tag{5.72}$$

see Figure 5.14c. This upper bound is used to eliminate any $[\mathbf{p}_i](k)$ that satisfies

$$\mathrm{lb}\left([c_{i-1}]\left([\mathbf{p}_i](k),[\mathbf{q}_{i+1}]\right)\right)>\overline{c}_i\tag{5.73}$$

(see Step 7). In Figure 5.14c, the leftmost and rightmost boxes satisfy this condition and can thus be eliminated. This could be avoided in principle, but it is a useful simplification to eliminate boxes without bisecting them down to the width ε whenever possible. The first call of the algorithm computes $[c_n](\varepsilon)$ and the deepest call (in the sense of recursivity) evaluates

$$[c_0]([\mathbf{p}_1],[\mathbf{q}_2])=[c_0]([\mathbf{p}_1],\ldots,[\mathbf{p}_n]).\tag{5.74}$$

Remark 5.7 *In Table 5.8, $[\mathbf{p}_i]$ stands for the current box $[\mathbf{p}_i](k)$ of the partition.* ■

Remark 5.8 *No contractors are used in this simple version, but they should be involved to improve efficiency.* ■

The following lines should replace their counterparts of Table 5.8 when i is odd:

```
3'   c_i := -∞; c̄_i := -∞;
                          ...
7'      if (ub([r]) ⩾ c_i) then
8'         c_i = max(lb([r]), c_i);
9'         if (w([p_i]) < ε) then c̄_i = max(c̄_i,ub([r]));
```

Table 5.8. Inclusion function for the unconstrained minimax problem

Algorithm $[c_i](\text{in: } [\mathbf{p}_{i+1}], \ldots, [\mathbf{p}_n], \varepsilon; \text{ out: } [c_i])$ // i assumed to be even	
1	if $i = 0$ then return $[c_0]([\mathbf{p}_1], \ldots, [\mathbf{p}_n])$;
2	$\mathcal{Q} = \{[\mathbf{p}_)]\}$;
3	$\underline{c}_i := \infty$; $\overline{c}_i := \infty$;
4	do
5	pop first element out of $\mathcal{Q}$ into $[\mathbf{p}_i]$;
6	compute $[r] := [c_{i-1}]([\mathbf{p}_i], \ldots, [\mathbf{p}_n], w([\mathbf{p}_i]))$;
7	if $(\text{lb}([r]) \leqslant \overline{c}_i)$ then
8	$\overline{c}_i := \min(\text{ub}([r]), \overline{c}_i)$;
9	if $(w([\mathbf{p}_i]) < \varepsilon)$ then $\underline{c}_i := \min(\underline{c}_i, \text{lb}([r]))$;
10	else bisect $[\mathbf{p}_i]$ and put the resulting boxes at the end of $\mathcal{Q}$;
11	while $\mathcal{Q} \neq \varnothing$;
12	$[c_i] := [\underline{c}_i, \overline{c}_i]$.

The algorithm computing a guaranteed enclosure of c_n is then simply

Algorithm MINIMAX(in: $c_0, [\mathbf{p}_1], \ldots, [\mathbf{p}_n], \varepsilon$; out: $[c_n]$)	
1	$[c_n] := [c_n](\varepsilon)$.

5.6.2 Constrained case

Assume again that i is even, so

$$c_i(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n) = \min_{\mathbf{p}_i \in [\mathbf{p}_i]} c_{i-1}(\mathbf{p}_i, \ldots, \mathbf{p}_n), \quad \text{subject to } (\mathbf{p}_i, \ldots, \mathbf{p}_n) \in \mathbb{S}(i). \tag{5.75}$$

Denote again $(\mathbf{p}_{i+1}, \ldots, \mathbf{p}_n)$ by $\mathbf{q}_{i+1}$. Equation 5.75 then becomes

$$c_i(\mathbf{q}_{i+1}) = \min_{\mathbf{p}_i \in [\mathbf{p}_i]} c_{i-1}(\mathbf{p}_i, \mathbf{q}_{i+1}), \quad \text{subject to } (\mathbf{p}_i, \mathbf{q}_{i+1}) \in \mathbb{S}(i). \tag{5.76}$$

In the constrained case, Theorem 5.2 is replaced by the following one, which will also serve to obtain inclusion functions for each function c_i, including c_n.

Theorem 5.3 *Assume that i is even. If $[c_{i-1}]([\mathbf{p}_i], [\mathbf{q}_{i+1}])$ is an inclusion function for $c_{i-1}(\mathbf{p}_i, \mathbf{q}_{i+1})$, if $\{[\mathbf{p}_i](k),\ k \in \{1, \ldots, \overline{k}\}\}$ is a partition of $[\mathbf{p}_i]$ and if $[t_{\mathbb{S}(i)}]$ is an inclusion test for the set $\mathbb{S}(i)$ (*i.e.*, $[t_{\mathbb{S}(i)}]([\mathbf{p}_i], [\mathbf{q}_{i+1}]) = 1 \Rightarrow ([\mathbf{p}_i], [\mathbf{q}_{i+1}]) \subset \mathbb{S}(i)$ and $[t_{\mathbb{S}(i)}]([\mathbf{p}_i], [\mathbf{q}_{i+1}]) = 0 \Rightarrow ([\mathbf{p}_i], [\mathbf{q}_{i+1}]) \cap \mathbb{S}(i) = \varnothing$), then a lower bound for $c_i([\mathbf{q}_{i+1}])$ is*

$$\underline{c}_i\left([\mathbf{q}_{i+1}]\right) = \min_{\substack{k \in \{1, \ldots, \overline{k}\} \\ [t_{\mathbb{S}(i)}]([\mathbf{p}_i], [\mathbf{q}_{i+1}]) \neq 0}} \text{lb}\left([c_{i-1}]([\mathbf{p}_i](k), [\mathbf{q}_{i+1}])\right) \tag{5.77}$$

and an upper bound for $c_i\left([\mathbf{q}_{i+1}]\right)$ *is*

$$\overline{c}_i\left([\mathbf{q}_{i+1}]\right) = \min_{\substack{k \in \{1,\ldots,\overline{k}\} \\ [t_{\mathbb{S}(i)}](\mathbf{m}_i(k),[\mathbf{q}_{i+1}])=1}} \mathrm{ub}\left([c_{i-1}]([\mathbf{p}_i](k),[\mathbf{q}_{i+1}])\right), \quad (5.78)$$

where $\mathbf{m}_i(k) = \mathrm{mid}([\mathbf{p}_i](k))$. ■

Proof. The proof for the lower bound is trivial and only a proof for the upper bound is given. If $[t_{\mathbb{S}(i)}](\mathbf{m}_i(k),[\mathbf{q}_{i+1}]) = 1$, this implies that for any $\mathbf{q}_{i+1} \in [\mathbf{q}_{i+1}]$, $(\mathbf{m}_i(k),\mathbf{q}_{i+1}) \in \mathbb{S}(i)$. Then, according to (5.75),

$$\forall \mathbf{q}_{i+1} \in [\mathbf{q}_{i+1}],\ c_i(\mathbf{q}_{i+1}) \leqslant c_{i-1}(\mathbf{m}_i(k),\mathbf{q}_{i+1}). \quad (5.79)$$

The upper bound $\overline{c}_i(k)$ for $[c_{i-1}]([\mathbf{p}_i](k),[\mathbf{q}_{i+1}])$ is also an upper bound for $[c_{i-1}](\mathbf{m}_i(k),[\mathbf{q}_{i+1}])$ and thus an upper bound for $c_i([\mathbf{q}_{i+1}])$. The smallest of these $\overline{k}$ upper bounds as given by (5.78) is also an upper bound for $c_i([\mathbf{q}_{i+1}])$. ■

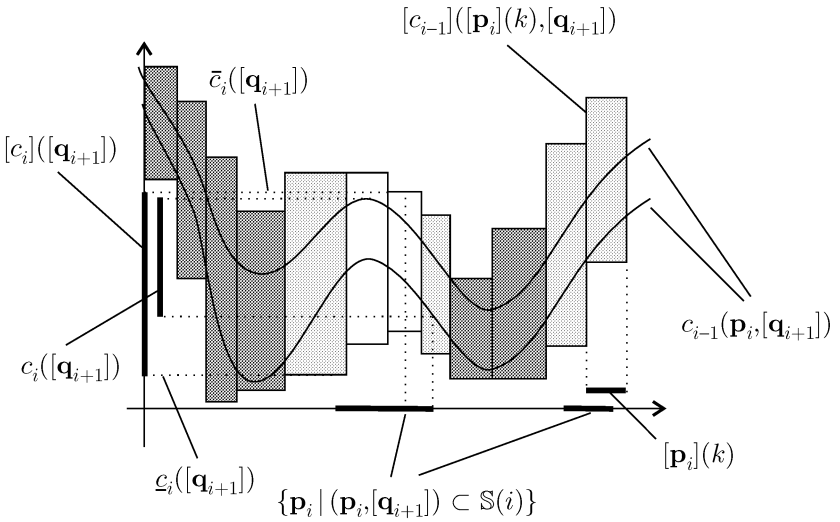


Fig. 5.15. Illustration of Theorem 5.3 used for constrained minimax optimization

Figure 5.15 illustrates this theorem. Boxes in dark grey correspond to $[\mathbf{p}_i](k)$s such that $[t_{\mathbb{S}(i)}]([\mathbf{p}_i](k),[\mathbf{q}_{i+1}]) = 0$. These boxes are not taken into account to get the bounds $\underline{c}_i\left([\mathbf{q}_{i+1}]\right)$ and $\overline{c}_i\left([\mathbf{q}_{i+1}]\right)$. Boxes in light grey are associated with $[\mathbf{p}_i](k)$s for which $[t_{\mathbb{S}(i)}]([\mathbf{p}_i](k),[\mathbf{q}_{i+1}]) = [0,1]$. They should be considered for the computation of $\underline{c}_i\left([\mathbf{q}_{i+1}]\right)$. White boxes satisfy $[t_{\mathbb{S}(i)}]\left(\mathrm{mid}\left([\mathbf{p}_i]\left(k\right)\right),[\mathbf{q}_{i+1}]\right) = 1$. They are involved in the computation of $\underline{c}_i\left([\mathbf{q}_{i+1}]\right)$ and $\overline{c}_i\left([\mathbf{q}_{i+1}]\right)$.

Table 5.9 implements the inclusion function provided by Theorem 5.3. It requires an inclusion function for c_0 and an inclusion test for $\mathbb{S}(i)$. If $([\mathbf{p}_i], \ldots, [\mathbf{p}_n])$ turns out to be outside $\mathbb{S}(i)$, then the current box $[\mathbf{p}_i]$ is removed (see Step 6). The current upper bound $\overline{c}_i$ is updated at Step 10 in order to allow the evaluation of (5.78) and the possible elimination of boxes (see Step 8). The current lower bound $\underline{c}_i$ is updated at Step 11 in order to allow the evaluation of (5.77). Recall that in the presentation of the algorithm it was assumed that i is even. Adaptation to deal with odd i is trivial. Note that Remarks 5.7 and 5.8 still apply.

The algorithm MINIMAX computing a guaranteed enclosure of c_n is the same as in the unconstrained case.

Table 5.9. Inclusion function for the constrained minimax problem

Algorithm $[c_i]$(in: $[\mathbf{p}_{i+1}], \ldots, [\mathbf{p}_n], \varepsilon$; out: $[c_i]$) // i assumed to be even	
1	if $i = 0$ then return $[c_0]([\mathbf{p}_1], \ldots, [\mathbf{p}_n])$;
2	$\mathcal{Q} = \{[\mathbf{p}_\diamond]\}$;
3	$\underline{c}_i := \infty$; $\overline{c}_i := \infty$;
4	do
5	pop first element out of $\mathcal{Q}$ into $[\mathbf{p}_i]$;
6	if $[t_{\mathbb{S}(i)}]([\mathbf{p}_i], \ldots, [\mathbf{p}_n]) \neq 0$ then
7	compute $[r] := [c_{i-1}]([\mathbf{p}_i], \ldots, [\mathbf{p}_n], w([\mathbf{p}_i]))$;
8	if $(\mathrm{lb}([r]) \leqslant \overline{c}_i)$ then
9	if $[t_{\mathbb{S}(i)}](\mathrm{mid}([\mathbf{p}_i]), [\mathbf{p}_{i+1}], \ldots, [\mathbf{p}_n]) = 1$ then
10	$\overline{c}_i := \min(\mathrm{ub}([r]), \overline{c}_i)$;
11	if $(w([\mathbf{p}_i]) < \varepsilon)$ then $\underline{c}_i := \min(\underline{c}_i, \mathrm{lb}([r]))$;
12	else bisect $[\mathbf{p}_i]$ and put the resulting boxes at the end of $\mathcal{Q}$;
13	while $\mathcal{Q} \neq \varnothing$;
14	$[c_i] := [\underline{c}_i, \overline{c}_i]$.

5.6.3 Dealing with quantifiers

This section deals with inequality problems involving the existential quantifier $\exists$ and universal quantifier $\forall$. Surveys of the available methods can be found in (Mishra, 1993; Caviness and Johnson, 1998; Dorato, 2000), and applications in (Ioakimidis, 1997; El Kahoui and Weber, 2000). This class of problems is particularly important in control theory (Jaulin and Walter, 1996; Liska and Steinberg, 1996; Steinberg and Liska, 1996; **?**; Hong et al., 1997; Jirstrand, 1997; Neubacher, 1997). Some examples related to robust control will be considered in Chapter 7. Interval analysis provides very promising tools to solve such problems (Jaulin and Walter, 1996; Benhamou and Goualard, 2000;

Ratschan, 2000a, 2000b). This section shows that problems involving $\exists$ and $\forall$ are closely related to minimax problems and that the algorithm MINIMAX can be used to solve them. For instance, proving that

$$\forall p_3 \in [1,3], \exists p_2 \in [1,2], \forall p_1 \in [0,1], p_1 + p_2 p_3 \leqslant 1 \tag{5.80}$$

amounts to proving that

$$\max_{p_3 \in [1,3]} \min_{p_2 \in [1,2]} \max_{p_1 \in [0,1]} p_1 + p_2 p_3 \leqslant 1. \tag{5.81}$$

More generally, proving that

$$\left(\begin{array}{l} \forall \mathbf{p}_3 \in [\mathbf{p}_3],\ \mathbf{p}_3 \in \mathbb{S}(3) \\ \exists \mathbf{p}_2 \in [\mathbf{p}_2],\ (\mathbf{p}_2, \mathbf{p}_3) \in \mathbb{S}(2) \\ \forall \mathbf{p}_1 \in [\mathbf{p}_1],\ (\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \in \mathbb{S}(1) \end{array} \right) \text{ such that } c_0(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \geqslant 0 \tag{5.82}$$

can be cast into proving that

$$\begin{array}{ccccc} \min & \max & & \min & \\ \mathbf{p}_3 \in [\mathbf{p}_3] & \mathbf{p}_2 \in [\mathbf{p}_2] & & \mathbf{p}_1 \in [\mathbf{p}_1] & c_0(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \geqslant 0. \\ \mathbf{p}_3 \in \mathbb{S}(3) & (\mathbf{p}_2, \mathbf{p}_3) \in \mathbb{S}(2) & & (\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \in \mathbb{S}(1) & \end{array} \tag{5.83}$$

MINIMAX can thus be used to prove (or disprove) (5.82). Of course, it should be slightly modified to terminate as soon as the current enclosure for the global optimum c_3 has a positive lower bound (or a negative upper bound).

Consider now the problem of characterizing a set $\mathbb{S}$ defined by inequalities involving $\forall$ and $\exists$. For simplicity, it will be assumed that

$$\mathbb{S} = \left\{ \mathbf{p}_3 \in [\mathbf{p}_3]\,(0) \ \mid\ (\forall \mathbf{p}_2 \in [\mathbf{p}_2],\ \exists \mathbf{p}_1 \in [\mathbf{p}_1] \mid g(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \geqslant 0) \right\}, \tag{5.84}$$

but more general sets could be considered as well. $\mathbb{S}$ can be defined equivalently by

$$\mathbb{S} = \left\{ \mathbf{p}_3 \in [\mathbf{p}_3](0) \mid \min_{\mathbf{p}_2 \in [\mathbf{p}_2]} \max_{\mathbf{p}_1 \in [\mathbf{p}_1]} g(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \geqslant 0 \right\}. \tag{5.85}$$

A subbox $[\mathbf{p}_3]$ of $[\mathbf{p}_3](0)$ is inside $\mathbb{S}$ if

$$\min_{\mathbf{p}_3 \in [\mathbf{p}_3]} \min_{\mathbf{p}_2 \in [\mathbf{p}_2]} \max_{\mathbf{p}_1 \in [\mathbf{p}_1]} g(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \geqslant 0, \tag{5.86}$$

and outside $\mathbb{S}$ if

$$\max_{\mathbf{p}_3 \in [\mathbf{p}_3]} \min_{\mathbf{p}_2 \in [\mathbf{p}_2]} \max_{\mathbf{p}_1 \in [\mathbf{p}_1]} g(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) < 0. \tag{5.87}$$

Define the interval function

$$[h]([\mathbf{p}_3]) = \left[\min_{\mathbf{p}_3 \in [\mathbf{p}_3]} \min_{\mathbf{p}_2 \in [\mathbf{p}_2]} \max_{\mathbf{p}_1 \in [\mathbf{p}_1]} g(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3), \right. \\ \left. \max_{\mathbf{p}_3 \in [\mathbf{p}_3]} \min_{\mathbf{p}_2 \in [\mathbf{p}_2]} \max_{\mathbf{p}_1 \in [\mathbf{p}_1]} g(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \right]. \tag{5.88}$$

An enclosure $[h_e]([\mathbf{p}_3])$ of $[h]([\mathbf{p}_3])$ can be obtained by running Minimax for the lower and upper bounds. The accuracy parameter in Minimax could be taken as $\varepsilon = w([\mathbf{p}_3])$ as suggested in Jaulin and Walter (1996). An inclusion test for $\mathbb{S}$ is then

$$\begin{cases} [t]([\mathbf{p}_3]) = 1 & \text{if } \mathrm{lb}([h_e]([\mathbf{p}_3])) \geqslant 0, \\ [t]([\mathbf{p}_3]) = 0 & \text{if } \mathrm{ub}([h_e]([\mathbf{p}_3])) < 0, \\ [t]([\mathbf{p}_3]) = [0,1] & \text{otherwise.} \end{cases} \tag{5.89}$$

Sivia (see Table 3.2, page 58) can now be used to get inner and outer approximations of $\mathbb{S}$ by subpavings.

A possible application is the characterization of the projection of a set defined by non-linear inequalities onto a subspace, for instance, the projection of the three-dimensional sphere

$$\mathbb{O} = \left\{ (x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 \leqslant 1 \right\} \tag{5.90}$$

onto the (x, y)-plane, which is defined by

$$\mathbb{S} = \{ (x, y) \in \mathbb{R}^2 \mid \exists z,\ x^2 + y^2 + z^2 \leqslant 1 \}. \tag{5.91}$$

The accumulation set of the algorithm is here the set of all the points of $\mathbb{R}^3$ that are projected onto the boundary $\partial\mathbb{S}$ of $\mathbb{S}$. It has a dimension equal to that of $\partial\mathbb{S}$, *i.e.,* one. On the other hand, if the characterization of $\mathbb{S}$ were performed by running Sivia to characterize $\mathbb{O}$ and then by projecting the resulting boxes onto the (x, y)-plane, the accumulation set would be the boundary of $\mathbb{O}$ (which has a dimension equal to two instead of one for $\partial\mathbb{S}$), so complexity would be higher. Moreover this approach would not be able to provide an inner approximation of $\mathbb{S}$, contrary to the one advocated here.

5.7 Cost Contours

Consider the cost function $c : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{p} \mapsto c(\mathbf{p})$, for which an inclusion function $[c]$ is assumed available. The problem to be addressed now is the characterization of the m *level sets* $\mathbb{L}_1, \ldots, \mathbb{L}_m$ associated with m given values $c_1, \ldots, c_m$ of the cost, defined by

$$\mathbb{L}_i \triangleq \{ \mathbf{p} \in [\mathbf{p}] \mid c(\mathbf{p}) = c_i \} \quad i = 1, \ldots, m, \tag{5.92}$$

where $[\mathbf{p}]$ is the box of interest. Table 5.10 presents the algorithm ISOCRIT (for iso-criterion) (Didrit et al., 1997) performing this characterization. Upon completion of ISOCRIT, the union of the boxes listed in $\mathcal{L}$ contains the m $\mathbb{L}_i$s.

The structure of ISOCRIT is similar to that of SIVIA, presented in Chapter 3. At Step 4, the smallest interval containing the intersection of $[c]([\mathbf{p}])$ and $\{c_1, \ldots, c_m\}$ is computed. A contraction is performed at Step 5, which may lead to the empty set if there is no c_i in $[c]([\mathbf{p}])$. In such a case, $[\mathbf{p}]$ does not intersect any level set of interest and is eliminated. At Steps 7 and 8, before storing $[\mathbf{p}]$ into the list $\mathcal{L}$ it is also required that $[h]$, computed at Step 4, be a degenerate interval. When this condition occurs, $[\mathbf{p}]$ is a small box, that intersects one level set of interest at most. The value of the cost associated with this level set is stored with $[\mathbf{p}]$ in $\mathcal{L}$ for further treatment.

Table 5.10. Algorithm for characterizing level sets

	Algorithm ISOCRIT(in: $c(.), c_1, \ldots, c_m, [\mathbf{p}], \varepsilon$; out: $\mathcal{L}$)
1	$\mathcal{Q} := \{[\mathbf{p}]\}$; $\mathcal{L} := \varnothing$;
2	do
3	pop first box out of $\mathcal{Q}$ into $[\mathbf{p}]$;
4	$[h] := [[c]([\mathbf{p}]) \cap \{c_1, \ldots, c_m\}]$;
5	contract $[\mathbf{p}]$ under the constraint $[c]([\mathbf{p}]) \in [h]$;
6	if $[h] \neq \varnothing$,
7	if $w([\mathbf{p}]) < \varepsilon$ and if $[h]$ is punctual, then
8	put the pair $([\mathbf{p}], [h])$ into $\mathcal{L}$;
9	else
10	bisect $[\mathbf{p}]$ and put the resulting boxes in $\mathcal{Q}$;
11	while $\mathcal{Q} \neq \varnothing$.

Example 5.11 *For the Branin function of Example 5.4, page 120, and Example 5.6, page 123, for* $c_1 = 1$, $c_2 = 10$, $c_3 = 50$, $c_4 = 150$ *and* $\varepsilon = 0.2$, ISOCRIT *yields the paving of Figure 5.16 after 6951 iterations and* 0.8 *s on a* PENTIUM 90. *The figure suggests the existence of at least three local minimizers.* ■

5.8 Conclusions

This chapter has presented solvers to treat difficult non-linear problems in a guaranteed way. The notions of inclusion test and of set contractor allowed us to focus attention on the management of search space rather than on the technical tools to be employed (interval analysis, consistency techniques and CSPs).

Interval analysis is present in these solvers through the notions of contractor and inclusion test. This is why the literature often calls them *interval solvers*.

This chapter concludes Part II, devoted to interval tools. It has provided algorithms to solve problems that are the bread and butter of engineering applications, such as finding all the solutions of systems of equations and inequalities and optimizing cost functions of various types under various constraints.

These tools will be applied in Part III to typical engineering problems. Chapter 6 deals with the estimation of unknown quantities of interest from uncertain experimental data, Chapter 7 is devoted to the control of uncertain systems and Chapter 8 to problems of robotics. Every effort will be made to make the problems treated understandable by readers with other applications in mind.

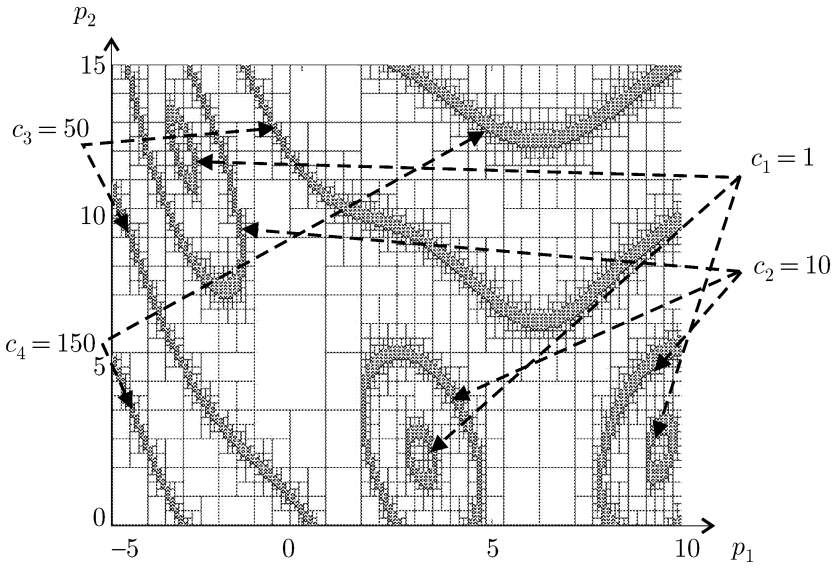


Fig. 5.16. Paving obtained by ISOCRIT for the Branin function

Part III

Applications

6. Estimation

6.1 Introduction

Consider a set $\mathcal{X}$ consisting of real variables $x_1, \ldots, x_n$, which form a vector $\mathbf{x}$. For the sake of simplicity, $n = \dim \mathbf{x}$ will be taken as finite. A variable in $\mathcal{X}$ may, for instance, represent

- the time at which a given event occurs (or the value taken by any other independent variable),
- the value of some physical parameter, such as the rate constant of a chemical reaction,
- the value taken by some quantity of interest at a given instant of time.

Each of the variables in $\mathbf{x}$ will be assumed to be partially or totally unknown, and it is the purpose of estimation to use all the available data to obtain more accurate information about the numerical values of all or some of them, see, e.g., Walter and Pronzato (1997). When these quantities are assumed to be constant, one often speaks of *parameter identification*. Assume that relations can be defined between variables, based on physical laws and on hypotheses about the system under consideration. Examples of such relations are $x_1 \leqslant 3x_2$ and $x_3 = \sin(x_1 + x_2)$. These relations define a subset $\mathbb{M}$ of $\mathbb{R}^n$, called the *constrained set*, containing all the $\mathbf{x}$s that satisfy all of them. The letter $\mathbb{M}$ has been chosen as a reminder of the fact that the constrained set is a *model* of reality. Note that if $z(t)$ is a time-dependent quantity of interest, the function z is not considered as a variable, but its values at all time instants of interest form as many variables. Thus, $z(1), z(5)$ and $z(10)$ may be variables. Note also that any quantity that can be assumed to be known exactly need not be incorporated in $\mathcal{X}$.

Remark 6.1 *Differential relations, such as* $\ddot{y}(t) + 3\dot{y}(t) - \sin t = 0$ *cannot be incorporated as such, since the number* n *of variables is taken as finite. The only continuous-time systems to be considered in this chapter are thus those for which explicit solutions can be calculated.* ■

Remark 6.2 *With the approach presented in this chapter, there is no need to distinguish between input variables, which are known and more or less under control, and output variables, which are observed on the system as it reacts*

L. Jaulin et al., *Applied Interval Analysis*

to inputs and perturbations. This approach is thus well suited to behavioural modelling as advocated by Willems (1986a, 1986b), where no such distinction is made. When convenient, the distinction remains possible, of course. ■

Example 6.1 *Consider a system with input $u(t)$ and output $y(t)$, where t is time. Assume that*

$$\forall t \in \mathbb{R}^+,\ y(t) = u(t) + \exp(-pt), \tag{6.1}$$

where p is some unknown scalar parameter. Assume that two measurements $y(t_1)$ and $y(t_2)$ of the output are collected at times t_1 and t_2. The set of all variables of interest may then be

$$\mathcal{X} = \{t_1, t_2, u_1, u_2, p, y_1, y_2\}, \tag{6.2}$$

where u_1, u_2, y_1 and y_2 represent $u(t_1), u(t_2), y(t_1)$ and $y(t_2)$. The constrained set is then the set $\mathbb{M}$ of all $(t_1, t_2, u_1, u_2, p, y_1, y_2)$ such that $y_1 = u_1 + e^{-pt_1}$ and $y_2 = u_2 + e^{-pt_2}$. ■

Independently of the relations defining $\mathbb{M}$, assume that information is available about the values that $\mathbf{x}$ may take. This information may come from measurements and from prior knowledge about their reliability. Two expressions of this information will be considered.

The first one is by a real function $\check{c}$ in the $\mathbf{x}$-space, to be used as a measure of the feasibility of $\mathbf{x}$, which will be called the *prior feasibility function*. By convention, the value of $\check{c}(\mathbf{x})$ will decrease when the feasibility of $\mathbf{x}$ increases. The prior feasibility function may have been obtained by a maximum-likelihood approach based on statistical information about the nature of measurement noise, but many other interpretations could be thought of (e.g., fuzzy logic).

The second expression of prior information is by a subset $\check{\mathbb{X}}$ of $\mathbb{R}^n$ containing all feasible values for $\mathbf{x}$, which will be called the *prior feasible set*. This set may have been derived from the technical data sheets of sensors, which usually contain information about the maximum error committed in any given range of operation.

Example 6.2 *Consider again Example 6.1, and assume that approximate values for all the variables have been obtained, given by*

$$\check{\mathbf{x}} = (\check{t}_1, \check{t}_2, \check{u}_1, \check{u}_2, \check{p}, \check{y}_1, \check{y}_2). \tag{6.3}$$

A possible prior feasibility function is

$$\begin{aligned} \check{c}(t_1, t_2, u_1, u_2, p, y_1, y_2) = {} & w_{t_1} \left(t_1 - \check{t}_1\right)^2 + w_{t_2} \left(t_2 - \check{t}_2\right)^2 \\ & + w_{u_1} \left(u_1 - \check{u}_1\right)^2 + w_{u_2} \left(u_2 - \check{u}_2\right)^2 \\ & + w_p \left(p - \check{p}\right)^2 \\ & + w_{y_1} \left(y_1 - \check{y}_1\right)^2 + w_{y_2} \left(y_2 - \check{y}_2\right)^2, \end{aligned} \tag{6.4}$$

where w_{x_i} is a positive coefficient expressing the confidence associated with the approximate value $\check{x}_i$. When no $\check{x}_i$ is available, one may take $w_{x_i} = 0$. Note that the minimum of $\check{c}$ is reached at $\check{\mathbf{x}} = (\check{t}_1, \check{t}_2, \check{u}_1, \check{u}_2, \check{p}, \check{y}_1, \check{y}_2)$ and that the associated value of $\check{c}$ is zero. One may alternatively decide to represent prior knowledge on $\mathbf{x}$ by a prior feasible set $\check{\mathbb{X}}$, containing $\check{\mathbf{x}}$ and large enough to include all values of $\mathbf{x}$ that are deemed acceptable. Here, the prior feasible set might be defined as

$$\begin{aligned}\check{\mathbb{X}} &= \{\mathbf{x} = (t_1, t_2, u_1, u_2, p, y_1, y_2) \mid t_1 \in [\check{t}_1], \ldots, y_2 \in [\check{y}_2]\} \\ &= [\check{t}_1] \times [\check{t}_2] \times [\check{u}_1] \times [\check{u}_2] \times [\check{p}] \times [\check{y}_1] \times [\check{y}_2], \end{aligned} \tag{6.5}$$

where $[\check{t}_1], \ldots, [\check{y}_2]$ are prior intervals assumed to contain the true values for the corresponding variables. ■

Estimation will be viewed as the action of taking the constrained set $\mathbb{M}$ into account to make the information available on $\mathbf{x}$ more accurate. Two approaches will be considered. The first one, described in Section 6.2, is based on the prior feasibility function $\check{c}(\mathbf{x})$. The posterior (set) estimate $\widehat{\mathbb{X}}_{\mathrm{c}}$ is then defined as the set of all global minimizers of this prior feasibility function over the constrained set $\mathbb{M}$:

$$\widehat{\mathbb{X}}_{\mathrm{c}} = \arg\min_{\mathbf{x} \in \mathbb{M}} \check{c}(\mathbf{x}). \tag{6.6}$$

The second approach, presented in Sections 6.3 and 6.4, aims to characterize the *posterior feasible set* $\widehat{\mathbb{X}}_{\mathrm{s}}$, defined as

$$\widehat{\mathbb{X}}_{\mathrm{s}} = \check{\mathbb{X}} \cap \mathbb{M}. \tag{6.7}$$

For both approaches, a projection of the posterior estimates onto the space of the variables of interest should be performed.

Remark 6.3 *Here* prior *means before taking the constrained set into account, whereas usually in statistics it means before taking the data into account.* ■

Example 6.3 *Consider again Example 6.2. With the first approach, the posterior estimate is*

$$\begin{aligned}\widehat{\mathbb{X}}_{\mathrm{c}} = \quad & \arg\min \quad \check{c}(t_1, t_2, u_1, u_2, p, y_1, y_2). \\ & y_1 = u_1 \exp(-pt_1) \\ & y_2 = u_2 \exp(-pt_2) \end{aligned} \tag{6.8}$$

It contains the best values for $\mathbf{x}$ (in the sense of $\check{c}$) among those in $\mathbb{M}$. With the second approach, the posterior feasible set might be

$$\begin{aligned}\widehat{\mathbb{X}}_{\mathrm{s}} = \{&\mathbf{x} = (t_1, t_2, u_1, u_2, p, y_1, y_2) \mid \check{c}(\mathbf{x}) \leqslant \delta, \\ &y_1 = u_1 \exp(-pt_1) \text{ and } y_2 = u_2 \exp(-pt_2)\}, \end{aligned} \tag{6.9}$$

where δ is some prespecified positive real number. If one is only interested in the value of p, then one should project $\widehat{\mathbb{X}}_{\mathrm{c}}$ or $\widehat{\mathbb{X}}_{\mathrm{s}}$ onto parameter space. ■

6.2 Parameter Estimation Via Optimization

Assume that the set $\mathcal{X}$ of all variables can be partitioned into two sets $\mathcal{Z}$ and $\mathcal{Y}$, where $\mathcal{Z} = \{z_1, \ldots, z_{n_z}\}$ and $\mathcal{Y} = \{y_1, \ldots, y_{n_y}\}$, such that there exists a function $\boldsymbol{\phi} : \mathbb{R}^{n_z} \to \mathbb{R}^{n_y}$ for which the following equivalence holds

$$(z_1, \ldots, z_{n_z}, y_1, \ldots, y_{n_y})^{\mathrm{T}} \in \mathbb{M} \Leftrightarrow (y_1, \ldots, y_{n_y})^{\mathrm{T}} = \boldsymbol{\phi}(z_1, \ldots, z_{n_z}). \tag{6.10}$$

By an abuse of notation, we shall feel free to write, more concisely,

$$(\mathbf{z}, \mathbf{y}) \in \mathbb{M} \Leftrightarrow \mathbf{y} = \boldsymbol{\phi}(\mathbf{z}). \tag{6.11}$$

Define the *posterior feasibility function* as

$$\hat{c}(\mathbf{z}) \triangleq \check{c}(\mathbf{z}, \boldsymbol{\phi}(\mathbf{z})). \tag{6.12}$$

Solving (6.6) amounts to the unconstrained minimization of $\hat{c}(\mathbf{z})$. Except in important special cases such as when the posterior feasibility function $\hat{c}$ is quadratic in $\mathbf{z}$, $\hat{c}(\mathbf{z})$ is not convex and the usual local iterative algorithms of non-linear programming may get trapped in local minima. Global guaranteed methods such as Optimize (page 119), should therefore be preferred whenever applicable.

Example 6.4 *Consider again Example 6.1, and take* $\mathcal{Z} = \{t_1, t_2, u_1, u_2, p\}$ *and* $\mathcal{Y} = \{y_1, y_2\}$ *so*

$$\boldsymbol{\phi}(\mathbf{z}) = \begin{pmatrix} u_1 \exp(-pt_1) \\ u_2 \exp(-pt_2) \end{pmatrix}. \tag{6.13}$$

Solving (6.8) amounts to finding the set of all the global minimizers of

$$\hat{c}(\mathbf{z}) = \check{c}(t_1, t_2, u_1, u_2, p, u_1 \exp(-pt_1), u_2 \exp(-pt_2)). \tag{6.14}$$

Note that if t_1, t_2, u_1 *and* u_2 *were assumed to be known exactly, they should not appear in* $\mathbf{z}$*, and optimization would be with respect to the parameter* p *only. The situation considered here is much more general, since the values taken by the input and output, as well as the time instants at which they are measured, may be uncertain and may need to be estimated.* ■

In what follows, the prior feasibility function $\check{c}(\mathbf{x})$ will characterize the distance between $\mathbf{x}$ and some given vector $\check{\mathbf{x}}$ of prior estimates for the variables. A weighted L_2 norm

$$\check{c}(\mathbf{x}) = ||\mathbf{x} - \check{\mathbf{x}}||_2^2 = \sum_{i=1}^{n} w_{x_i}(x_i - \check{x}_i)^2, \tag{6.15}$$

will be used in Section 6.2.1 and a weighted L_∞ norm

$$\check{c}(\mathbf{x}) = ||\mathbf{x} - \check{\mathbf{x}}||_\infty = \max_{i=1}^{n} w_{x_i}|x_i - \check{x}_i|, \tag{6.16}$$

in Section 6.2.2. In both cases, w_{x_i} is a prespecified positive weight expressing the degree of confidence in the prior estimate $\check{x}_i$. These two norms may lead to totally different posterior estimates $\widehat{\mathbf{x}}$, as illustrated by Figure 6.1. Geometrically, $\widehat{\mathbf{x}}$ is the projection of $\check{\mathbf{x}}$ onto the constrained set $\mathbb{M}$, as defined for the norm employed.

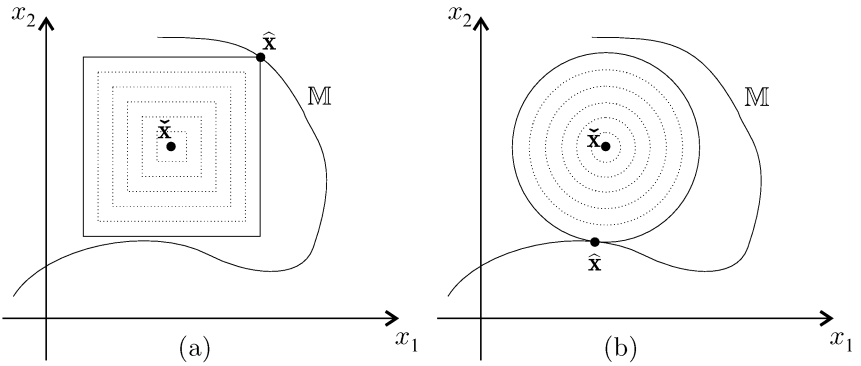


Fig. 6.1. The posterior estimate $\hat{\mathbf{x}}$ is here very sensitive to the norm employed; (a) L_∞-norm, (b) L_2-norm

Remark 6.4 *If nothing is known about the variable x_i, w_{x_i} may be taken equal to* 0, *in which case (6.15) and (6.16) no longer correspond to norms.* ■

6.2.1 Least-square parameter estimation in compartmental modelling

Compartmental models are widely used in biology and pharmacology to study metabolisms and the fate of drugs (Jacquez, 1972). They also find applications in ecology and chemical engineering (Happel, 1986). A compartmental model consists of a finite set of homogeneous reservoirs, called compartments and represented by circles, which may exchange material as indicated by arrows. The evolution of the quantity of material in each of the compartments is described by a set of first-order ordinary differential equations, usually assumed to be linear and time-invariant, with the flow of material leaving Compartment i proportional to the quantity q_i of material in this compartment. The equations describing the behaviour of the compartmental model are obtained by writing down conservation equations, under the form of a state equation. As in Kieffer and Walter (1998), consider for example the system described by Figure 6.2.

The evolution of the vector $\mathbf{q} = (q_1, q_2)^{\mathrm{T}}$ of the quantities of material in the two compartments is described by the linear time-invariant state equation

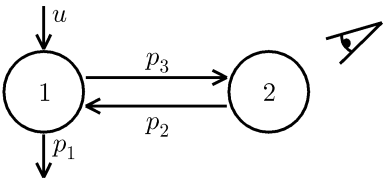


Fig. 6.2. Two-compartment model

$$\begin{cases} \dot{q}_1 = -\left(p_3 + p_1\right) q_1 + p_2 q_2 + u, \\ \dot{q}_2 = p_3 q_1 - p_2 q_2. \end{cases} \tag{6.17}$$

Take the system in zero initial condition ($\mathbf{q}(0_-) = \mathbf{0}$), and assume that a Dirac input $u(t) = \delta(t)$ is applied to Compartment 1, so $q_1(0_+) = 1$ and $q_2(0_+) = 0$. Assume also that the content of Compartment 2 is observed at 16 instants of time, according to

$$y_i = q_2\left(t_i\right), \quad i = 1, ..., 16. \tag{6.18}$$

It is trivial to show that

$$y_i = \alpha\left(\exp(-\lambda_1 t_i) - \exp(-\lambda_2 t_i)\right), \quad i = 1, ..., 16, \tag{6.19}$$

where

$$\alpha = \frac{p_3}{\sqrt{\left(p_1 - p_2 + p_3\right)^2 + 4 p_2 p_3}}, \tag{6.20}$$

$$\lambda_1 = \frac{p_1 + p_2 + p_3 - \sqrt{\left(p_1 - p_2 + p_3\right)^2 + 4 p_2 p_3}}{2} \tag{6.21}$$

and

$$\lambda_2 = \frac{p_1 + p_2 + p_3 + \sqrt{\left(p_1 - p_2 + p_3\right)^2 + 4 p_2 p_3}}{2}. \tag{6.22}$$

These equations define the constrained set $\mathbb{M}$. Assume further that the measurement times t_i are known exactly, and thus need not be considered as variables. The variables of the problem are then $\mathbf{p} = (p_1, p_2, p_3)^{\mathrm{T}}$, which takes the role of $\mathbf{z}$, and $\mathbf{y} = (y_1, \ldots, y_{16})^{\mathrm{T}}$. Prior values $\check{y}_i$ of the variables y_i have been obtained as a result of the measurements performed on the system under study, and are given in Table 6.1.

No prior information is available about $\mathbf{p}$, and the measurements $\check{y}_i$ are all deemed equally reliable, so the prior feasibility function is chosen as

$$\check{c}(\mathbf{p}, \mathbf{y}) = \sum_{i=1}^{16} \left(\check{y}_i - y_i\right)^2, \tag{6.23}$$

which does not depend on $\mathbf{p}$. The associated minimization problem is

$$\min_{(\mathbf{p},\mathbf{y})\in\mathbb{M}} \sum_{i=1}^{16} \left(\check{y}_i - y_i\right)^2 . \tag{6.24}$$

Now, $(\mathbf{p},\mathbf{y}) \in \mathbb{M}$ is equivalent to

$$\forall i \in \{1,\ldots,16\}, y_i = \phi_i(\mathbf{p}), \tag{6.25}$$

where $\phi_i(\mathbf{p})$ is computed according to (6.19)–(6.22). Minimizing the prior feasibility function defined by (6.23) under the constraint $(\mathbf{p},\mathbf{y}) \in \mathbb{M}$ thus amounts to the unconstrained minimization of the posterior feasibility function

$$\hat{c}(\mathbf{p}) = \sum_{i=1}^{16} \left(\check{y}_i - \phi_i(\mathbf{p})\right)^2 . \tag{6.26}$$

Table 6.1. Experimental data

t_i	1	2	3	4	5	6	7	8
$\check{y}_i$	0.0532	0.0478	0.0410	0.0328	0.0323	0.0148	0.0216	0.0127

t_i	9	10	11	12	13	14	15	16
$\check{y}_i$	0.0099	0.0081	0.0065	0.0043	0.0013	0.0015	0.0060	0.0126

For a search box in parameter space taken as $[0.01, 2.0] \times [0.05, 3.0] \times [0.05, 3.0]$, and with the precision parameters ε_{p} and ε_{c} both equal to 10^{-9}, Hansen's algorithm for unconstrained optimization (page 121) encloses the two global minimizers of this cost function in the two boxes

$$\begin{aligned} &[1.925402, 1.925404] \times [0.232717, 0.232719] \times [0.145075, 0.145077] \\ &[0.232717, 0.232719] \times [1.925402, 1.925404] \times [0.145075, 0.145077]. \end{aligned} \tag{6.27}$$

Notice that these boxes can be deduced from one another by exchanging their interval values for p_1 and p_2, whereas p_3 takes the same interval value in both boxes. This is consistent with the conclusion of an identifiability study (Walter and Pronzato, 1997), which indicates that the system of Figure 6.2 is only locally identifiable, and that p_1 and p_2 can be interchanged without modifying the input–output relation, whereas p_3 is uniquely identifiable from the experimental data. It seems important to stress that the conclusion of the present estimation was not based on such a prior identifiability study, which can be dispensed with when global tools are used as here. Two compartmental systems representative of the solutions in (6.27) are drawn on Figure 6.3.

Figure 6.4 presents the data $\check{y}(t_i)$ and the fitted response $\hat{y}(t)$ associated with the global minimizers.

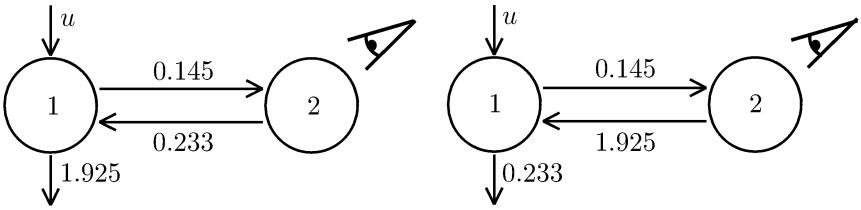


Fig. 6.3. Two radically different compartmental models with the same observed behaviour

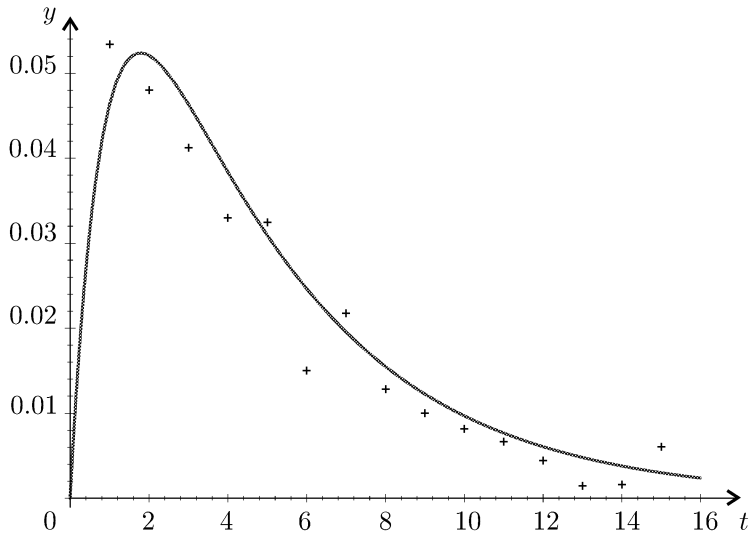


Fig. 6.4. Data $\check{y}(t_i)$ (+) and estimated model output $\hat{y}(t)$ (curve)

Remark 6.5 *When optimization is with respect to* $\mathbf{p}$*, Hansen's algorithm needs about one day on a* PENTIUM 233 *to reach this conclusion. By first optimizing with respect to* α, λ_1 *and* λ_2 *and then solving (6.20) to (6.22) for* $\mathbf{p}$ *by* SIVIAX*, page 104, it is possible to cut down computing time to about one minute (Kieffer and Walter, 1998).* ■

6.2.2 Minimax parameter estimation

As in Section 6.2.1, consider a system for which n_y measurements $\check{y}_1, \ldots, \check{y}_{n_y}$, associated with the output variables $y_1, \ldots, y_{n_y}$, have been collected at known instants of time t_i, $i = 1, \ldots, n_y$. The $\check{y}_i$s thus correspond to approximate values for the unknown variables y_i that would have been collected in ideal conditions. Assume that the system depends on an unknown parameter vector $\mathbf{p} \in \mathbb{R}^{n_p}$, to be estimated. The variables of the estimation problem are then

$$\mathbf{x} = (p_1, \ldots, p_{n_\mathrm{p}}, y_1, \ldots, y_{n_\mathrm{y}})^\mathrm{T}. \tag{6.28}$$

Following (6.16), the prior feasibility function will be taken as

$$\check{c}(\mathbf{p}, \mathbf{y}) = \check{c}(\mathbf{y}) = ||\check{\mathbf{y}} - \mathbf{y}||_\infty = \max_{i \in \{1,\ldots,n_\mathrm{y}\}} |\check{y}_i - y_i|. \tag{6.29}$$

Since nothing is known about $\mathbf{p}$, we have chosen $w_{p_i} = 0$ and thus $\check{c}(\mathbf{p}, \mathbf{y})$ does not depend on $\mathbf{p}$; see Remark 6.4. Assume that the relations defining the constrained set are given by

$$y_i = \phi(\mathbf{p}, t_i), \ i = 1, \ldots, n_\mathrm{y}, \tag{6.30}$$

where $\phi(\mathbf{p}, t)$ is some prespecified function. From (6.12), the posterior feasibility function to be minimized is

$$\hat{c}(\mathbf{p}) = \check{c}(\phi(\mathbf{p}, t_1), \ldots, \phi(\mathbf{p}, t_{n_\mathrm{y}})) = \max_{i \in \{1,\ldots,n_\mathrm{y}\}} |\check{y}_i - \phi(\mathbf{p}, t_i)|. \tag{6.31}$$

Estimating $\mathbf{p}$ then amounts to computing the global minimizers of

$$\hat{c}(\mathbf{p}) = \max_{i \in \{1,\ldots,n_\mathrm{y}\}} |f_i(\mathbf{p})|, \tag{6.32}$$

with $f_i(\mathbf{p}) = \check{y}_i - \phi(\mathbf{p}, t_i)$. The resulting minimax optimization problem is known as a *discrete Chebyshev problem*. This type of problem also appears in *sensor fusion* (McKendall, 1990; McKendall and Mintz, 1992) or in *decision theory* (Berger, 1985), when one should minimize the maximum probability of unacceptable error or risk.

Since $\hat{c}(\mathbf{p})$ is not differentiable everywhere, traditional gradient-type methods are usually *very* inefficient, besides having all the well-known limitations attached to local methods. Most existing methods are also essentially local, and based on the iterative application of linear or quadratic programming techniques. Interval solvers have been used in Wolfe (1999) for one-dimensional problems (dim $\mathbf{p} = 1$), and in Zuhe et al. (1990) and Jaulin (2001b) for a more general case. Optimize (page 119) can be used, as illustrated on the following example. (It is not appropriate to resort to Minimax of Section 5.6, as the maximization is with respect to a finite number of times.)

Example 6.5 *Assume that the variables $p_1, \ldots, p_4, y_1, \ldots, y_{10}$ are related by*

$$y_i = p_1 \exp(p_2 t_i) + p_3 \exp(p_4 t_i), \ i = 1, \ldots, 10. \tag{6.33}$$

The p_is and y_is form the vectors $\mathbf{p}$ and $\mathbf{y}$. Since a permutation of p_1 with p_3 and of p_2 with p_4 does not affect the validity of the relations, $\mathbf{p}$ is not identifiable uniquely. Any reliable parameter estimation method should therefore lead to symmetrical solutions, provided that the search domain is sufficiently large to contain all of them. Assume that the data of Table 6.2 have been collected at known instants of time t_i. These data are displayed on Figure 6.5. The prior feasibility function is

$$\check{c}(\mathbf{p},\mathbf{y}) = \check{c}(\mathbf{y}) = \max_{i=1,\dots,10} |y_i - \check{y}_i|. \tag{6.34}$$

Again it does not depend on $\mathbf{p}$*, because no prior information is available about it. The constrained set is*

$$\mathbb{M} = \{(\mathbf{p},\mathbf{y}) \mid y_i = p_1 \exp(p_2 t_i) + p_3 \exp(p_4 t_i),\ i = 1,\dots,10\}, \tag{6.35}$$

and the posterior feasibility function is

$$\hat{c}(\mathbf{p}) = \max_{i=1,\dots,10} |\check{y}_i - \phi(\mathbf{p}, t_i)|. \tag{6.36}$$

The minimization of this function will be performed in Example 6.7. ■

Table 6.2. Experimental data

t_i	0.25	1	2.25	4	6.25
$\check{y}_i$	6.9465	0.8902	−3.0562	−3.7537	−2.8262

t_i	9	12.25	16	20.25	25
$\check{y}_i$	−1.6660	−0.7961	−0.3086	−0.1330	−0.1218

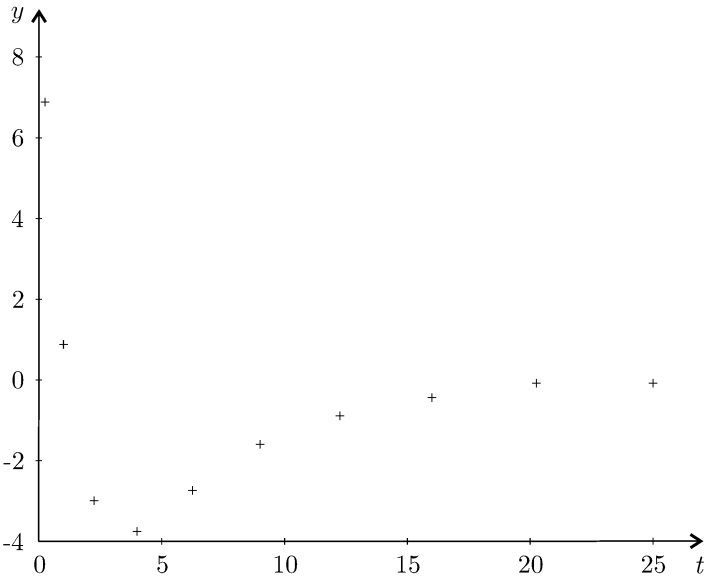


Fig. 6.5. Data of the minimax estimation example

OPTIMIZE (page 119) can be used to find $\hat{\mathbf{p}}$ that minimizes $\hat{c}$ as defined by (6.32). At Step 6 of this algorithm, a contractor $\mathcal{C}([\mathbf{p}])$ associated with the constraint

$$\max_{i\in\{1,\ldots,n_\mathrm{y}\}} |f_i(\mathbf{p})| \leqslant \overline{c} \tag{6.37}$$

should be provided, where $f_i(\mathbf{p}) = \breve{y}_i - \phi(\mathbf{p}, t_i)$ and $\overline{c}$ is an upper bound for $\hat{c}(\hat{\mathbf{p}})$. $\mathcal{C}([\mathbf{p}])$ can be obtained by contracting the CSP

$$\mathcal{H}: \begin{pmatrix} f_1(\mathbf{p}) = c_1 \\ \vdots \\ f_{n_\mathrm{y}}(\mathbf{p}) = c_{n_\mathrm{y}} \\ p_1 \in [p_1], \ldots, p_{n_\mathrm{p}} \in [p_{n_\mathrm{p}}] \\ c_1 \in [-\overline{c}, \overline{c}], \ldots, c_{n_\mathrm{y}} \in [-\overline{c}, \overline{c}] \end{pmatrix}. \tag{6.38}$$

For the local search required at Step 4 of OPTIMIZE to decrease the upper bound $\overline{c}$, the approach of Jaulin (2001b) will be used, which assumes that each parameter p_i appears only once in the expression for $f_j(\mathbf{p})$. This approach is fast, does not need the evaluation of gradients or subgradients of the cost function, is easy to implement and illustrates the ability of interval methods to deal with local optimization.

For a given $\mathbf{p}$, a given index i of axis in parameter space and a given upper bound $\overline{c}$ for $\hat{c}(\hat{\mathbf{p}})$ (if no better $\overline{c}$ is available, take $\overline{c} = c(\mathbf{p})$), define the following sets:

$$\begin{aligned} \mathbb{L}_i(\mathbf{p}) &\triangleq \{\mathbf{q} \in \mathbb{R}^{n_\mathrm{p}} \mid \forall \ell \in \{1, \ldots, i-1, i+1, \ldots, n_\mathrm{p}\},\ p_\ell = q_\ell\}, \\ \mathbb{S}_k(\overline{c}) &\triangleq \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid f_k(\mathbf{p})| \leqslant \overline{c}\}, \\ \mathbb{S}(\overline{c}) &\triangleq \textstyle\bigcap_{k=1}^{n_\mathrm{y}} \mathbb{S}_k(\overline{c}) = \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid \hat{c}(\mathbf{p}) \leqslant \overline{c}\}, \\ \mathbb{Q}_i(\mathbf{p}, \overline{c}) &\triangleq \mathbb{L}_i(\mathbf{p}) \cap \mathbb{S}(\overline{c}) \\ &= (\mathbb{L}_i(\mathbf{p}) \cap \mathbb{S}_1(\overline{c})) \cap \cdots \cap \left(\mathbb{L}_i(\mathbf{p}) \cap \mathbb{S}_{n_\mathrm{y}}(\overline{c})\right). \end{aligned} \tag{6.39}$$

A representation of the sets $\mathbb{L}_i(\mathbf{p})$, $\mathbb{S}(\overline{c})$ and $\mathbb{Q}_i(\mathbf{p}, \overline{c})$ is given in Figure 6.6 for $\dim \mathbf{p} = 2$ and $i = 1$. Any point $\mathbf{q}$ inside $\mathbb{Q}_i(\mathbf{p}, \overline{c})$ satisfies $\hat{c}(\mathbf{q}) \leqslant \overline{c}$ and can thus be used to decrease the upper bound $\overline{c}$ along the direction i.

The algorithm CROSS, presented in Table 6.3, takes advantage of this idea to decrease the upper bound $\overline{c}$ for $\hat{c}(\hat{\mathbf{p}})$. The small positive real number κ is used to stop the procedure when the improvement is not significant enough. The set $\mathbb{Q}_i(\mathbf{p}, \overline{c})$ computed at Step 5 is, in general, a segment or a finite union of aligned segments. At Step 7, $\mathbf{q}$ is usually taken as the centre of the largest segment of $\mathbb{Q}_i(\mathbf{p}, \overline{c})$. The situation $\mathbb{Q}_i(\mathbf{p}, \overline{c}) = \varnothing$ (at Step 6) may only be encountered when $\hat{c}\,(\mathbf{p}) > \overline{c}$, *i.e.*, when the loop is executed for the first time. If the improvement on the upper bound is deemed sufficient ($\overline{c} - \widetilde{c} > \kappa$), then the loop is executed again from $\tilde{\mathbf{q}}$.

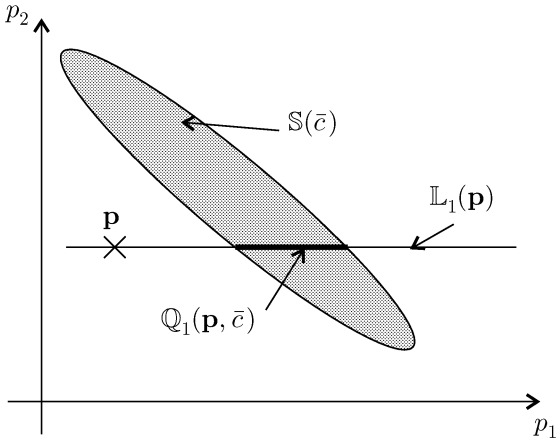


Fig. 6.6. Local search from $\mathbf{p}$ along the direction associated with p_1

Table 6.3. A local algorithm to decrease the upper bound of the optimum cost

Algorithm CROSS(in: c, $\mathbf{p}$, κ; inout: $\overline{c}$)	
1	$\widetilde{c} := \overline{c}$; $\tilde{\mathbf{q}} := \mathbf{p}$;
2	do
3	$\overline{c} := \widetilde{c}$; $\mathbf{p} := \tilde{\mathbf{q}}$;
4	for all $i \in \{1, \ldots, n_\mathrm{p}\}$
5	$\mathbb{Q}_i(\mathbf{p}, \overline{c}) := \mathbb{L}_i(\mathbf{p}) \cap \mathbb{S}(\overline{c})$;
6	if $\mathbb{Q}_i(\mathbf{p}, \overline{c}) = \varnothing$, next i;
7	select $\mathbf{q}$ inside $\mathbb{Q}_i(\mathbf{p}, \overline{c})$;
8	if $c(\mathbf{q}) \leqslant \widetilde{c}, \{\tilde{\mathbf{q}} := \mathbf{q}, \widetilde{c} := c(\mathbf{q})\}$;
9	while $\overline{c} - \widetilde{c} > \kappa$. // κ threshold to be chosen by user

The behaviour of this algorithm will now be illustrated on a very simple two-dimensional problem.

Example 6.6 *Finding the smallest disk $\mathcal{D}$ containing n points $A_1, \ldots, A_n$ of $\mathbb{R}^2$ is a minimax problem. The centre of the solution disk is the minimizer of the cost function*

$$c\,(p_1, p_2) = \max_{j \in \{1, \ldots, n\}} \left\{ \sqrt{(p_1 - x_{A_j})^2 + (p_2 - y_{A_j})^2} \right\}, \tag{6.40}$$

and its radius is the minimum $\widehat{c}$. If, for instance, $n = 3$ and the three points are $A_1(0, 4)$, $A_2(0, -4)$, $A_3\,(4, 0)$, then the minimizer is $\hat{\mathbf{p}} = \mathbf{0}$ and the minimum is $\widehat{c} = 4$. Note that c is not differentiable at $\hat{\mathbf{p}}$. Figure 6.7 presents level sets of c. CROSS *is run starting at $\mathbf{p} = (2, 8)^T$ and $\overline{c} = 6$. $\mathbb{S}_1(\overline{c})$, $\mathbb{S}_2(\overline{c})$ and $\mathbb{S}_3(\overline{c})$ are the disks in light grey on Figure 6.8. The darker set is $\mathbb{S}\,(\overline{c})$.*

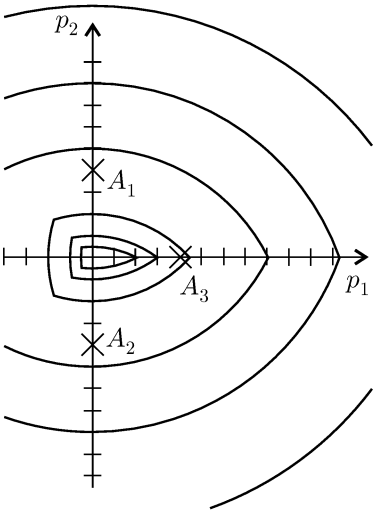


Fig. 6.7. Level sets of the cost function of Example 6.6

The loop is first executed for $i = 1$, *and at Step 5* CROSS *computes* $\mathbb{Q}_1(\mathbf{p}, \overline{c})$. *Since*

$$\begin{aligned}\mathbb{Q}_1(\mathbf{p}, \overline{c}) = &(\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_1(\overline{c})) \cap (\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_2(\overline{c})) \\ &\cap (\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_3(\overline{c})),\end{aligned} \tag{6.41}$$

computing $\mathbb{Q}_1(\mathbf{p}, \overline{c})$ *amounts to computing* $\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_j(\overline{c}), j = 1, 2, 3$. *This can be performed automatically by forward–backward propagation on the CSP with variables* p_i *and* r *and with the constraint* $|f_j(\mathbf{p})| - r = 0$ *and the domains* $[0, \overline{c}]$ *for* r *and* $\mathbb{R}$ *for* p_i. *For instance, for* $j = 1$,

$$\mathbb{S}_1(\overline{c}) = \{\mathbf{p} \in \mathbb{R}^2 \mid \sqrt{(p_1 - x_{A_1})^2 + (p_2 - y_{A_1})^2} \leqslant \overline{c}\}, \tag{6.42}$$

and

$$\mathbb{L}_1(\mathbf{p}) = \{\mathbf{q} \in \mathbb{R}^2 \mid q_2 = p_2\}. \tag{6.43}$$

As $\mathbf{p} = (2, 8)^T$, $\overline{c} = 6$, *and the coordinates of* A_1 *are* $x_{A_1} = 0$ *and* $y_{A_1} = 4$,

$$\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_1(\overline{c}) = \{\mathbf{p} \in \mathbb{R}^2 \mid \sqrt{p_1^2 + (p_2 - 4)^2} \leqslant 6,\ p_2 = 8\}. \tag{6.44}$$

$\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_1(\overline{c})$ *can now be computed as follows:*

$$\begin{aligned}\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_1(\overline{c}) &= \{\mathbf{p} \in \mathbb{R}^2 \mid \sqrt{p_1^2 + (8 - 4)^2} = r,\ r \in [0, 6],\ p_2 = 8\}, \\ &= \{\mathbf{p} \in \mathbb{R}^2 \mid p_1^2 = r^2 - 16,\ r \in [0, 6],\ p_2 = 8\}, \\ &= \{\mathbf{p} \in \mathbb{R}^2 \mid p_1 = \mathrm{sqr}^{-1}(r^2 - 16),\ r \in [0, 6],\ p_2 = 8\}, \\ &= \{\mathbf{p} \in \mathbb{R}^2 \mid p_1 \in \mathrm{sqr}^{-1}([0, 6]^2 - 16),\ p_2 = 8\}, \\ &= \{\mathbf{p} \in \mathbb{R}^2 \mid p_1 \in [-\sqrt{20}, \sqrt{20}],\ p_2 = 8\}, \\ &= [-\sqrt{20}, \sqrt{20}] \times [8, 8].\end{aligned}$$

The function sqr^{-1} *is the reciprocal function of the square function* sqr, *which should not be confused with the classical square root function (e.g.,* $\mathrm{sqr}^{-1}(4) = \{-2, 2\}$ *whereas* $\sqrt{4} = 2$*). (Note that* $\mathrm{sqr}^{-1}([4, 9]) = [-3, -2] \cup [2, 3]$*; it may thus be necessary to deal with unions of intervals.) The same reasoning, applied to* $\mathbb{L}_1(\mathbf{p}) \cap \mathbb{S}_2(\overline{c})$*, leads to the empty set. Therefore,* $\mathbb{Q}_1(\mathbf{p}, \overline{c}) = \varnothing$*. The horizontal direction* $i = 1$ *is thus eliminated and the loop of* Cross *is now executed for the vertical direction* $i = 2$*. We get*

$$\mathbb{Q}_2(\mathbf{p}, \overline{c}) = \mathbb{L}_2(\mathbf{p}) \cap \mathbb{S}(\overline{c}) = [2, 2] \times [-1.657, 1.657], \tag{6.45}$$

which corresponds to the thick segment in Figure 6.8. When the loop is left and if the centre of $\mathbb{Q}_2(\mathbf{p}, \overline{c})$ *is chosen as* $\tilde{\mathbf{q}}$*, then* $\tilde{\mathbf{q}} = (2, 0)^T$ *and* $\overline{c} = \sqrt{20}$*.* Cross *is then run again from* $\mathbf{p} = \tilde{\mathbf{q}}$*.* ■

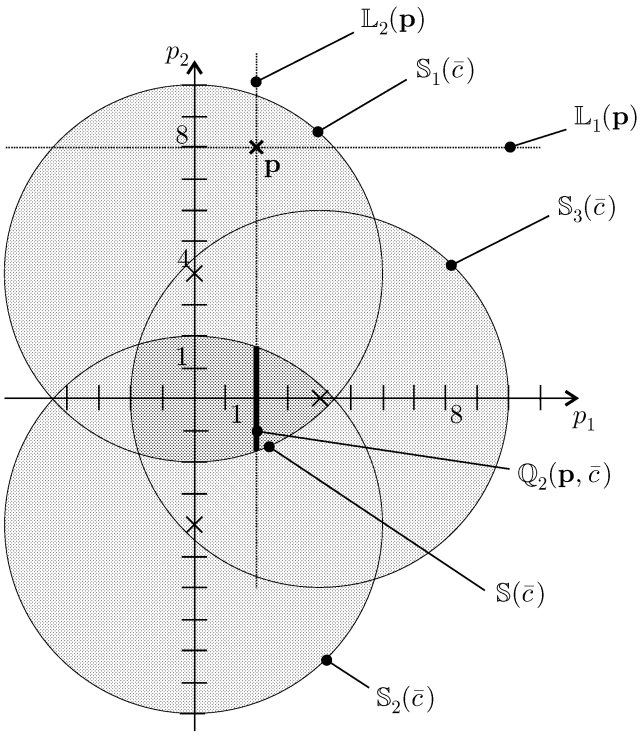


Fig. 6.8. An iteration of Cross

Example 6.7 *Consider again Example 6.5, where the cost function is given by (6.36). When the initial parameter vector and the options are well chosen, the procedure* minimax *of the* Matlab *toolbox* optim *(Brayton et al., 1979),*

finds a cost equal to 0.0657 *in about* 5 *s on a* PENTIUM 133. *This procedure, however, is local, is sensitive with respect to the initial parameter vector (it may even diverge), does not provide any guarantee on its results (even locally), often stops because of ill conditioning and never detects that the problem has two solutions for* $\mathbf{p}$. *By contrast, the approach advocated here is able to solve this minimax problem globally and efficiently. On the same computer, for* $\varepsilon = 0.05$ *(in* OPTIMIZE*),* $\kappa = 0.001$ *(in* CROSS*) and a search box equal to* $[-60, 60] \times [-1, 0] \times [-60, 60] \times [-1, 0]$, OPTIMIZE *proves in* 1.7 *s and after* 109 *bisections that* $\widehat{c} \in [0.0653, 0.0657]$. *The resulting subpaving* $\overline{\mathbb{S}}$ *(which contains all the global minimizers) consists of* 44 *boxes and has two symmetrical disconnected components. This shows that interval solvers can compete with* MATLAB *procedures, even from the point of view of computing time.* ■

6.3 Parameter Bounding

6.3.1 Introduction

The bounding approach to parameter estimation has received renewed attention in the 1990s (Walter, 1990; **?**; Deller et al., 1993; Norton, 1994; Norton, 1995; Milanese et al., 1996; Walter and Pronzato, 1997, and the many references therein). Of the reasons for this interest, we shall quote only two. First, this approach can deal with unavoidable structural errors resulting from the fact that the equations used to describe the system are always an approximation of reality. These errors are often deterministic, and thus not adequately described by random variables. Second, parameter bounding is well suited to the guaranteed characterization of parameter uncertainty, a prerequisite for a number of methods in robust control (see Chapter 7). In the context of bounded-error estimation, interval methods have been introduced independently by Moore (1992) and Jaulin and Walter (1993b).

In a bounding approach, the set to be characterized is $\widehat{\mathbb{X}}_{\mathrm{s}} = \check{\mathbb{X}} \cap \mathbb{M}$. Even if some algorithm could in principle be found to perform this task, an accurate characterization of $\widehat{\mathbb{X}}_{\mathrm{s}}$ often turns out to be too complex because $\mathcal{X} = \{x_1, \ldots, x_n\}$ generally contains many elements. In practice, however, it is often possible to partition $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$ into three vectors $\mathbf{y}$, $\mathbf{p}$ and $\mathbf{t}$ in such a way that there exists a function $\boldsymbol{\phi}$ such that

$$\mathbf{x} \in \mathbb{M} \Leftrightarrow (\mathbf{y}, \mathbf{p}, \mathbf{t}) \in \mathbb{M} \Leftrightarrow \mathbf{y} = \boldsymbol{\phi}(\mathbf{p}, \mathbf{t}). \tag{6.46}$$

When the vectors $\mathbf{p}$ and $\mathbf{t}$ are concatenated to form a vector $\mathbf{z}$, this corresponds to (6.10), page 144. Distinguishing $\mathbf{p}$ and $\mathbf{t}$ will make it possible to distinguish parameters to be estimated (the *parameter vector* $\mathbf{p} = (p_1, \ldots, p_{n_{\mathrm{p}}})^{\mathrm{T}}$) from other uncertain quantities (the vector of the values taken by the *independent variables* $\mathbf{t} = (t_1, \ldots, t_{n_{\mathrm{t}}})^{\mathrm{T}}$) introduced only to allow the estimation of $\mathbf{p}$. In practice $\mathbf{t}$ may correspond to the actual instants of time at which experimental data are collected, if these instants are uncertain. The

vector $\mathbf{y} = (y_1, \ldots, y_{n_\mathrm{y}})^\mathrm{T}$ (the *output vector*) consists of variables whose values could be computed from the constrained set $\mathbb{M}$ if the values of $\mathbf{p}$ and $\mathbf{t}$ were known. A *simulator* $\boldsymbol{\phi}$ is a function from $\mathbb{R}^{n_\mathrm{p}+n_\mathrm{t}}$ to $\mathbb{R}^{n_\mathrm{y}}$ computing $\mathbf{y}$ from $\mathbf{p}$ and $\mathbf{t}$. Since the variables of interest are stored in $\mathbf{p}$, the set to be characterized is the projection $\widehat{\mathbb{P}}$ of $\widehat{\mathbb{X}}_\mathrm{s}$ onto parameter space:

$$\widehat{\mathbb{P}} = \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid \exists \mathbf{t} \in \mathbb{R}^{n_\mathrm{t}}, \exists \mathbf{y} \in \mathbb{R}^{n_\mathrm{y}}, \, (\mathbf{y}, \mathbf{p}, \mathbf{t}) \in \widehat{\mathbb{X}}_\mathrm{s} = \check{\mathbb{X}} \cap \mathbb{M}\}. \tag{6.47}$$

Because of (6.46), this implies that

$$\widehat{\mathbb{P}} = \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid \exists \mathbf{t} \in \mathbb{R}^{n_\mathrm{t}}, \mathbf{y} = \boldsymbol{\phi}(\mathbf{p}, \mathbf{t}) \text{ and } (\mathbf{y}, \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}\} \tag{6.48}$$

$$= \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid \exists \mathbf{t} \in \mathbb{R}^{n_\mathrm{t}}, \; (\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}\}. \tag{6.49}$$

Sivia can be used for the characterization of $\widehat{\mathbb{P}}$, provided that an inclusion test $[\tau_\mathrm{p}]([\mathbf{p}])$ is available for the test

$$\tau_\mathrm{p}(\mathbf{p}) \triangleq (\exists \mathbf{t} \in \mathbb{R}^{n_\mathrm{y}} \mid (\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}). \tag{6.50}$$

Let $[\tau_\mathrm{ypt}]([\mathbf{y}], [\mathbf{p}], [\mathbf{t}])$ be an inclusion test for the test $\tau_\mathrm{ypt} \triangleq ((\mathbf{y}, \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}})$. An inclusion test for the test

$$\tau_\mathrm{pt}(\mathbf{p}, \mathbf{t}) \triangleq \big((\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}\big) \tag{6.51}$$

is then

$$[\tau_\mathrm{pt}]([\mathbf{p}], [\mathbf{t}]) = [\tau_\mathrm{ypt}]([\boldsymbol{\phi}]([\mathbf{p}], [\mathbf{t}]), [\mathbf{p}], [\mathbf{t}]), \tag{6.52}$$

where $[\boldsymbol{\phi}]([\mathbf{p}], [\mathbf{t}])$ is an inclusion function for $\boldsymbol{\phi}(\mathbf{p}, \mathbf{t})$. An inclusion test for $\tau_\mathrm{p}(\mathbf{p})$ is finally given by the algorithm of Table 6.4. The initial search box $[\check{\mathbf{t}}]$ is assumed to be large enough to contain the projection of $\check{\mathbb{X}}$ onto $\mathbf{t}$-space.

On the one hand, the algorithm attempts to partition $[\check{\mathbf{t}}]$ into subboxes $[\mathbf{t}]$ such that

$$(\boldsymbol{\phi}([\mathbf{p}], [\mathbf{t}]), [\mathbf{p}], [\mathbf{t}]) \cap \check{\mathbb{X}} = \varnothing, \tag{6.53}$$

for all of these subboxes. All subboxes of $[\check{\mathbf{t}}]$ still to be studied are stored in the queue $\mathcal{Q}$. If the algorithm succeeds, which means that $[\mathbf{p}] \cap \mathbb{P} = \varnothing$, it returns 0 at Step 3.

On the other hand, the algorithm tries to find one $\mathbf{t}$ such that

$$(\boldsymbol{\phi}([\mathbf{p}], \mathbf{t}), [\mathbf{p}], \mathbf{t}) \subset \check{\mathbb{X}}. \tag{6.54}$$

When such a $\mathbf{t}$ is found, this means that $[\mathbf{p}] \subset \mathbb{P}$ and the algorithm returns 1 at Step 5.

The test at Step 7 is introduced to avoid splitting $[\check{\mathbf{t}}]$ *ad infinitum* and to introduce some relation with the splitting policy followed for $[\mathbf{p}]$ by Sivia. When the algorithm fails to reach a conclusion, $[0, 1]$ is returned at Step 3.

Table 6.4. Inclusion test for the test of a box $[\mathbf{p}]$ for feasibility

Algorithm $[\tau_{\mathrm{p}}]$(in: τ_{pt}, $[\mathbf{p}]$, $[\check{\mathbf{t}}]$; out: $[\tau_{\mathrm{p}}]$)	
1	$\mathcal{Q} := \{[\check{\mathbf{t}}]\}$; $[\tau_{\mathrm{p}}] := 0$;
2	repeat
3	if $\mathcal{Q} = \varnothing$ then return;
4	pop the first box out of $\mathcal{Q}$ into $[\mathbf{t}]$;
5	if $([\tau_{\mathrm{pt}}]([\mathbf{p}], \mathrm{mid}([\mathbf{t}])) = 1)$ then $[\tau_{\mathrm{p}}] := 1$; return;
6	if $([\tau_{\mathrm{pt}}]([\mathbf{p}], [\mathbf{t}]) \neq 0)$ then
7	if $w([\mathbf{t}]) < w([\mathbf{p}])$, $[\tau_{\mathrm{p}}] := [0, 1]$;
8	else bisect $[\mathbf{t}]$ and put the resulting boxes at the end of $\mathcal{Q}$;
9	forever.

In practice, the dimension n_{t} of $\mathbf{t}$ is high compared to the dimension n_{p} of $\mathbf{p}$, and the bisections performed at Step 8 make the complexity of the algorithm exponential with n_{t}. We now consider a situation where such bisections can be avoided. Assume that the prior feasible set is a box, written as

$$\check{\mathbb{X}} \triangleq [\check{y}_1] \times \cdots \times [\check{y}_{n_{\mathrm{y}}}] \times [\check{p}_1] \times \cdots \times [\check{p}_{n_{\mathrm{p}}}] \times [\check{t}_1] \times \cdots \times [\check{t}_{n_{\mathrm{t}}}]. \tag{6.55}$$

Assume also that t_i corresponds to the time at which the measurement $\check{y}_i$ of y_i is collected, $i = 1, \ldots, n_{\mathrm{y}}$. As a result, n_{t} is equal to n_{y} and the ith component of the relation $\mathbf{y} = \boldsymbol{\phi}(\mathbf{p}, \mathbf{t})$ can now be written as $y_i = \phi_i(\mathbf{p}, t_i)$. This means that the value of y_i does not depend on the time at which the other measurements are collected, a very common situation. The test $\tau_{\mathrm{p}}(\mathbf{p})$ defined by (6.50) is then equivalent to

$$\left(\begin{array}{l} \exists t_1 \in [\check{t}_1], \ldots, \exists t_{n_{\mathrm{y}}} \in [\check{t}_{n_{\mathrm{y}}}] \text{ such that} \\ p_1 \in [\check{p}_1](0), \ldots, p_{n_{\mathrm{p}}} \in [\check{p}_{n_{\mathrm{p}}}](0) \\ \phi_1(\mathbf{p}, t_1) \in [\check{y}_1], \ldots, \phi_{n_{\mathrm{y}}}(\mathbf{p}, t_{n_{\mathrm{y}}}) \in [\check{y}_{n_{\mathrm{y}}}] \end{array} \right), \tag{6.56}$$

i.e., to

$$\left(\begin{array}{l} p_1 \in [\check{p}_1](0), \ldots, p_{n_{\mathrm{p}}} \in [\check{p}_{n_{\mathrm{p}}}](0) \\ \exists t_1 \in [\check{t}_1] \mid \phi_1(\mathbf{p}, t_1) \in [\check{y}_1] \\ \qquad \vdots \\ \exists t_{n_{\mathrm{y}}} \in [\check{t}_{n_{\mathrm{y}}}] \mid \phi_{n_{\mathrm{y}}}(\mathbf{p}, t_{n_{\mathrm{y}}}) \in [\check{y}_{n_{\mathrm{y}}}] \end{array} \right). \tag{6.57}$$

Let $[\sigma_i]([\mathbf{p}])$ be an inclusion test for $\sigma_i(\mathbf{p}) \triangleq (p_i \in [\check{p}_i])$ and let $[\eta_i]([\mathbf{p}])$ be an inclusion test for the test $\eta_i(\mathbf{p}) \triangleq (\exists t_i \in [t_i] \mid \phi_i(\mathbf{p}, t_i) \in [\check{y}_i])$. An inclusion test for $\tau_{\mathrm{p}}(\mathbf{p})$ is then given by

$$[\tau_{\mathrm{p}}]([\mathbf{p}]) = [\sigma_1]([\mathbf{p}]) \wedge \cdots \wedge [\sigma_{n_{\mathrm{p}}}]([\mathbf{p}]) \wedge [\eta_1]([\mathbf{p}]) \wedge \cdots \wedge [\eta_{n_{\mathrm{y}}}]([\mathbf{p}]). \tag{6.58}$$

Now, the inclusion test $[\eta_i]([\mathbf{p}])$, implemented along the lines of Table 6.4, requires only one-dimensional bisections along t_i. Evaluating $[\tau_{\mathrm{p}}]([\mathbf{p}])$ thus requires n_{y} searches in one-dimensional spaces instead of one search in an n_{y}-dimensional space, a drastic simplification.

An even simpler situation will be considered in the next section, where the value of $\mathbf{t}$ is assumed to be known exactly and thus need not be included in the variables to be considered. We shall return to uncertain measurements of independent variables in Section 6.3.4.

6.3.2 The values of the independent variables are known

Assume that $\mathbf{t}$ need not be incorporated in the list of variables of the estimation problem, for instance because the errors committed when measurering the measurement times are negligible. Equation (6.49) then simplifies to

$$\widehat{\mathbb{P}} = \{\mathbf{p} \in \mathbb{R}^{n_{\mathrm{p}}} \mid (\boldsymbol{\phi}(\mathbf{p}), \mathbf{p}) \in \check{\mathbb{X}}\}. \tag{6.59}$$

Assume also that $\check{\mathbb{X}}$ is a box, which means that prior knowledge on each variable is independent of prior knowledge on the other, *i.e.*,

$$\check{\mathbb{X}} \triangleq [\check{y}_1] \times \cdots \times [\check{y}_{n_{\mathrm{y}}}] \times [\check{p}_1] \times \cdots \times [\check{p}_{n_{\mathrm{p}}}] = [\check{\mathbf{y}}] \times [\check{\mathbf{p}}]. \tag{6.60}$$

Then

$$\begin{aligned} (\boldsymbol{\phi}(\mathbf{p}), \mathbf{p}) \in \check{\mathbb{X}} &\Leftrightarrow \boldsymbol{\phi}(\mathbf{p}) \in [\check{\mathbf{y}}] \text{ and } \mathbf{p} \in [\check{\mathbf{p}}] \\ &\Leftrightarrow \mathbf{p} \in \boldsymbol{\phi}^{-1}([\check{\mathbf{y}}]) \text{ and } \mathbf{p} \in [\check{\mathbf{p}}] \\ &\Leftrightarrow \mathbf{p} \in [\check{\mathbf{p}}] \cap \boldsymbol{\phi}^{-1}([\check{\mathbf{y}}]). \end{aligned} \tag{6.61}$$

Thus

$$\widehat{\mathbb{P}} = [\check{\mathbf{p}}] \cap \boldsymbol{\phi}^{-1}([\check{\mathbf{y}}]), \tag{6.62}$$

and characterizing $\widehat{\mathbb{P}}$ is a set-inversion problem, which can be solved using Sivia. Note that a number of specific methods are available to characterize $\widehat{\mathbb{P}}$ when $\boldsymbol{\phi}(\mathbf{p})$ is linear. In this case, $\widehat{\mathbb{P}}$ is a polytope, which can be characterized exactly (Walter and Piet-Lahanier, 1989) or enclosed in a simpler-shaped set such as an ellipsoid (Fogel and Huang, 1982) or a box (Milanese and Belforte, 1982; Belforte et al., 1990). When $\boldsymbol{\phi}(\mathbf{p})$ is non-linear, there are much fewer methods leading to guaranteed results. One of them is based on signomial programming (Milanese and Vicino, 1991). Another approach based on interval analysis and similar to Sivia was independently developed by Moore (1992).

Example 6.8 *A two-parameter problem will be used as an illustrative example, which will make it possible to draw pictures of the paving obtained. This example is taken from Jaulin and Walter (1993a) and is a simplified version of a problem considered in Milanese and Vicino (1991). An example related*

Table 6.5. Measurement times and corresponding interval data

i	t_i	$[\check{y}_i]$
1	0.75	$[2.7, 12.1]$
2	1.5	$[1.04, 7.14]$
3	2.25	$[-0.13, 3.61]$
4	3	$[-0.95, 1.15]$
5	6	$[-4.85, -0.29]$
6	9	$[-5.06, -0.36]$
7	13	$[-4.1, -0.04]$
8	17	$[-3.16, 0.3]$
9	21	$[-2.5, 0.51]$
10	25	$[-2, 0.67]$

to electrochemistry can be found in Braems et al. (2001). The set $\widehat{\mathbb{P}}$ *to be characterized consists of all parameter vectors* $\mathbf{p}$ *such that the graph of the function*

$$f(\mathbf{p}, t) = 20\exp(-p_1 t) - 8\exp(-p_2 t) \tag{6.63}$$

crosses all the data bars of Figure 6.9. The numerical values of the corresponding interval data are given in Table 6.5.
In this simulated example, the interval data have been computed by in a sim-

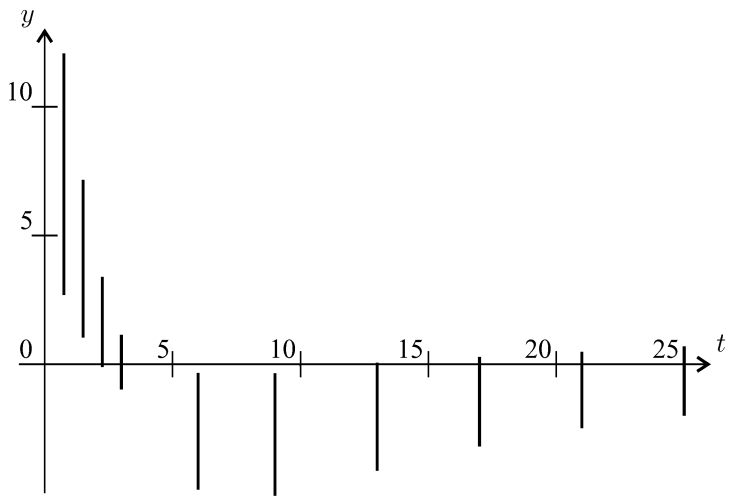


Fig. 6.9. Experimental data of Example 6.8, together with their uncertainty intervals

plified way by adding a centred error interval with radius $\rho_i = 0.5|y_i| + 1$ *to the* i*th component of the data vector*

$$\check{\mathbf{y}} = (7.39, 4.09, 1.74, 0.097, -2.57, \\ -2.71, -2.07, -1.44, -0.98, -0.66)^T, \tag{6.64}$$

for $i = 1, \ldots, 10$. *The posterior feasible set for the parameters is given by (6.62), where the search domain* $[\check{\mathbf{p}}]$ *is taken as* $[-0.1, 1.5]^{\times 2}$. *The coordinate functions of* $\boldsymbol{\phi}$ *are given by*

$$\phi_i(\mathbf{p}) = 20\exp(-p_1 t_i) - 8\exp(-p_2 t_i), \ i = 1, \ldots, 10. \tag{6.65}$$

In less than 4 *s on a* PENTIUM 133, SIVIA *generates the paving of Figure 6.10, thus bracketing the posterior feasible set for* $\mathbf{p}$ *between inner and outer approximations.* ■

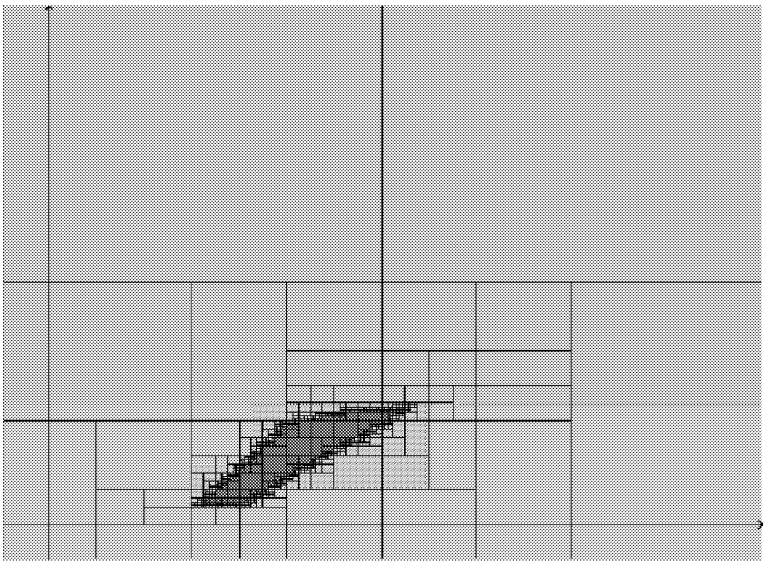


Fig. 6.10. Paving generated by SIVIA for Example 6.8 to bracket the posterior feasible set $\widehat{\mathbb{P}}$ for the parameters between inner and outer approximations; the outer frame corresponds to the box $[-0.1, 1.5]^{\times 2}$

6.3.3 Robustification against outliers

The approach considered so far relied on the hypothesis that the prior feasible set $\check{\mathbb{X}}$ did contain the actual values for the variables, which is unfortunately not always realistic. Assume again that $\check{\mathbb{X}}$ is the box defined by (6.55). Even

if each prior interval component of $\check{\mathbb{X}}$ is obtained from a reasonable model of measurement inaccuracy, for instance deduced from sensor technical data sheets, in practice things do not always happen as expected. For instance,

- some constraints in $\mathbb{M}$ may not always hold true,
- a sensor may fail during data collection,
- there might be rare situations where some error bounds turn out to be optimistic.

A variable x_i whose actual value does not belong to its prior interval is called an *outlier*. Robot localization (Chapter 8) will provide a context where such outliers are more or less unavoidable. The presence of outliers may be detected after completion of the estimation process, if the posterior feasible set turns out to be empty. Unfortunately, it may also escape detection. To protect oneself against the fact that $\check{\mathbb{X}}$ may not contain the actual values of some of the variables in $\mathbf{x}$, one may enlarge (or relax) $\check{\mathbb{X}}$. For this purpose, consider a *relaxing function* $\lambda : \mathbb{R}^n \to [0,1]$ such that $\lambda = 1$ if and only if $\mathbf{x} \in \check{\mathbb{X}}$. The *relaxed prior feasible set*

$$\check{\mathbb{X}}_\alpha \triangleq \{\mathbf{x} \in \mathbb{R}^n \mid \lambda(\mathbf{x}) \geqslant \alpha\} = \lambda^{-1}([\alpha, 1]) \tag{6.66}$$

for $\alpha \in [0,1]$ contains $\check{\mathbb{X}}$. Moreover $\check{\mathbb{X}}_1 = \check{\mathbb{X}}$ and $\check{\mathbb{X}}_0 = \mathbb{R}^n$.

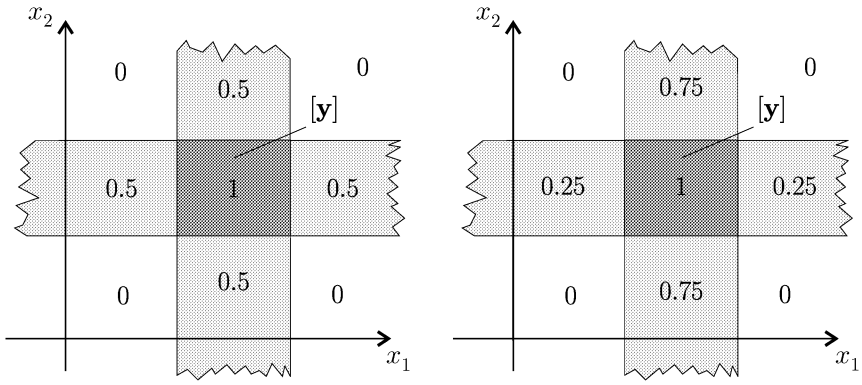


Fig. 6.11. Relaxing functions

Example 6.9 *Define the* characteristic function $\pi_{[a,b]} : \mathbb{R} \to \mathbb{R}$ *of the interval* $[a,b]$ *as*

$$\pi_{[a,b]}(x) = \begin{cases} 1 \text{ if } x \in [a,b] \\ 0 \text{ if } x \notin [a,b]. \end{cases} \tag{6.67}$$

The function

$$\lambda_1(\mathbf{x}) = \frac{\pi_{[y_1]}(x_1) + \cdots + \pi_{[y_n]}(x_n)}{n} \tag{6.68}$$

is then relaxing for the box $[\mathbf{y}] = [y_1] \times \cdots \times [y_n]$. *If the* $w_i, i \in \{1, \ldots, n\}$, *are positive weights, then the function*

$$\lambda_w(\mathbf{x}) = \frac{w_1\pi_{[y_1]}(x_1) + \cdots + w_n\pi_{[y_n]}(x_n)}{w_1 + \cdots + w_n} \tag{6.69}$$

is also relaxing for the box $[\mathbf{y}]$. *Figure 6.11 illustrates the relaxing functions* $(\pi_{[y_1]}(x_1) + \pi_{[y_2]}(x_2))/2$ *(left) and* $(3\pi_{[y_1]}(x_1) + \pi_{[y_2]}(x_2))/4$ *(right).* ■

Allowing q out of the n variables x_i to escape their prior feasible intervals amounts to enlarging the prior feasible box $\check{\mathbb{X}}$ by choosing

$$\lambda(\mathbf{x}) = \frac{\pi_{[\check{x}_1]}(x_1) + \cdots + \pi_{[\check{x}_n]}(x_n)}{n} \text{ and } \alpha = 1 - \frac{q}{n}, \tag{6.70}$$

where the $[\check{x}_i]$s are the interval components of $\check{\mathbb{X}}$. An inclusion test for $\check{\mathbb{X}}_\alpha$ can be obtained via an interval evaluation of $\lambda(\mathbf{x})$ and the characterization of $\check{\mathbb{X}}_\alpha$ can be performed by using SIVIA. For the characterization of the *relaxed posterior feasible set* for the parameters, defined by

$$\widehat{\mathbb{P}}_\alpha = \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid \exists \mathbf{t} \in \mathbb{R}^{n_\mathrm{t}},\ (\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}_\alpha\}, \tag{6.71}$$

see (6.49), SIVIA can also be used, provided that an inclusion test is available for the test

$$\tau_\mathrm{p}^\alpha \triangleq (\exists \mathbf{t} \in \mathbb{R}^{n_\mathrm{t}} \mid (\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}_\alpha). \tag{6.72}$$

Such a test is given by the algorithm of Table 6.4, where the test $\tau_\mathrm{pt}(\mathbf{p}, \mathbf{t})$ is replaced by $\tau_\mathrm{pt}^\alpha(\mathbf{p}, \mathbf{t}) \triangleq \big((\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}_\alpha\big)$. To illustrate this approach, assume that the values taken by the independent variables are certain, so the entries of $\mathbf{t}$ are not variables of the estimation problem. The prior feasible domain for the variables is then the box $\check{\mathbb{X}} = [\check{\mathbf{y}}] \times [\check{\mathbf{p}}]$. The posterior feasible set for the parameters is thus

$$\hat{\mathbb{P}} = \{\mathbf{p} \in \mathbb{R}^{n_\mathrm{p}} \mid (\boldsymbol{\phi}(\mathbf{p}), \mathbf{p}) \in [\check{\mathbf{y}}] \times [\check{\mathbf{p}}]\}. \tag{6.73}$$

Allow up to q of the n_y output variables y_i to escape their prior feasible intervals. This amounts to replacing the box $[\check{\mathbf{y}}]$ in (6.73) by the set

$$\check{\mathbb{Y}}_q \triangleq \{\mathbf{y} \in \mathbb{R}^{n_\mathrm{y}} \mid \pi_{[\check{y}_1]}(y_1) + \cdots + \pi_{[\check{y}_{n_\mathrm{y}}]}(y_{n_\mathrm{y}}) \geqslant n_\mathrm{y} - q\}, \tag{6.74}$$

and $\hat{\mathbb{P}}$ by

$$\hat{\mathbb{P}}_q = \{\mathbf{p} \in [\check{\mathbf{p}}] \mid \boldsymbol{\phi}(\mathbf{p}) \in \check{\mathbb{Y}}_q\} = [\check{\mathbf{p}}] \cap \boldsymbol{\phi}^{-1}(\check{\mathbb{Y}}_q). \tag{6.75}$$

SIVIA can be used to characterize $\hat{\mathbb{P}}_q$ for any prespecified integer value of q in $[0, n_\mathrm{y}]$.

Example 6.10 *Consider again Example 6.8, but assume now that the vector* $\check{\mathbf{y}}$ *comprising all the available data is*

$$\check{\mathbf{y}} = (7.39, 0, 1.74, 0.097, -2.57, -2.71, -2.07, 0, -0.98, -0.66)^T, \quad (6.76)$$

which illustrates a situation where there are two outliers ($\check{y}(2) = 0$ *instead of* 4.09 *and* $\check{y}(8) = 0$ *instead of* -1.44*). The resulting interval data are depicted in Figure 6.12.* SIVIA *generates the subpavings depicted in Figure 6.13 for* $q = 0$ *(a),* $q = 1$ *(b) and* $q = 2$ *(c), respectively associated with the sets* $\hat{\mathbb{P}}_0$, $\hat{\mathbb{P}}_1$ *and* $\hat{\mathbb{P}}_2$*. The required accuracy was taken as* $\varepsilon = 0.005$ *and the prior search box for the parameters was* $[\check{\mathbf{p}}] = [-0.1, 1.5] \times [-0.1, 1.5]$*.* $\hat{\mathbb{P}}_0$ *turns out to be empty, which proves that there is at least one outlier in the data. The fact that* $\hat{\mathbb{P}}_1$ *is not empty should serve as a warning that outliers may go undetected. If one looks for an outer approximation of the posterior feasible set that would be obtained in the absence of outliers, then one should rather overestimate the actual number of outliers. Since there are indeed two outliers,* $\hat{\mathbb{P}}_2$ *provides such an outer approximation and contains the set* $\hat{\mathbb{P}}$ *represented in Figure 6.10.* $\hat{\mathbb{P}}_2$ *is disconnected because there are two different strategies to eliminate two interval data in order to be able to be consistent with the eight remaining ones.* ■

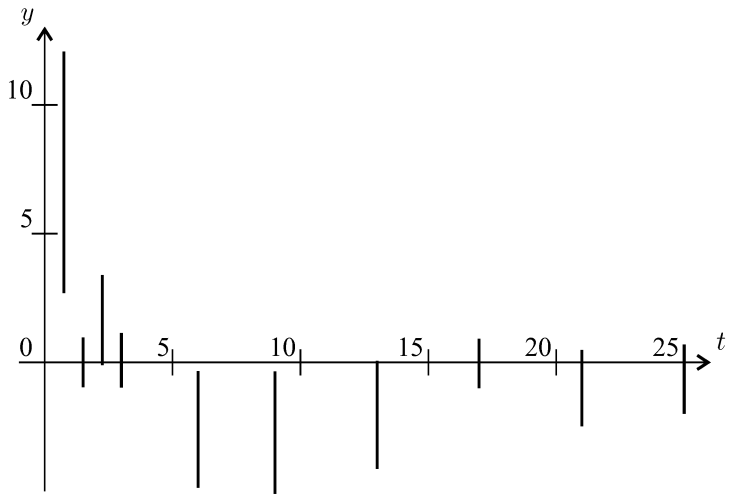


Fig. 6.12. Interval data with two outliers

The size of $\hat{\mathbb{P}}_q$ increases with q and a compromise must of course be struck between the level of protection against outliers and the size of the resulting set estimate for the parameters. A possible strategy (Jaulin et al., 1996) is to start assuming that $q = 0$, and to increase q by one as long as $\hat{\mathbb{P}}_q$ remains

empty. This choice, which corresponds to a guaranteed implementation of OMNE (for *Outlier Minimal Number Estimator* (Lahanier et al., 1987)), leads to stopping at $\hat{\mathbb{P}}_1$ in Example 6.10.

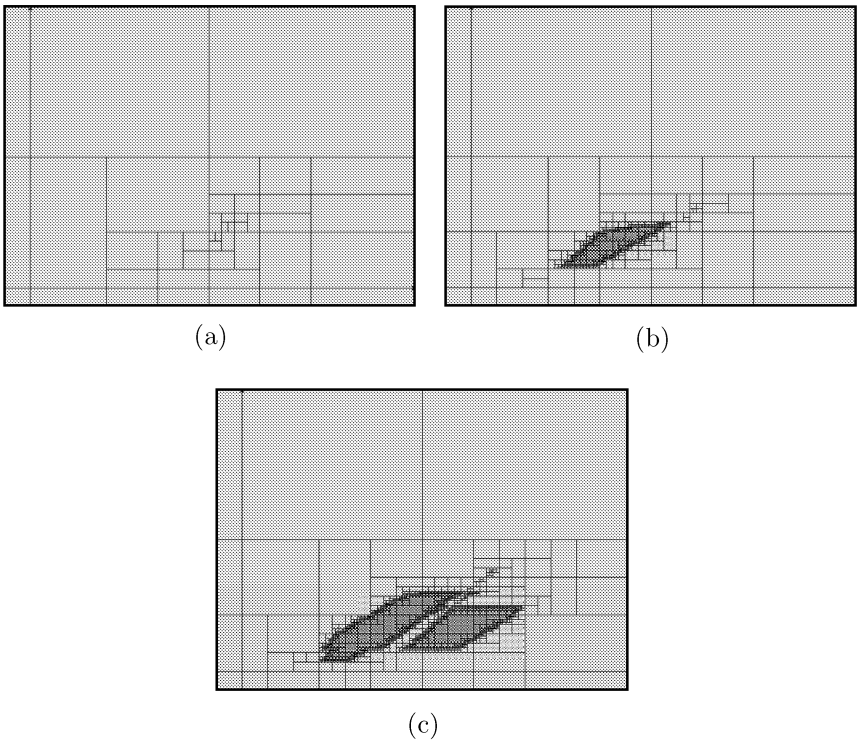


Fig. 6.13. Pavings generated by SIVIA for Example 6.10; (a) no outlier assumed; (b) up to one outlier assumed; (c) up to two outliers assumed; the frames correspond to the parameter box $[-0.1, 1.5] \times [-0.1, 1.5]$

6.3.4 The values of the independent variables are uncertain

In the literature devoted to parameter bounding, little attention has been paid to uncertainty in the measurement of independent variables. If the coordinate functions of $\boldsymbol{\phi}(\mathbf{p}, \mathbf{t})$ are bilinear in $\mathbf{t}$ and $\mathbf{p}$ and if the prior feasible domains $[\check{\mathbf{t}}]$ and $[\check{\mathbf{p}}]$ for $\mathbf{t}$ and $\mathbf{p}$ are boxes, the problem of characterizing the posterior feasible set $\hat{\mathbb{P}}$ for the parameters can be solved exactly with the technique described in Cerone (1991, 1996). Ellipsoidal outer approximations of this set can also be computed (Norton, 1987; Clément and Gentil, 1990; Pronzato and Walter, 1994; Norton, 1996; Veres and Norton, 1996).

We shall consider a more general situation where $\boldsymbol{\phi}(\mathbf{p}, \mathbf{t})$ may be non-linear in $\mathbf{p}$ and in $\mathbf{t}$. To the best of our knowledge, guaranteed and accurate results in such a non-linear bounding context were first presented in Jaulin and Walter (1999). Non-linear parameter estimation with such *errors in variables* has been considered in the context of least squares for many years (Schwetlick and Tiller, 1985), but the results are obtained by local methods and thus not guaranteed.

Table 6.6. Prior feasible intervals for the data of Example 6.11

i	$\check{t}_i$	$[\check{t}_i]$	$[\check{y}_i]$
1	0.75	$[-0.25, 1.75]$	$[2.7, 12.1]$
2	1.5	$[0.5, 2.5]$	$[1.04, 7.14]$
3	2.25	$[1.25, 3.25]$	$[-0.13, 3.61]$
4	3	$[2, 4]$	$[-0.95, 1.15]$
5	6	$[5, 7]$	$[-4.85, -0.29]$
6	9	$[8, 10]$	$[-5.06, -0.36]$
7	13	$[12, 14]$	$[-4.1, -0.04]$
8	17	$[16, 18]$	$[-3.16, 0.3]$
9	21	$[20, 22]$	$[-2.5, 0.51]$
10	25	$[24, 26]$	$[-2, 0.67]$

Example 6.11 *Consider again Example 6.5, page 149. Assume now that the values taken by the independent variables are uncertain. The prior interval* $[\check{t}_i]$ *is obtained by adding the interval* $[-1, 1]$ *to the associated measurement time* $\check{t}_i$*. The resulting prior intervals* $[\check{t}_i]$ *for the* t_i*s,* $i = 1, \ldots, 10$*, are given in Table 6.6. Figure 6.14 presents the data. The uncertainty associated with each pair of output and time data is materialized by a grey box. The set* $\hat{\mathbb{P}}$ *to be characterized consists of all the values of* $\mathbf{p} = (p_1, p_2)^T$ *such that the graph of the function*

$$f(\mathbf{p}, t) = 20 \exp(-p_1 t) - 8 \exp(-p_2 t) \tag{6.77}$$

goes through all ten boxes of Figure 6.14. It is defined as

$$\hat{\mathbb{P}} = \{\mathbf{p} \in \mathbb{R}^{n_p} \mid \exists \mathbf{t} \in \mathbb{R}^{n_t},\ (\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}\}, \tag{6.78}$$

where

$$\phi_i(\mathbf{p}, \mathbf{t}) = \phi_i(\mathbf{p}, t_i) = 20 \exp(-p_1 t_i) - 8 \exp(-p_2 t_i) \tag{6.79}$$

and

$$\check{\mathbb{X}} = [\check{y}_1] \times \cdots \times [\check{y}_{10}] \times [\check{p}_1] \times [\check{p}_2] \times [\check{t}_1] \times \cdots \times [\check{t}_{10}]. \tag{6.80}$$

The prior box for the parameters is taken as

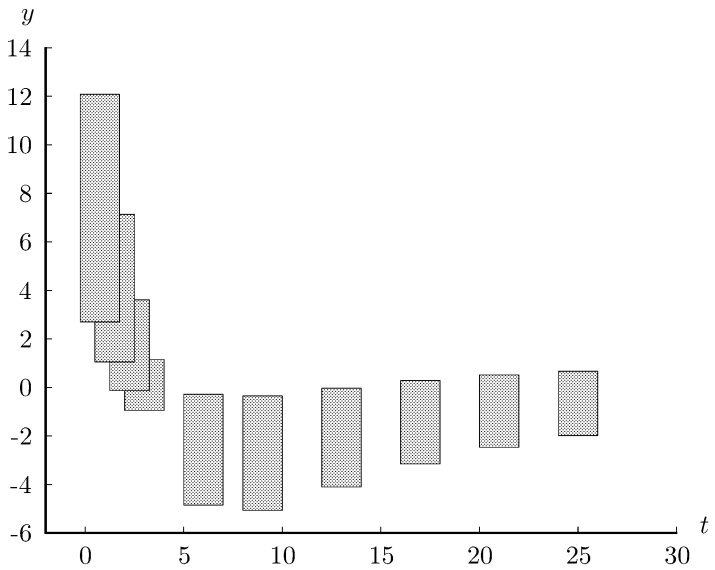


Fig. 6.14. Uncertain experimental data with uncertain measurement times

$$[\check{\mathbf{p}}] = [\check{p}_1] \times [\check{p}_2] = [0, 1.2] \times [0, 0.5]. \tag{6.81}$$

SIVIA *can be used to characterize* $\hat{\mathbb{P}}$*, provided that an inclusion test is available for*

$$\exists \mathbf{t} \in \mathbb{R}^{10} \mid (\boldsymbol{\phi}(\mathbf{p}, \mathbf{t}), \mathbf{p}, \mathbf{t}) \in \check{\mathbb{X}}. \tag{6.82}$$

This test can be rewritten as

$$\begin{aligned} \exists (t_1, \ldots, t_{10}) \in \mathbb{R}^{10}, (\phi_1(\mathbf{p}, t_1), \ldots, \phi_{10}(\mathbf{p}, t_{10}), p_1, p_2, t_1, \ldots, t_{10}) \\ \in [\check{y}_1] \times \cdots \times [\check{y}_{10}] \times [\check{p}_1] \times [\check{p}_2] \times [\check{t}_1] \times \cdots \times [\check{t}_{10}], \end{aligned} \tag{6.83}$$

or equivalently as

$$\begin{aligned} &\exists \begin{pmatrix} t_1 \\ \vdots \\ t_{10} \end{pmatrix} \in \begin{pmatrix} [\check{t}_1] \\ \vdots \\ [\check{t}_{10}] \end{pmatrix} \mid \begin{pmatrix} \phi_1(\mathbf{p}, t_1) \\ \vdots \\ \phi_{10}(\mathbf{p}, t_{10}) \end{pmatrix} \in \begin{pmatrix} [\check{y}_1] \\ \vdots \\ [\check{y}_{10}] \end{pmatrix} \\ &\textit{and } \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \in \begin{pmatrix} [\check{p}_1] \\ [\check{p}_2] \end{pmatrix} \end{aligned} \tag{6.84}$$

i.e., *as*

$$(\mathbf{p} \in [\check{\mathbf{p}}]) \wedge \eta_1(\mathbf{p}) \wedge \cdots \wedge \eta_{n_\mathrm{y}}(\mathbf{p}), \tag{6.85}$$

where $\eta_i(\mathbf{p})$ *is the test*

$$\eta_i(\mathbf{p}) = (\exists t_i \in [\check{t}_i] \mid \phi_i(\mathbf{p}, t_i) \in [\check{y}_i]). \tag{6.86}$$

An inclusion test for $\eta_i(\mathbf{p})$ is then obtained by the algorithm of Table 6.4. For $\varepsilon = 0.01$, SIVIA *generates the subpavings represented on Figure 6.15 in* 38 *s on a* PENTIUM 133 *(Jaulin and Walter, 1999). The dark grey boxes have been proved to be included in $\hat{\mathbb{P}}$ and the light grey boxes have been proved to have an empty intersection with $\hat{\mathbb{P}}$. No conclusion has been reached for the black boxes.* ■

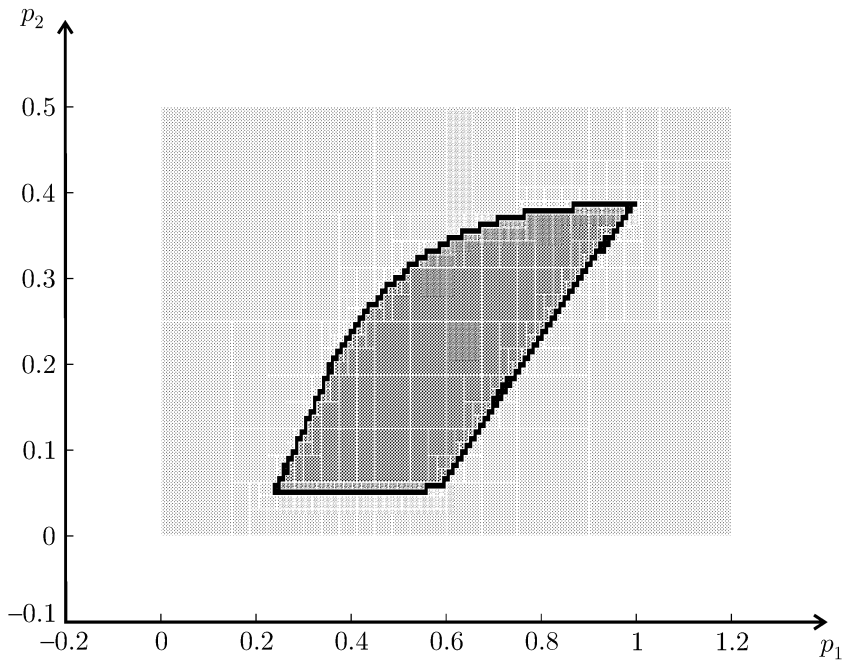


Fig. 6.15. Paving generated to bracket the posterior feasible set $\hat{\mathbb{P}}$ for the parameters of Example 6.11

6.3.5 Computation of the interval hull of the posterior feasible set

In Sections 6.3.2 to 6.3.4, parameter bounding was illustrated by a modified version of a problem treated in Milanese and Vicino (1991). The original problem was the computation of the interval hull $[\hat{\mathbb{P}}]$ of the posterior feasible set $\hat{\mathbb{P}}$ for the parameters. This becomes especially interesting if a more accurate outer approximation of the posterior feasible set cannot be obtained or turns out to be too complicated to be useful. In order to facilitate comparison, the problem considered in Milanese and Vicino (1991) will also serve here to illustrate the methodology.

Example 6.12 *Consider the same situation as in Example 6.8, except that* $n_{\mathrm{p}} = 4$ *(instead of* 2*) and the constrained set* $\mathbb{M}$ *is defined by the equations*

$$y_i = p_1 e^{-p_2 t_i} + p_3 e^{-p_4 t_i},\ i = 1, \ldots, 10. \tag{6.87}$$

The prior feasible box for the parameters is $[\check{\mathbf{p}}] = [2, 60] \times [0, 1] \times [-30, -1] \times [0, 0.5]$. *As in Example 6.5, the output data are given by (6.64) and the known measurement times are given by Table 6.5, but the radius of the* i*th interval output datum is now taken as* $\rho_i = 0.05|y_i| + 0.1$ *(instead of* $\rho_i = 0.5|y_i| + 1$ *in Example 6.5). The posterior feasible set is given by*

$$\widehat{\mathbb{P}} = \{\mathbf{p} \in \mathbb{R}^4 \mid (\boldsymbol{\phi}(\mathbf{p}), \mathbf{p}) \in [\check{\mathbf{y}}] \times [\check{\mathbf{p}}]\} = \boldsymbol{\phi}^{-1}([\check{\mathbf{y}}]) \cap [\check{\mathbf{p}}]. \tag{6.88}$$

Now, the set $\boldsymbol{\phi}^{-1}([\check{\mathbf{y}}])$ *can be defined by the following inequalities*

$$p_1 e^{-p_2 t_i} + p_3 e^{-p_4 t_i} \in [\check{y}_i],\ i = 1, \ldots, 10, \tag{6.89}$$

and thus Hull*, described in Chapter 5 at page 116, can be used to characterize* $[\hat{\mathbb{P}}]$. *For* $\varepsilon = 0.001$*, it takes less than* 8 *s on a* Pentium 133 *to find inner and outer boxes for* $[\hat{\mathbb{P}}]$*, given by*

$$[\mathbf{p}_{\mathrm{in}}] = [17.2, 26.79] \times [0.301, 0.49] \times [-16, -5.4] \times [0.0767, 0.1354],$$
$$[\mathbf{p}_{\mathrm{out}}] = [17.05, 27] \times [0.298, 0.495] \times [-16.2, -5.34] \times [0.0763, 0.136].$$

In Milanese and Vicino (1991) a signomial approach is proposed to solve this problem, based on Falk's algorithm (Falk, 1973). This approach yields less accurate results than the one advocated here and requires a computing time that is larger by an order of magnitude (Jaulin, 2000a). ■

6.4 State Bounding

6.4.1 Introduction

This section deals with the guaranteed estimation of the state vector of a non-linear discrete-time system in a bounded-error context. Readers unfamiliar with the concept of state may refer to Section 7.2, page 188, for a first introduction in a simpler linear framework. Consider a non-linear discrete-time system described by

$$\begin{cases} x_1(k) &= f_1(x_1(k-1), \ldots, x_{n_{\mathrm{x}}}(k-1), k), \\ \vdots & \vdots \ \vdots \\ x_{n_{\mathrm{x}}}(k) &= f_{n_{\mathrm{x}}}(x_1(k-1), \ldots, x_{n_{\mathrm{x}}}(k-1), k), \\ y_1(k) &= g_1(x_1(k), \ldots, x_{n_{\mathrm{x}}}(k), k), \\ \vdots & \vdots \ \vdots \\ y_{n_{\mathrm{y}}}(k) &= g_{n_{\mathrm{y}}}(x_1(k), \ldots, x_{n_{\mathrm{x}}}(k), k), \end{cases} \tag{6.90}$$

where k is the time index, ranging from 1 to $\bar{k}$, $x_1(k), \ldots, x_{n_x}(k)$ are the *state variables*, $y_1(k), \ldots, y_{n_y}(k)$ are the output variables and the f_is and the g_is are known functions, which may be represented by finite algorithms. For the sake of simplicity of exposition, input variables were not introduced, but they would pose no particular conceptual difficulties. In vector form, (6.90) can be written more concisely as

$$\begin{cases} \mathbf{x}(k) = \mathbf{f}(\mathbf{x}(k-1), k), \\ \mathbf{y}(k) = \mathbf{g}(\mathbf{x}(k), k). \end{cases} \tag{6.91}$$

The output vector $\mathbf{y}(k)$ is assumed to be measured on the system, and the problem to be considered is the estimation of the state $\mathbf{x}(k)$ from the information available. When the data are processed in real time, the data $\check{\mathbf{y}}(i)$ to be collected on the system after time k are not available during the estimation of $\mathbf{x}(k)$. The most common assumption is thus that only past measurements can be taken into account, but we shall also consider the off-line case where results of measurements posterior to time k can be used to estimate $\mathbf{x}(k)$.

In a linear context (*i.e.*, when $\mathbf{f}$ and $\mathbf{g}$ are linear), many tools are available for state estimation. They can be classified according to how they deal with uncertainty. Some of them do not take explicitly into account the fact that (6.91) is an approximation of reality and that the measurements are corrupted by noise. This is the case, for instance, for Luenberger state observers (Luenberger, 1966). Other estimators, such as the ubiquitous Kalman filter (Kalman, 1960; Sorenson, 1983), are based on a statistical description of uncertainty and assume that the measurement noise and state perturbations are realizations of random variables, with known statistical properties. The last group of methods corresponds to state bounding (Schweppe, 1968; Witsenhausen, 1968; Bertsekas and Rhodes, 1971; Chernousko, 1994; Durieu et al., 1996; Maksarov and Norton, 1996; Milanese et al., 1996; Kurzhanski and Valyi, 1997, and the references therein). These methods are based on the assumption that all uncertain variables belong to known compact sets, and attempt to build simple sets, such as ellipsoids, orthotopes or parallelotopes, guaranteed to contain all state vectors consistent with this assumption.

In a non-linear context, the methodology is far less developed, and still the subject of active research even in the deterministic case (Kang and Krener, 1998). When uncertainty is explicitly taken into account, this is most often by resorting to linearization. An extended Kalman filter (Gelb, 1974), based on linearization of (6.91) around the state trajectory, is usually employed. This linearization is inherently local and may fail to produce reliable estimates. It also makes any statistical interpretation of the covariance matrices computed by the algorithm questionable, because the propagation of the statistical properties of the perturbations through non-linear equations is largely unknown. In this section, a new non-linear state bounding approach will be presented, partly based on Kieffer et al. (1998, 1999) and Jaulin, Kieffer, Braems and Walter (2001).

The set of all the variables involved in (6.90) is

$$\begin{array}{l}\{\ x_1(0),\ldots,x_{n_{\rm x}}(0),\\ \quad x_1(1),\ldots,x_{n_{\rm x}}(1),\ y_1(1),\ldots,y_{n_{\rm y}}(1),\\ \quad \vdots\\ \quad x_1(\bar{k}),\ldots,x_{n_{\rm x}}(\bar{k}),\ y_1(\bar{k}),\ldots,y_{n_{\rm y}}(\bar{k})\ \}.\end{array} \tag{6.92}$$

Assume that $x_1(0),\ldots,x_{n_{\rm x}}(0)$ are known to belong to some prior bounded intervals $[\check{x}_1(0)],\ldots,[\check{x}_{n_{\rm x}}(0)]$. The interval $[\check{x}_i(0)]$ represents the prior knowledge on the initial state variable $x_i(0)$ and may be taken arbitrarily large in the absence of information. Let $\check{y}_i(k)$ be the result of the measure of the output variable $y_i(k)$. In a bounded-error context similar to that of Section 6.3, the measures performed at time k yield prior intervals $[\check{y}_i(k)]$ assumed to contain the actual values of the output variables $y_i(k)$. When estimation must be performed in real time, $[\check{y}_i(k)]$ is taken as $[-\infty,\infty]$ before measurement. The prior domains $[\check{x}_i(k)]$ for the state variables $x_i(k)$'s, $i=1,\ldots,n_{\rm x}$, $k=1,\ldots,\bar{k}$, are also taken as $[-\infty,\infty]$.

The initial state variables $x_1(0),\ldots,x_{n_{\rm x}}(0)$ have a special status, because if their values were known, then the values of all the other variables could be computed with the help of (6.90). For this reason, they will be called *source variables*, and $\mathbf{x}(0)$ will be called the *source vector*. Let $\mathbf{y}$ be the vector of all output variables

$$\mathbf{y}=(y_1(1),\ldots,y_{n_{\rm y}}(1),\ldots,y_1(\bar{k}),\ldots,y_{n_{\rm y}}(\bar{k}))^{\rm T}, \tag{6.93}$$

and $\boldsymbol{\phi}$ be the vector function that computes the values taken by $\mathbf{y}$ when (6.90) is simulated starting at $\mathbf{x}(0)$. We shall first consider the off-line estimation of the set $\hat{\mathbb{X}}(0)$ of all the initial vectors $\mathbf{x}(0)=(x_1(0),\ldots,x_{n_{\rm x}}(0))^{\rm T}$ that belong to the prior domain $[\check{\mathbf{x}}(0)]$ and are consistent with the prior domain for $\mathbf{y}$, *i.e.*, with

$$[\check{\mathbf{y}}]=[\check{y}_1(1)]\times\cdots\times\left[\check{y}_{n_{\rm y}}(1)\right]\times\cdots\times\left[\check{y}_1(\bar{k})\right]\cdots\times\left[\check{y}_{n_{\rm y}}(\bar{k})\right]. \tag{6.94}$$

This set is given by

$$\hat{\mathbb{X}}(0)=[\check{\mathbf{x}}(0)]\cap\boldsymbol{\phi}^{-1}\left([\check{\mathbf{y}}]\right). \tag{6.95}$$

Its characterization is thus a set-inversion problem, which can be solved by SIVIA, as illustrated in Section 6.4.2. Section 6.4.3 will extend the methodology to the estimation of all variables (and not just of the initial state). The use of a contractor based on forward–backward propagation will be shown to be particularly well suited to the context of state estimation. Finally, Section 6.4.4 will deal with the recursive case where only the past measurements can be taken into account.

6.4.2 Bounding the initial state

This section illustrates the use of SIVIA to characterize the set of all initial state vectors that are consistent with the prior interval data (Jaulin, 1994). Consider the non-linear state equation

$$\begin{cases} x_1(k+1) = \cos(x_1(k)x_2(k)), \\ x_2(k+1) = 3x_1(k) - \sin(x_2(k)), \\ y(k) \qquad = x_1^2(k) - x_2(k). \end{cases} \tag{6.96}$$

The ten output data

$$\begin{aligned} \check{\mathbf{y}} &= (y(0), \ldots, y(9))^{\mathrm{T}} \\ &= (3, -5, 0.6, 2.2, -3.8, -1.4, 0.4, -1.2, -1.8, 2.6)^{\mathrm{T}} \end{aligned} \tag{6.97}$$

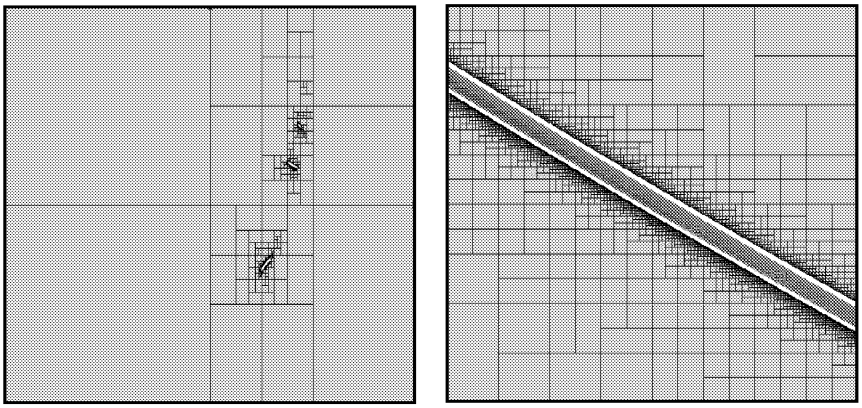
have been generated by simulating (6.96) for $k = 0, \ldots, 9$ from the unknown initial state vector $\mathbf{x}^*(0) = (2, 1)^{\mathrm{T}}$, and by adding a realization of some random noise in $[-0.5, 0.5]$ to the resulting output $y^*(k)$. The prior feasible box $[\check{\mathbf{y}}]$ for $\mathbf{y}$ was obtained by adding the error interval $[-0.5, 0.5]$ to all entries of $\check{\mathbf{y}}$. Thus

$$\begin{aligned} [\check{\mathbf{y}}] = \quad & [2.5, 3.5] \times [-5.5, -4.5] \times [0.1, 1.1] \times [1.7, 2.7] \times \\ & [-4.3, -3.3] \times [-1.9, -0.9] \times [-0.1, 0.9] \times \\ & [-1.7, -0.7] \times [-2.3, -1.3] \times [2.1, 3.1]. \end{aligned} \tag{6.98}$$

The set of all the initial state vectors $\mathbf{x}(0)$ in the prior box $[\check{\mathbf{x}}(0)]$ that are consistent with $[\check{\mathbf{y}}]$ is given by (6.95). For $\varepsilon = 0.01$ and $[\check{\mathbf{x}}(0)] = [-5, 5]^{\times 2}$, the characterization depicted in Figure 6.16a is obtained by SIVIA in less than 1 s on a PENTIUM 233. The zoom around the true value $\mathbf{x}^*(0)$ of the initial state presented on Figure 6.16b has been obtained for $\varepsilon = 0.0001$ and $[\check{\mathbf{x}}(0)] = [1.98, 2.02] \times [0.98, 1.02]$.

6.4.3 Bounding all variables

As already mentioned, if a punctual value $\mathbf{x}(0)$ is known for the initial state vector, the punctual values of all other variables can be calculated by simulating (6.90) from $\mathbf{x}(0)$. However, the approach presented in Section 6.4.2 to characterize the set $\hat{\mathbb{X}}(0)$ does not provide any bounds for the other variables of interest. Such bounds could be obtained by set simulation of (6.90) from $\hat{\mathbb{X}}(0)$, but this problem is much more difficult than set inversion and its consideration will be deferred to Section 6.4.4. For the time being, we shall consider a conceptually simpler approach. A slight adaptation of SIVIA described in Table 6.7 will first be used to get an outer approximation $\overline{\mathbb{X}}(0)$ of $\hat{\mathbb{X}}(0)$ consisting of a union of boxes, each of which has a width smaller than ε. The main difference with the version of Chapter 3, page 56, is that subboxes



(a) (b)

Fig. 6.16. Characterization of the set of all initial states consistent with interval data in $[-5,5]^{\times 2}$ (a) and in $[1.98,2.02]\times[0.98,1.02]$ (b)

of $[\mathbf{x}(0)]$ which could have been proved feasible are nevertheless bisected until their widths become smaller than the required accuracy parameter ε. As a result, more boxes are obtained upon completion of the algorithm than necessary for a given quality of the description of $\hat{\mathbb{X}}(0)$, but set simulation starting from these smaller boxes can be expected to be more accurate than with the characterization obtained with the initial version of SIVIA. In the algorithm of Table 6.7, $\mathcal{L}$ is a list of boxes in the space of the initial state, initialized as the empty list. The first call is made with $[\mathbf{x}(0)]=[\tilde{\mathbf{x}}(0)]$, where $[\tilde{\mathbf{x}}(0)]$ is the prior box for the initial state vector. $\mathcal{C}_{\hat{\mathbb{X}}(0)}$ is a suitable contractor for the solution set $\hat{\mathbb{X}}(0)$ (see Section 4.5, page 97). After completion of SIVIA, the union of all boxes in $\mathcal{L}$ is an outer approximation $\overline{\mathbb{X}}(0)$ of the set $\hat{\mathbb{X}}(0)=\boldsymbol{\phi}^{-1}([\tilde{\mathbf{y}}])\cap[\tilde{\mathbf{x}}(0)]$.

Table 6.7. Adaptation of SIVIA to set simulation

Algorithm SIVIA(in: $\mathcal{C}_{\hat{\mathbb{X}}(0)},[\mathbf{x}(0)],\varepsilon$; inout: $\mathcal{L}$)	
1	$[\mathbf{x}(0)]:=\mathcal{C}_{\hat{\mathbb{X}}(0)}([\mathbf{x}(0)])$;
2	if $([\mathbf{x}(0)]=\varnothing)$, then return;
3	if $(w([\mathbf{x}(0)])<\varepsilon)$ then $\mathcal{L}=\mathcal{L}\cup\{[\mathbf{x}(0)]\}$; return;
4	bisect $([\mathbf{x}(0)])$ into $[\mathbf{x}(0)]_1$ and $[\mathbf{x}(0)]_2$;
5	SIVIA$(\mathcal{C}_{\hat{\mathbb{X}}(0)},[\mathbf{x}(0)]_1,\varepsilon,\mathcal{L})$; SIVIA$(\mathcal{C}_{\hat{\mathbb{X}}(0)},[\mathbf{x}(0)]_2,\varepsilon,\mathcal{L})$.

The set simulator described in Table 6.8 can then be employed. It uses inclusion functions for $\mathbf{f}$ and $\mathbf{g}$ to compute boxes guaranteed to contain the

successive values of the state and output vectors for any initial state vector in any given box of $\overline{\mathbb{X}}(0)$ (listed in $\mathcal{L}$). It then suffices to take unions of boxes to get outer approximations of the set of all values of $\mathbf{x}(k)$ or $\mathbf{y}(k)$ for any $\mathbf{x}(0)$ in $\hat{\mathbb{X}}(0)$.

Table 6.8. Simulation of the state equations for all boxes of the list $\mathcal{L}$ produced by SIVIA of Table 6.7

Algorithm SETSIMULATION(in: $\mathcal{L}$; out $[\mathbf{x}(1)], \ldots, [\mathbf{x}(\bar{k})], [\mathbf{y}(1)], \ldots, [\mathbf{y}(\bar{k})]$)	
1	$[\mathbf{x}(1)] = \varnothing; \ldots; [\mathbf{x}(\bar{k})] = \varnothing;\ [\mathbf{y}(1)] = \varnothing; \ldots; [\mathbf{y}(\bar{k})] = \varnothing;$
2	for all $[\mathbf{x}'(0)]$ in $\mathcal{L}$
3	for $k = 1$ to $\bar{k}$
4	$[\mathbf{x}'(k)] := [\mathbf{f}]([\mathbf{x}'(k-1)]);$
5	$[\mathbf{x}(k)] := [\mathbf{x}(k)] \sqcup [\mathbf{x}'(k)];$
6	$[\mathbf{y}'(k)] := [\mathbf{g}]([\mathbf{x}'(k)]);$
7	$[\mathbf{y}(k)] := [\mathbf{y}(k)] \sqcup [\mathbf{y}'(k)].$

As an illustration, consider the non-linear discrete-time system

$$\begin{cases} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} = \begin{pmatrix} 0.1x_1(k-1) + x_2(k-1)\exp(x_1(k-1)) \\ x_1(k-1) + 0.1x_2^2(k-1) + \sin(k) \end{pmatrix}, \\ y(k) \qquad\quad = x_2(k)/x_1(k), \end{cases} \tag{6.99}$$

with $k \in \{1, \ldots, 15\}$. Interval data have been generated as follows. First, starting from the true value $\mathbf{x}^*(0) = (-1, 0)^{\mathrm{T}}$ of the initial state vector, the true values $\mathbf{x}^*(k)$ and $y^*(k), k \in \{1, \ldots, 15\}$ were computed by simulation. To each noise-free output $y^*(k)$ a random error was then added, with a uniform distribution in $[-e, e]$, to generate noisy data $\check{y}(k)$. Finally, the prior domain for $y(k)$ was taken equal to $[\check{y}(k)] = [\check{y}(k) - e, \check{y}(k) + e]$. $[\check{y}(k)]$ is thus guaranteed to contain the unknown noise-free output $y^*(k)$. The problem to be solved is then: *given the equations of the system (6.99), the interval data* $[\check{y}(k)]$, *and bounded intervals* $[\check{x}_1(0)]$ *and* $[\check{x}_2(0)]$ *containing the initial state variables* $x_1(0)$ *and* $x_2(0)$, *compute (accurate) interval enclosures for the values of the variables* $x_1(k)$, $x_2(k)$ *and* $y(k)$, $k = 1, \ldots, 15$.

The contractor for the set $\hat{\mathbb{X}}(0)$ required at Step 1 of Table 6.7 may be the forward–backward contractor $\mathcal{C}_{\downarrow\uparrow}$ described in Section 4.2.4, page 77, applied to the CSP $\mathcal{H} : (\boldsymbol{\phi}(\mathbf{x}(0)) - \mathbf{y} = 0, \mathbf{x}(0) \in [\check{\mathbf{x}}(0)], \mathbf{y} \in [\check{\mathbf{y}}])$. A first simulator $\boldsymbol{\phi}$ is given by Table 6.9. The statements of this simulator are transformed into primitive constraints by introducing auxiliary variables in a second simulator, as indicated in Table 6.10. The resulting contractor is in Table 6.11.

The prior domains for the components of the initial state vector were taken as

$$[\check{x}_1(0)] = [-1.2, -0.8], \qquad [\check{x}_2(0)] = [-0.2, 0.2]. \tag{6.100}$$

Table 6.9. Simulator

Algorithm ϕ(in: $x_1(0), x_2(0)$; out: $y(1), \ldots, y(\bar{k})$)
for $k := 1$ to $\bar{k}$,
$\quad x_1(k) := 0.1 * x_1(k-1) + x_2(k-1) * \exp(x_1(k-1))$;
$\quad x_2(k) := x_1(k-1) + 0.1 * x_2^2(k-1) + \sin(k)$;
$\quad y(k) := x_2(k)/x_1(k)$.

Table 6.10. Simulator with auxiliary variables

Algorithm ϕ(in: $x_1(0), x_2(0)$; out: $y(1), \ldots, y(\bar{k})$)
for $k := 1$ to $\bar{k}$,
$\quad z_1(k) := \exp(x_1(k-1))$;
$\quad z_2(k) = x_2(k-1) * z_1(k)$;
$\quad x_1(k) := 0.1 * x_1(k-1) + z_2(k)$;
$\quad z_3(k) := 0.1 * \mathrm{sqr}(x_2(k-1))$;
$\quad z_4(k) := z_3(k) + \sin(k)$;
$\quad x_2(k) := x_1(k-1) + z_4(k)$;
$\quad y(k) := x_2(k)/x_1(k)$.

In the absence of noise (*i.e.*, $e = 0$), the contractor of Table 6.11 is able to find the actual values of all the variables with an accuracy of 8 digits in 0.1 s on a PENTIUM 133. *No* bisection turned out to be necessary to get this result. The boxes drawn on the left part of Figure 6.17 are those obtained after each iteration of the contractor $\mathcal{C}_{\downarrow\uparrow}$. More details about the resolution of this example with the methodology advocated here can be found in Jaulin, Braems, Kieffer and Walter (2001).

For $e = 0.5$ (*i.e.*, in the presence of noise), the volume of $\hat{\mathbb{X}}(0)$ is no longer equal to zero, and thus, even with an ideal contractor, bisections have to be performed (see the right part of Figure 6.17). Computing time is now about 3 s for $\varepsilon = 0.001$, on a PENTIUM 133. The prior interval data $[\check{y}(k)]$ are on the left part of Figure 6.18 and the corresponding contracted intervals $[\hat{y}(k)]$ are on the right part of the same figure. Since $x_1^*(5)$ and $x_1^*(13)$ are almost equal to zero and since $y(k) = x_2(k)/x_1(k)$, no contraction was achieved for $[y(5)]$ and $[y(13)]$. Figure 6.19 presents the contracted domains obtained for the state variables $x_1(k)$ (left) and $x_2(k)$ (right), as functions of k.

6.4.4 Bounding by constraint propagation

When there are many variables in $\mathcal{X}$, as in long-range state estimation, one should avoid bisecting boxes in the Cartesian product of the domains of all

Table 6.11. Forward–backward contractor for the set $\hat{\mathbb{X}}(0)$ of initial state vectors

Algorithm $\mathcal{C}_{\hat{\mathbb{X}}(0)}$(in: $[\check{y}(1)], \ldots, [\check{y}(\bar{k})]$; inout: $[x_1(0)], [x_2(0)]$)

1 for $k := 1$ to $\bar{k}$
2 $[x_1(k)] := [-\infty, \infty]$; $[x_2(k)] := [-\infty, \infty]$;
3 $[z_1(k)] := [-\infty, \infty]$; $[z_2(k)] := [-\infty, \infty]$;
4 $[z_3(k)] := [-\infty, \infty]$; $[z_4(k)] := [-\infty, \infty]$;
5 do
6 for $k := 1$ to $\bar{k}$, // forward
7 $[z_1(k)] := [z_1(k)] \cap \exp([x_1(k-1)])$;
8 $[z_2(k)] := [z_2(k)] \cap ([x_2(k-1)] * [z_1(k)])$;
9 $[x_1(k)] := [x_1(k)] \cap (0.1 * [x_1(k-1)] + [z_2(k)])$;
10 $[z_3(k)] := [z_3(k)] \cap (0.1 * \mathrm{sqr}([x_2(k-1)]))$;
11 $[z_4(k)] := [z_4(k)] \cap ([z_3(k)] + \sin(k))$;
12 $[x_2(k)] := [x_2(k)] \cap ([x_1(k-1)] + [z_4(k)])$;
13 $[y(k)] := [y(k)] \cap ([x_2(k)]/[x_1(k)])$;
14 for $k := \bar{k}$ down to 1, // backward
15 $[x_2(k)] := [x_2(k)] \cap ([y(k)] * [x_1(k)])$;
16 $[x_1(k)] := [x_1(k)] \cap ([x_2(k)]/[y(k)])$;
17 $[x_1(k-1)] := [x_1(k-1)] \cap ([x_2(k)] - [z_4(k)])$;
18 $[z_4(k)] := [z_4(k)] \cap ([x_2(k)] - [x_1(k-1)])$;
19 $[z_3(k)] := [z_3(k)] \cap ([z_4(k)] - \sin(k))$;
20 $[x_2(k-1)] := [x_2(k-1)] \cap \left(0.1 * \mathrm{sqr}^{-1}([z_3(k)])\right)$;
21 $[x_1(k-1)] := [x_1(k-1)] \cap (10 * ([x_1(k)] - [z_2(k)]))$;
22 $[z_2(k)] := [z_2(k)] \cap ([x_1(k)] - 0.1 * [x_1(k-1)])$;
23 $[x_2(k-1)] := [x_2(k-1)] \cap ([z_2(k)]/[z_1(k)])$;
24 $[z_1(k)] := [z_1(k)] \cap ([z_2(k)]/[x_2(k-1)])$;
25 $[x_1(k-1)] := [x_1(k-1)] \cap \log([z_1(k)])$;
26 while contraction is significant.

variables, in order to avoid a combinatorial explosion of complexity. Two approaches may be considered for this purpose.

The first one is based on selecting a set of *source* variables, as small as possible, such that the value of all the other variables can be computed in a unique way from the knowledge of the values of these source variables by using the available constraints. This is the approach followed in the previous two sections, where the source variables were $x_1(0), \ldots, x_{n_x}(0)$. The source variables were stored in a single *source vector* ($\mathbf{x}(0)$ in Sections 6.4.2 and 6.4.3) and search was performed in the *source space* to which this source vector belongs, using a set-inversion technique.

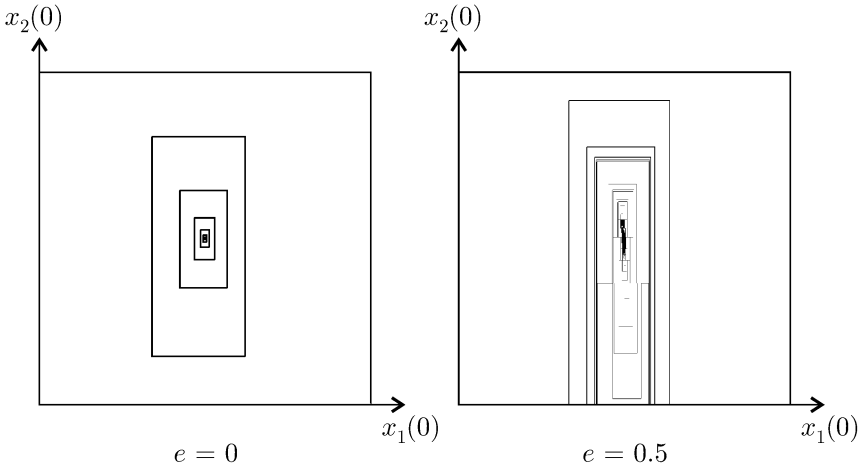


Fig. 6.17. Left: contractions generated in a noise-free context; right: contractions and bisections generated in a noisy context; the two frames are $[-1.2, -0.8] \times [-0.2, 0.2]$ in the $(x_1(0), x_2(0))$-space

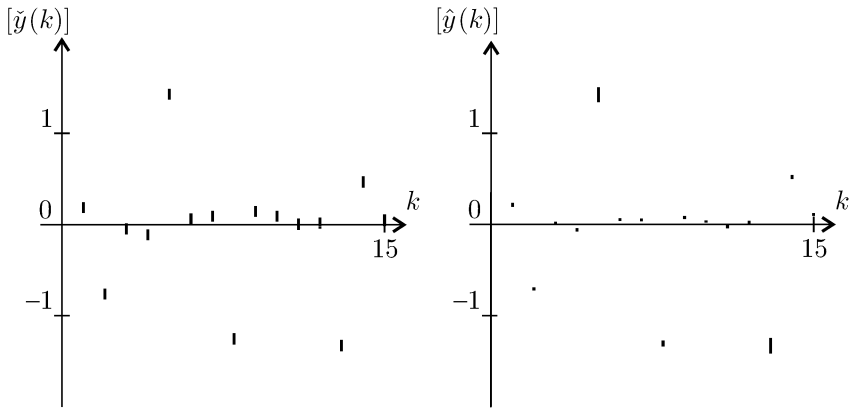


Fig. 6.18. In the presence of noise ($e = 0.5$), interval output data $[\check{y}(k)]$ (left) and contracted interval outputs $[\hat{y}(k)]$ containing $y^*(k)$ obtained by taking the constrained set into account (right)

The *clustering approach* (Dechter and Dechter, 1987; Dechter and Pearl, 1989; Meiri et al., 1990; Gyssens et al., 1994; Seybold et al., 1998) to be considered now partitions $\mathcal{X}$ into groups of variables to form vectors, whenever possible. The constraints of $\mathbb{M}$ are then transformed into constraints on these vectors. The variables in $\mathcal{X}$ should be grouped in such a way that the constraint graph is a tree (*i.e.*, does not contain cycles). The principle of the approach will first be illustrated on a simple problem.

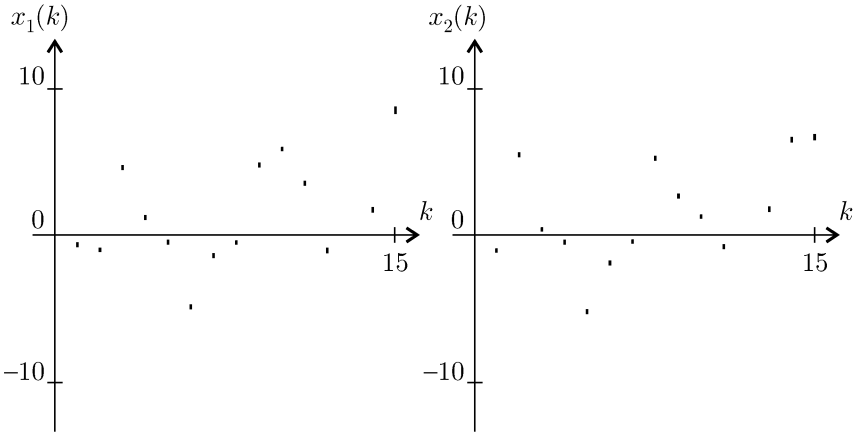


Fig. 6.19. In the presence of noise ($e = 0.5$), contracted domains for $x_1(k)$ (left) and $x_2(k)$ (right) as functions of k

Example 6.13 *Consider the set of variables* $\mathcal{X} = \{x_1, \ldots, x_7\}$*, with the constraints*

$$\begin{aligned} &x_1 e^{x_2} + x_3 \leqslant 2, \\ &x_1 x_3 = 5, \\ &x_2 - x_3 = 0, \\ &x_2 - x_3\sqrt{x_4} = 7, \\ &x_2 + x_3 x_5 e^{x_2} = 1, \\ &x_5 + \sin(x_6 x_7) = 0. \end{aligned} \tag{6.101}$$

The constraint graph *is depicted on Figure 6.20a. A link between two variables* x_i *and* x_j *means that there exists a constraint involving both. This graph is not a tree because it contains cycles. A (simple but not very efficient) heuristic to group variables in order to transform the constraint graph into a tree is as follows.*

1. *Select all cycles with length equal to* ℓ *(at the beginning,* ℓ *is taken equal to three, but if no such cycles are found, take* $\ell = 4, 5 \ldots$*). In this example, there are four cycles with length three, namely* (x_1, x_2, x_3)*,* (x_2, x_3, x_4)*,* (x_2, x_3, x_5) *and* (x_5, x_6, x_7)*.*
2. *Pool the variables associated with the arc that belongs to the maximum number of selected cycles into a single vector. Here, the arc is* (x_2, x_3)*, which belongs to the first three cycles, and* x_2 *and* x_3 *are pooled into the vector* $(x_2, x_3)^T$*. The graph of Figure 6.20b is thus obtained. These two steps are repeated until the graph becomes a tree, depicted in Figure 6.20c.* ■

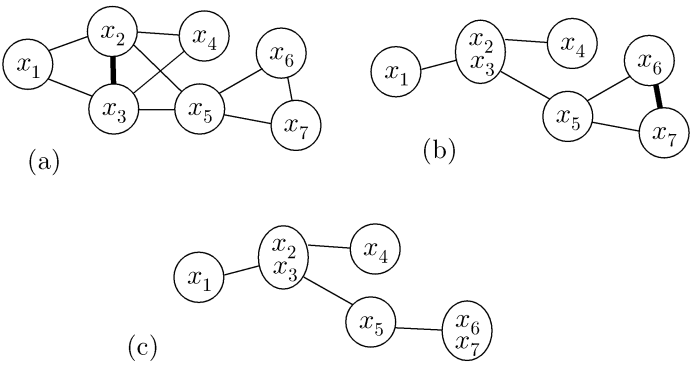


Fig. 6.20. Clustering variables to transform a graph with cycles into a tree

After grouping the variables in such a way that the constraint graph is a tree, we have a finite set $\mathcal{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ of vector variables $\mathbf{v}_i$ with dimensions $d_i \in \mathbb{N}$, and prior domains $\check{\mathbb{V}}_i \subset \mathbb{R}^{d_i}, i \in \{1, \ldots, n\}$. Some vector variables, say $\mathbf{v}_i$ and $\mathbf{v}_j$, are related by *binary constraints* $\mathbb{C}_{i,j}$, which can be viewed as subsets of $\mathbb{R}^{d_i} \times \mathbb{R}^{d_j}$. There may also exist *unary constraints* $\mathbb{C}_i$ expressing relations between components of $\mathbf{v}_i$. The prior domain $\check{\mathbb{V}}_i$ for $\mathbf{v}_i$ is then obtained by intersecting $\mathbb{C}_i$, which can be viewed as a subset of $\mathbb{R}^{d_i}$, with the Cartesian product of the prior domains of the components of $\mathbf{v}_i$.

Example 6.14 *Consider again the problem of Example 6.13. The set of vector variables* $\mathcal{V}$ *is* $\{v_1, \mathbf{v}_2, v_3, v_4, \mathbf{v}_5\}$*, where* $v_1 = x_1, \mathbf{v}_2 = (x_2, x_3)^T$*,* $v_3 = x_4, v_4 = x_5$ *and* $\mathbf{v}_5 = (x_6, x_7)^T$*. There is only one unary constraint, given by*

$$\mathbb{C}_2 = \left\{\mathbf{v}_2 \in \mathbb{R}^2, x_2 - x_3 = 0\right\}. \tag{6.102}$$

If the prior domains for x_2 *and* x_3 *are denoted by* $\check{\mathbb{X}}_2$ *and* $\check{\mathbb{X}}_3$ *respectively, the prior domain for* $\mathbf{v}_2$ *is thus*

$$\check{\mathbb{V}}_2 = \left\{\mathbf{v}_2 \in \check{\mathbb{X}}_2 \times \check{\mathbb{X}}_3 \mid x_2 - x_3 = 0\right\}. \tag{6.103}$$

The binary constraints between these vector variables are

$$\begin{aligned}
\mathbb{C}_{1,2} &= \left\{(v_1, \mathbf{v}_2) \in \mathbb{R}^3 \mid x_1 e^{x_2} + x_3 \leqslant 2 \text{ and } x_1 x_3 = 5\right\}, \\
\mathbb{C}_{2,3} &= \left\{(\mathbf{v}_2, v_3) \in \mathbb{R}^3 \mid x_2 - x_3\sqrt{x_4} = 7\right\}, \\
\mathbb{C}_{2,4} &= \left\{(\mathbf{v}_2, v_4) \in \mathbb{R}^3 \mid x_2 + x_3 x_5 e^{x_2} = 1\right\}, \\
\mathbb{C}_{4,5} &= \left\{(v_4, \mathbf{v}_5) \in \mathbb{R}^3 \mid x_5 + \sin(x_6 x_7) = 0\right\}.
\end{aligned} \tag{6.104}$$

■

In the context of state estimation, unary constraints can be avoided and the constraint $\mathbb{C}_{i,j}$ can generally be put into the form

$$\mathbb{C}_{i,j} = \left\{(\tilde{\mathbf{v}}_i, \tilde{\mathbf{v}}_j) \in \mathbb{R}^{d_i} \times \mathbb{R}^{d_j} \mid \tilde{\mathbf{v}}_j = \mathbf{f}_j\left(\tilde{\mathbf{v}}_i\right)\right\}. \tag{6.105}$$

To simplify notation, we shall then refer to it as $\mathbb{C}_{i,j} : \mathbf{v}_j = \mathbf{f}_j(\mathbf{v}_i)$.

The value taken by $\mathbf{v}_i$ is *consistent* (with the constraints) if

$$\begin{aligned}\exists(\tilde{\mathbf{v}}_1 \in \check{\mathbb{V}}_1, \ldots, \tilde{\mathbf{v}}_{i-1} \in \check{\mathbb{V}}_{i-1}, \tilde{\mathbf{v}}_{i+1} \in \check{\mathbb{V}}_{i+1}, \ldots, \tilde{\mathbf{v}}_n \in \check{\mathbb{V}}_n) \mid \\ (\tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_{i-1}, \mathbf{v}_i, \tilde{\mathbf{v}}_{i+1}, \ldots, \tilde{\mathbf{v}}_n) \text{ satisfies all the constraints.}\end{aligned} \tag{6.106}$$

The set of all consistent $\mathbf{v}_i$s is called the *consistency domain* associated with the variable $\mathbf{v}_i$, and denoted by $\hat{\mathbb{V}}_i$. Consider two variables $\mathbf{v}_i$ and $\mathbf{v}_j$ related by a constraint $\mathbb{C}_{i,j}$. Define the *local contractor* of the domain $\mathbb{V}_i$ with respect to the variable $\mathbf{v}_j$ by

$$\rho_j\left(\mathbb{V}_i\right) = \left\{\tilde{\mathbf{v}}_i \in \mathbb{V}_i \mid \exists \tilde{\mathbf{v}}_j \in \mathbb{V}_j, (\tilde{\mathbf{v}}_i, \tilde{\mathbf{v}}_j) \in \mathbb{C}_{i,j}\right\}. \tag{6.107}$$

Figure 6.21 illustrates this definition. The adjective *local* indicates that only one constraint is taken into account. Note that $\hat{\mathbb{V}}_i \subset \rho_j\left(\check{\mathbb{V}}_i\right) \subset \check{\mathbb{V}}_i$. If the binary constraint is $\mathbb{C}_{i,j} : \mathbf{v}_j = \mathbf{f}_j\left(\mathbf{v}_i\right)$, then

$$\begin{aligned}\rho_j\left(\mathbb{V}_i\right) &= \mathbb{V}_i \cap \mathbf{f}_j^{-1}\left(\mathbb{V}_j\right), \\ \rho_i\left(\mathbb{V}_j\right) &= \mathbb{V}_j \cap \mathbf{f}_j\left(\mathbb{V}_i\right).\end{aligned} \tag{6.108}$$

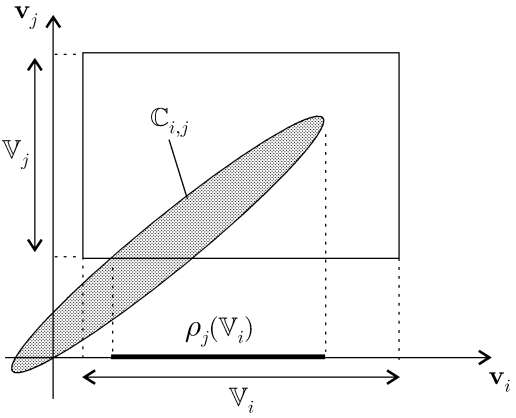


Fig. 6.21. Local contractor

The consistency domains $\hat{\mathbb{V}}_i$ can be obtained by extending to vector variables (Jaulin, Kieffer, Braems and Walter, 2001) the *forward–backward propagation algorithm* proposed in Benhamou et al. (1999) for variables of $\mathbb{R}$. First, one node is selected in the tree to become its root. The tree is then scanned from its leaves down to its root. This is the *forward propagation step*. During this scan, the domains of the nodes are contracted with respect to their

children. As a result, the domain associated with the root is its consistency domain. Finally, the tree is scanned from its root up to its leaves. This is the *backward propagation step*. During this second scan, the domains of the nodes are contracted with respect their fathers. Upon completion, each domain is equal to the corresponding consistency domains (*i.e.*, for any $i, \mathbb{V}_i = \hat{\mathbb{V}}_i$). A proof can be found in Jaulin, Kieffer, Braems and Walter (2001). Note that the same idea for discrete domains can be found in Freuder (1978), Montanari and Rossi (1991) and Collin et al. (1991).

Example 6.15 *Consider the tree of Figure 6.22. Its root is* $\mathbf{v}_1$ *and its leaves are* $\mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6, \mathbf{v}_7, \mathbf{v}_8$. *Forward propagation consists of the following sequence of set operations (initially, for any* i, $\mathbb{V}_i := \check{\mathbb{V}}_i$*):*

$$\begin{array}{l} \mathbb{V}_2 := \mathbb{V}_2 \cap \rho_4(\mathbb{V}_2) \cap \rho_5(\mathbb{V}_2) \cap \rho_6(\mathbb{V}_2); \\ \mathbb{V}_3 := \mathbb{V}_3 \cap \rho_7(\mathbb{V}_3) \cap \rho_8(\mathbb{V}_3); \\ \mathbb{V}_1 := \mathbb{V}_1 \cap \rho_2(\mathbb{V}_1) \cap \rho_3(\mathbb{V}_1). \end{array} \tag{6.109}$$

At this stage, $\mathbb{V}_1 = \hat{\mathbb{V}}_1$. *Backward propagation consists in computing*

$$\begin{array}{l} \mathbb{V}_2 := \mathbb{V}_2 \cap \rho_1(\mathbb{V}_2); \\ \mathbb{V}_4 := \mathbb{V}_4 \cap \rho_2(\mathbb{V}_4); \\ \mathbb{V}_5 := \mathbb{V}_5 \cap \rho_2(\mathbb{V}_5); \\ \mathbb{V}_6 := \mathbb{V}_6 \cap \rho_2(\mathbb{V}_6); \\ \mathbb{V}_3 := \mathbb{V}_3 \cap \rho_1(\mathbb{V}_3); \\ \mathbb{V}_7 := \mathbb{V}_7 \cap \rho_3(\mathbb{V}_7); \\ \mathbb{V}_8 := \mathbb{V}_8 \cap \rho_3(\mathbb{V}_8). \end{array} \tag{6.110}$$

Now, for all $i, \mathbb{V}_i = \hat{\mathbb{V}}_i$. ■

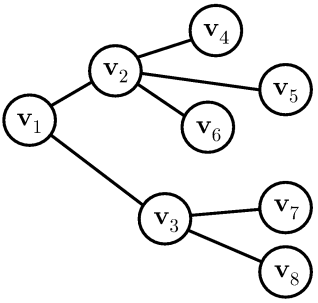


Fig. 6.22. Tree of Example 6.15

Let us apply this methodology to the estimation of the state of the discrete-time system

$$\begin{cases} \mathbf{x}(k) = \mathbf{f}(\mathbf{x}(k-1)), \\ \mathbf{y}(k) = \mathbf{g}(\mathbf{x}(k)), \end{cases} \qquad k = 1, \ldots, \bar{k}, \tag{6.111}$$

where $\mathbf{x}(k) \in \mathbb{R}^n$ is the state vector and $\mathbf{y}(k) \in \mathbb{R}^m$ is the output vector. Equation (6.111) is a special case of (6.90), which could be treated in its general form along the same lines. The functions $\mathbf{f}$ and $\mathbf{g}$ may be non-linear. Two types of estimators will be considered, namely *causal estimators* where the estimate of the state at time k can only be based on measurements of the output up to time k, and *non-causal estimators* where all measurements of the output vector are available at the outset. On-line estimation can only use causal estimators, but off-line estimation allows one to use non-causal estimators and to take advantage of all the available data to estimate the state vector at any given time.

Causal estimator. At time k, the set of all vector variables to be considered is

$$\mathcal{V}_k = \{\mathbf{x}(0), \ldots, \mathbf{x}(k), \mathbf{y}(1), \ldots, \mathbf{y}(k)\}\,. \tag{6.112}$$

The set of the associated prior domains is

$$\mathcal{D}_k = \left\{\check{\mathbb{X}}(0), \ldots, \check{\mathbb{X}}(k), \check{\mathbb{Y}}(1), \ldots, \check{\mathbb{Y}}(k)\right\}. \tag{6.113}$$

For simplicity, assume that no specific prior information on $\mathbf{x}(1), \ldots, \mathbf{x}(k)$ is available. Thus $\check{\mathbb{X}}(1), \ldots, \check{\mathbb{X}}(k)$ are each taken as $\mathbb{R}^n$. The past measurements $\check{\mathbf{y}}(i)$, $0 \leq i \leq k$ are used to form the prior feasible domains $\check{\mathbb{Y}}(i)$ based on their reliability. The set of the constraints involved is

$$\begin{aligned} \mathcal{C}_k = &\{\mathbf{x}(\ell) = \mathbf{f}(\mathbf{x}(\ell-1));\ \ell = 1, \ldots, k\} \\ &\cup \{\mathbf{y}(\ell) = \mathbf{g}(\mathbf{x}(\ell));\ \ell = 1, \ldots, k\}\,. \end{aligned} \tag{6.114}$$

The corresponding graph is represented on Figure 6.23. Despite the presence of arrows, this graph is *not* oriented, and these arrows are only meant to indicate the directions along which the associated functions operate.

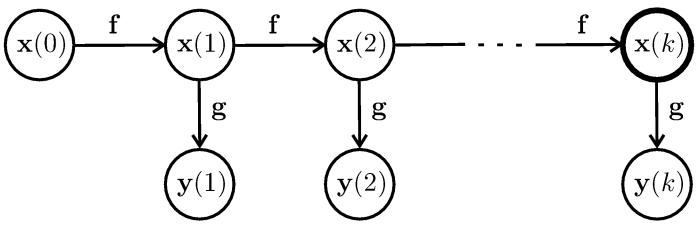


Fig. 6.23. Graph associated with (6.111); the bold circle indicates the root selected $\mathbf{x}(k)$, but any other root could be chosen

Forward propagation applied to the graph of Figure 6.23 computes the consistency domain $\hat{\mathbb{X}}(k)$ for $\mathbf{x}(k)$, provided that $\mathbf{x}(k)$ is chosen as the root of the tree. This results in the causal state estimator (CSE) of Table 6.12.

Table 6.12. Causal state estimator based on forward propagation

Algorithm CSE(in: $\check{\mathbb{X}}(0), \check{\mathbb{Y}}(1), \ldots, \check{\mathbb{Y}}(k)$; out: $\hat{\mathbb{X}}(k)$)	
1	$\mathbb{X}(0) := \check{\mathbb{X}}(0)$;
2	for $\ell := 1$ to k,
3	$\quad \mathbb{X}(\ell) := \mathbf{f}(\mathbb{X}(\ell-1)) \cap \mathbf{g}^{-1}(\check{\mathbb{Y}}(\ell))$;
4	$\hat{\mathbb{X}}(k) := \mathbb{X}(k)$.

At time $k+1$, the measurement $\check{\mathbf{y}}(k+1)$ becomes available and the algorithm of Table 6.12 can be used again to compute $\hat{\mathbb{X}}(k+1)$. One should obviously rather use the recursive causal state estimator (RCSE) of Table 6.13 (Kieffer et al., 1998, 1999), to take advantage of the results already obtained at time k. This algorithm performs an optimal contraction of the domain for $\mathbf{x}(k)$ after the measurement at time k.

Table 6.13. Recursive causal set state estimator

Algorithm RCSE(in: $\check{\mathbb{X}}(0)$; out: $\hat{\mathbb{X}}(1), \ldots, \hat{\mathbb{X}}(\overline{k})$)	
1	$\hat{\mathbb{X}}(0) := \check{\mathbb{X}}(0)$; $k = 1$;
2	for $k := 1$ to $\overline{k}$,
3	$\quad$ wait for $\check{\mathbb{Y}}(k)$;
4	$\quad \hat{\mathbb{X}}(k) := \mathbf{f}(\hat{\mathbb{X}}(k-1)) \cap \mathbf{g}^{-1}(\check{\mathbb{Y}}(k))$.

Figure 6.24 illustrates the principle of one iteration of RCSE. At time $k-1$, the state is known to belong to $\hat{\mathbb{X}}(k-1)$. The *predicted set* $\mathbf{f}(\hat{\mathbb{X}}(k-1))$ thus contains all possible values of the state at time k. When a measurement becomes available at time k, $\mathbf{g}^{-1}(\check{\mathbb{Y}}(k))$ contains all state vectors that could have led to a measured output belonging to $\check{\mathbb{Y}}(k)$. Thus, at time k, the state belongs to the *corrected set* $\mathbf{f}(\hat{\mathbb{X}}(k-1)) \cap \mathbf{g}^{-1}(\check{\mathbb{Y}}(k))$. It is not required that $\hat{\mathbb{X}}(k-1)$ or $\mathbf{f}(\hat{\mathbb{X}}(k-1))$ consist of a single connected component, as will be seen in Section 8.4.6, page 263, when tracking a moving robot.

Non-causal estimator. Assume now that the $\overline{k}$ output measurements $\check{\mathbf{y}}(k)$, $k \in \{1, \ldots, \overline{k}\}$, and the associated prior feasible domains $\check{\mathbb{Y}}(k)$ are all available at the outset. Forward–backward propagation then makes it possible to obtain posterior feasible domains that are consistent with past and future information. Table 6.14 describes the resulting procedure NCSE (for non-causal state estimator) when the root is taken as $\mathbf{x}(\overline{k})$.

Upon completion of the algorithm, $\hat{\mathbb{X}}(k)$ is the consistency domain for $\mathbf{x}(k)$ and $\hat{\mathbb{Y}}(k)$ the consistency domain for $\mathbf{y}(k)$, and the latter contains the actual output $\mathbf{y}^*(k)$ with a better accuracy than the prior domain $\check{\mathbb{Y}}(k)$ (provided, of course, that the constrained set and prior domains have been properly chosen).

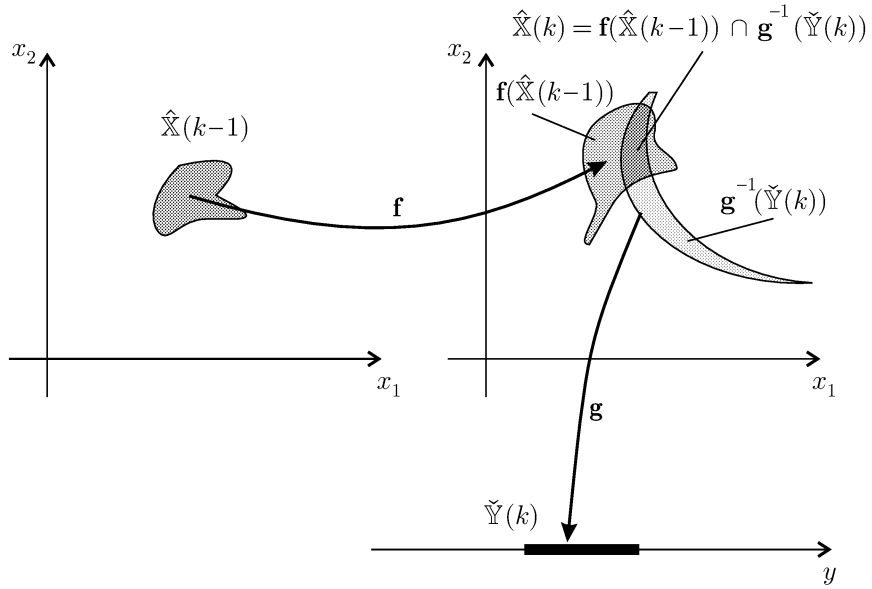


Fig. 6.24. Principle of one iteration of the recursive causal state estimator (RCSE)

Table 6.14. Non-causal state estimator based on forward–backward propagation

Algorithm: NCSE(in: $\check{\mathbb{X}}(0), \check{\mathbb{Y}}(1), \ldots, \check{\mathbb{Y}}(\bar{k})$; out: $\hat{\mathbb{X}}(0), \ldots, \hat{\mathbb{X}}(\bar{k})$)	
1	$\hat{\mathbb{X}}(0) := \check{\mathbb{X}}(0)$;
2	for $k := 1$ to $\bar{k}$, // forward
3	$\quad \hat{\mathbb{X}}(k) := \mathbf{f}(\hat{\mathbb{X}}(k-1)) \cap \mathbf{g}^{-1}(\check{\mathbb{Y}}(k))$;
4	for $k := \bar{k}$ down to 1, // backward
5	$\quad \hat{\mathbb{Y}}(k) := \mathbf{g}(\hat{\mathbb{X}}(k))$; $\hat{\mathbb{X}}(k-1) := \hat{\mathbb{X}}(k-1) \cap \mathbf{f}^{-1}(\hat{\mathbb{X}}(k))$.

Remark 6.6 *The implementation of these estimators requires a representation for sets and an implementation of the local contractor ρ_i. A domain $\mathbb{V}$ is represented by a subpaving that encloses it. In the present context, the local contractor ρ_i corresponds either to the image $\mathbf{f}(\mathbb{V})$ of a set $\mathbb{V}$ by a vector function $\mathbf{f}$ or to the reciprocal image $\mathbf{f}^{-1}(\mathbb{W})$ of a set $\mathbb{W}$ by a vector function $\mathbf{f}$. A guaranteed enclosure of $\mathbf{f}(\mathbb{V})$ can be computed by* ImageSp *(see Chapter 3, page 60) and a guaranteed enclosure of $\mathbf{f}^{-1}(\mathbb{W})$ can be computed by* Sivia *(see Chapter 3, page 56).* ■

Illustration. Consider the non-linear system

$$\begin{cases} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} = 3 \begin{pmatrix} \sin(x_1(k-1) + x_2(k-1)) \\ \cos(x_1(k-1) + x_2(k-1)) \end{pmatrix}, \\ y(k) \quad = |x_1(k)|, \end{cases} \tag{6.115}$$

with $k \in \{1, \ldots, 10\}$. For $\mathbf{x}^*(0) = (0 \;\; 0)^{\mathrm{T}}$, the true values $\mathbf{x}^*(k)$ and $y^*(k), k \in \{1, \ldots, 10\}$, have been generated by simulation of this system. The output data $\check{y}(k)$ have then been obtained by adding a bounded noise to the noise-free output $y^*(k)$

$$\check{y}(k) = y^*(k) + n(k), \tag{6.116}$$

where $n(k)$ is a realization of a random variable uniformly distributed in $[-0.1, 0.1]$. The prior domain for the kth output is then taken as

$$\check{\mathbb{Y}}(k) = \check{y}(k) + [-0.1, 0.1], \tag{6.117}$$

so

$$y^*(k) \in \check{\mathbb{Y}}(k), \;\; k = 1, \ldots, 10. \tag{6.118}$$

The prior domain for $\mathbf{x}(k)$ is taken as $\check{\mathbb{X}}(k) = \mathbb{R}^2$. The posterior domains obtained for $\mathbf{x}(k)$ by causal and non-causal estimation are depicted on Figure 6.25. The total computing time for both estimators is less than one minute on a PENTIUM 133. The frames of all subfigures are $[-4, 4] \times [-4, 4]$. The first subfigure is entirely grey, which illustrates the obvious fact that the causal estimator is unable to provide any information about $\mathbf{x}(0)$ ($\check{\mathbb{X}}(0) = \mathbb{R}^2$), in contrast to the non-causal estimator. The last two subfigures, for $k = 10$, are identical because both estimators have now processed the same information.

6.5 Conclusions

Estimation problems involve uncertain variables related by constraints. These constraints are used to decrease the uncertainty in the variables, and advantage may thus be taken of the tools available for the solution of constraint satisfaction problems (CSPs). This chapter has shown how the formalism and algorithms of CSPs can be adapted to estimation. A unified framework has been proposed for a large class of estimation problems, and the efficiency of the interval solvers of Chapter 5 has been demonstrated in this context.

It is now possible to obtain guaranteed estimates of parameters or state variables, even when these parameters are not identifiable or when these state variables are not observable. Non-linear and time-varying constraints are easily handled. The approach could be extended to the case where $\mathcal{X}$ consists of infinitely many variables, some of which may for instance be integer or Boolean variables.

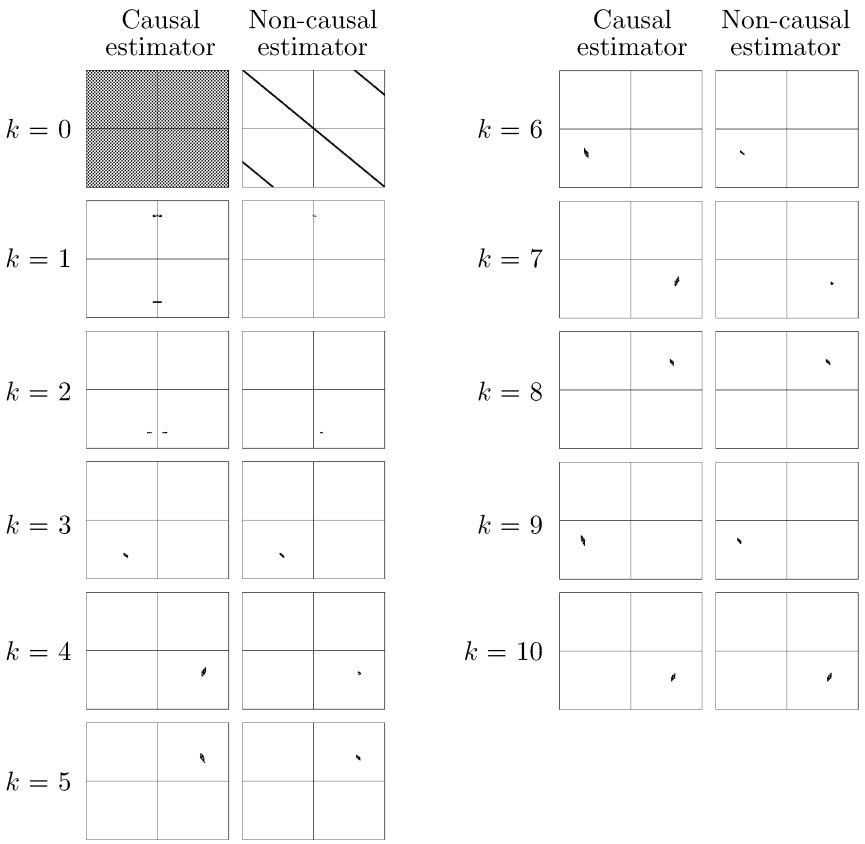


Fig. 6.25. Causal and non-causal set estimates

Least-square estimation suffers from the fact that the cost function to be minimized is a sum of terms involving the same parameters, so multi-occurence of these parameters is unavoidable and tends to make inclusion functions for the cost function very pessimistic, which complicates the elimination of uninteresting parts of the search domain. Parameter and state bounding are comparatively easier to implement, and interval constraint propagation contributes to allowing the treatment of high-dimensional problems.

7. Robust Control

7.1 Introduction

The aim of this chapter is to illustrate the use of interval techniques presented in Part II to solve some robust control problems. Robustness is understood here with respect to uncertainty in the model of the process to be controlled. The problems considered range from the analysis of the properties of an existing uncertain system to the design of a controller for an uncertain process.

Only systems described by linear time-invariant differential equations will be considered. The main reason is that checking stability then amounts to checking non-linear inequalities. The control of systems defined by non-linear differential or difference equations is much more complicated. A first approach based on interval methods for the control of non-linear difference equations has been proposed in Jaulin and Walter (1997).

Robust control has been a very active field of research for many years (Horowitz, 1963; Dorato et al., 1993; Kwakernaak, 1993; Barmish, 1994; Boyd et al., 1994; Francis and Khargonekar, 1995) and it is impossible to present the state of the art in any detail in a single chapter. We shall focus attention on robust stability and robust control problems for uncertain systems that can be described by parametric models, the unknown parameters of which are assumed to lie between known bounds. This type of uncertainty is difficult to handle with reference techniques such as H_∞-analysis and μ-analysis, and interval techniques turn out to be particularly suitable, as examples will show.

We did our best to make this chapter understandable by readers who are not specialists in control. As a result, control-oriented readers may find some issues oversimplified and may complain that important techniques are not covered. We plead guilty, but hope that these readers will nevertheless find interesting material here, which they may easily extend and adapt to other contexts.

The chapter is organized as follows. In Section 7.2, basic notions about the stability of deterministic linear models are recalled, as well as the notion of stability degree. Uncertainty in the parameters of the model is introduced in Section 7.3. Various types of parameter dependency are considered, and some results from the literature are recalled, including the celebrated Kharitonov theorem and edge theorem. Some limitations of these results are indicated. Section 7.4 explains how the tools of interval analysis can be put to work for

L. Jaulin et al., *Applied Interval Analysis*

the analysis of the stability of existing uncertain systems when the tools of Section 7.3 cannot be used. Section 7.5 is dedicated to the design of controllers for uncertain processes.

7.2 Stability of Deterministic Linear Systems

Assume, for the time being, that the system Σ to be considered involves no uncertainty. In control, it is customary to distinguish *inputs*, which are signals used to act on Σ, and *outputs*, which are signals observed on Σ as it reacts. The vector of all inputs at time t will be denoted by $\mathbf{u}(t)$, and the vector of all outputs by $\mathbf{y}(t)$. We shall consider only continuous-time systems, but most of the notions to be presented can be transposed to discrete time. When Σ is linear, time-invariant and initially at rest ($\mathbf{u} \equiv \mathbf{0}$ and $\mathbf{y} \equiv \mathbf{0}$ before time $t = 0$), the Laplace transform of the output is related to that of the input by

$$\mathbf{y}(s) = \mathbf{G}(s)\mathbf{u}(s), \tag{7.1}$$

where s is the Laplace variable and $\mathbf{G}(s)$ is the *transfer matrix* associated with Σ. An important alternative representation of a linear time-invariant system Σ is the *state-space representation*

$$\Sigma : \begin{cases} \dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \\ \mathbf{y}(t) = \mathbf{C}\mathbf{x}(t), \end{cases} \tag{7.2}$$

where $\mathbf{x}$ is the *state vector*, $\dot{\mathbf{x}}$ is its first derivative with respect to time, and $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ are the drift, control and observation matrices, respectively. The state $\mathbf{x}(t)$ may be viewed as the minimal information that it is necessary to know at time t to be able to calculate the evolution of the system in response to known future inputs. The initial state vector is denoted by $\mathbf{x}(0)$ and corresponds to the *initial conditions*. We shall assume that the dimension n of $\mathbf{x}$ is finite, which will be the case whenever Σ is described by a set of linear ordinary differential equations.

The Laplace transform of the state-space representation (7.2) for zero initial conditions is

$$\begin{cases} s\mathbf{x}(s) = \mathbf{A}\mathbf{x}(s) + \mathbf{B}\mathbf{u}(s), \\ \mathbf{y}(s) \ = \mathbf{C}\mathbf{x}(s). \end{cases} \tag{7.3}$$

Eliminating $\mathbf{x}(s)$ from (7.3) we get $\mathbf{y}(s) = \mathbf{C}(s\mathbf{I}_n - \mathbf{A})^{-1}\mathbf{B}\mathbf{u}(s)$, where $\mathbf{I}_n$ is the $n \times n$ identity matrix, so the transfer matrix between $\mathbf{u}(s)$ and $\mathbf{y}(s)$ is

$$\mathbf{G}(s) = \mathbf{C}(s\mathbf{I}_n - \mathbf{A})^{-1}\mathbf{B}. \tag{7.4}$$

The entries of $\mathbf{G}(s)$ are thus rational functions of s. This would not be the case for systems involving delays or partial differential equations, which have an infinite-dimensional state vector.

Σ is said to be *asymtotically stable* if and only if its state $\mathbf{x}(t)$ converges to $\mathbf{0}$ when t tends to infinity provided that $\mathbf{u}(t) = \mathbf{0}$ for any $t \geqslant 0$. As a result, $\mathbf{y}(t)$ also converges to $\mathbf{0}$. Asymptotic stability will be the only type of stability considered in this book, and whenever we speak of a stable system, one should understand an asymptotically stable system. Conversely, when we speak of an *unstable* system, we mean a system that is not asymptotically stable. For obvious safety reasons, stability is a vital requirement for most controlled systems, and the remainder of this section will be devoted to methods for testing Σ for stability.

7.2.1 Characteristic polynomial

When $\mathbf{u}(t) = \mathbf{0}$ for any $t \geqslant 0$, the state vector satisfies

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}(0). \tag{7.5}$$

Since the entries of the matrix $e^{\mathbf{A}t}$ are linear combinations of terms of the form $e^{\lambda_i t}$, where the λ_is are the eigenvalues of $\mathbf{A}$, Σ is stable if and only if all the eigenvalues have strictly negative real parts, *i.e.*, $\mathrm{Re}(\lambda_i) < 0$, $i = 1, \ldots, n$. Now, the eigenvalues of $\mathbf{A}$ are also the roots of the *characteristic polynomial* of $\mathbf{A}$, defined by

$$P(s) = \det(s\mathbf{I}_n - \mathbf{A}). \tag{7.6}$$

This polynomial can be written as

$$P(s) = a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0. \tag{7.7}$$

$P(s)$ also corresponds to the denominators of the entries of the transfer matrix $\mathbf{G}(s)$, as evidenced by (7.4), unless there are simplifications between numerators and denominators.

By extension, we shall say that $P(s)$ is *stable* if and only if all its roots have strictly negative real parts, and that the roots of Σ are those of its characteristic polynomial. Thus, Σ is stable if and only if $P(s)$ is stable, that is, if all its roots are in the left part $\mathbb{C}^-$ of the complex plane. A necessary condition for $P(s)$ to be stable is that all its coefficients have the same sign. If the leading coefficient a_n in (7.7) is taken equal to 1, this condition translates into $a_i > 0$, $i = 0, \ldots, n-1$.

7.2.2 Routh criterion

Efficient methods are available to check whether a given polynomial is stable. The Routh criterion is based on the construction of the *Routh table* (Table 7.1). The coefficients a_i of the polynomial are stored in the first two rows of this table as indicated. These rows are padded with zeros on their right. The remaining $(n-1)$ rows of the Routh table are computed as follows:

$$\begin{array}{llll} b_1 = \dfrac{a_{n-1}a_{n-2} - a_n a_{n-3}}{a_{n-1}}, & b_2 = \dfrac{a_{n-1}a_{n-4} - a_n a_{n-5}}{a_{n-1}}, & \cdots & \\ c_1 = \dfrac{b_1 a_{n-3} - a_{n-1}b_2}{b_1}, & c_2 = \dfrac{b_1 a_{n-5} - a_{n-1}b_3}{b_1}, & \cdots & \\ \vdots & \vdots & \ddots & \end{array} \tag{7.8}$$

Table 7.1. Routh table

a_n	a_{n-2}	a_{n-4}	$\cdots$	0
a_{n-1}	a_{n-3}	a_{n-5}	$\cdots$	0
b_1	b_2	b_3	$\cdots$	0
c_1	c_2	c_3	$\cdots$	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	0
g_1	0			
h_1				

The number of roots of $P(s)$ with positive real part is equal to the number of sign changes in the first column of the Routh table. For instance, if the first column of the Routh table contains the sequence $(1, 12, -4, 3, 2, -1)$ in this order, then the number of sign changes is equal to three and $P(s)$ has three roots with positive real parts. Assume that a_n has been normalized to one and define the *Routh vector* as the vector of all the entries of the first column of the Routh table after discarding the first entry:

$$\mathbf{r} = (a_{n-1}, b_1, \cdots, h_1)^{\mathrm{T}}. \tag{7.9}$$

$P(s)$ is stable if and only if $\mathbf{r} > \mathbf{0}$ and unstable (*i.e.*, not asymptotically stable) if and only if there exists a component r_i of $\mathbf{r}$ such that $r_i \leqslant 0$. See Levine (1996) to deal with situations where some entries of $\mathbf{r}$ are zeros.

7.2.3 Stability degree

The location of the roots of $P(s)$ provides more information that just the stability or instability of Σ. When Σ is stable, the real parts of the roots of its characteristic polynomial are related to the speed with which $\mathbf{x}$ converges to $\mathbf{0}$ in the absence of input, and complex roots are responsible for oscillations in the process, if any. For instance, if the roots of $P(s)$ are

$$(-0.2 - 3j, -0.2 + 3j, -0.5 - 7j, -0.5 + 7j, -1 - 3j, -1 + 3j), \tag{7.10}$$

then, in the absence of input, all the components $y_i(t)$ of the output vector $\mathbf{y}(t)$ of the system Σ given by (7.1) or (7.2) have the form

$$\begin{aligned} y_i(t) = {} & \alpha_1 \sin(3t + \phi_1) \exp(-0.2t) + \alpha_2 \sin(7t + \phi_2) \exp(-0.5t) \\ & + \alpha_3 \sin(3t + \phi_3) \exp(-t), \end{aligned} \tag{7.11}$$

where the coefficients $\alpha_1, \alpha_2, \alpha_3, \phi_1, \phi_2$ and ϕ_3 depend on the initial conditions $\mathbf{x}(0)$. The function $y_i(t)$ is depicted on Figure 7.1 for $\alpha_1 = \alpha_2 = \alpha_3 = 1$ and $\phi_1 = \phi_2 = \phi_3 = 0$. The location of the roots is directly related to the temporal behaviour of the outputs of the system. To impose characteristics of this behaviour, it is thus natural to require that these roots belong to a prespecified region Γ of the complex plane. This corresponds to the notion of *Γ-stability*. Now, interval methods can prove the Γ-stability of a polynomial efficiently without computing its roots. For instance, proving that the polynomial

$$P(s) = s^8 + 16s^7 + 112s^6 + 448s^5 + 1120s^4 + 1792s^3 + 1792s^2 + 1024s + 256 \tag{7.12}$$

is Γ-stable, where Γ is the circle with radius 2 and centre -3, amounts to checking that the CSP

$$\mathcal{H} : (P(s) = 0,\ |s + 3| > 2) \tag{7.13}$$

has no solution, which can be done with SIVIAX described on page 104.

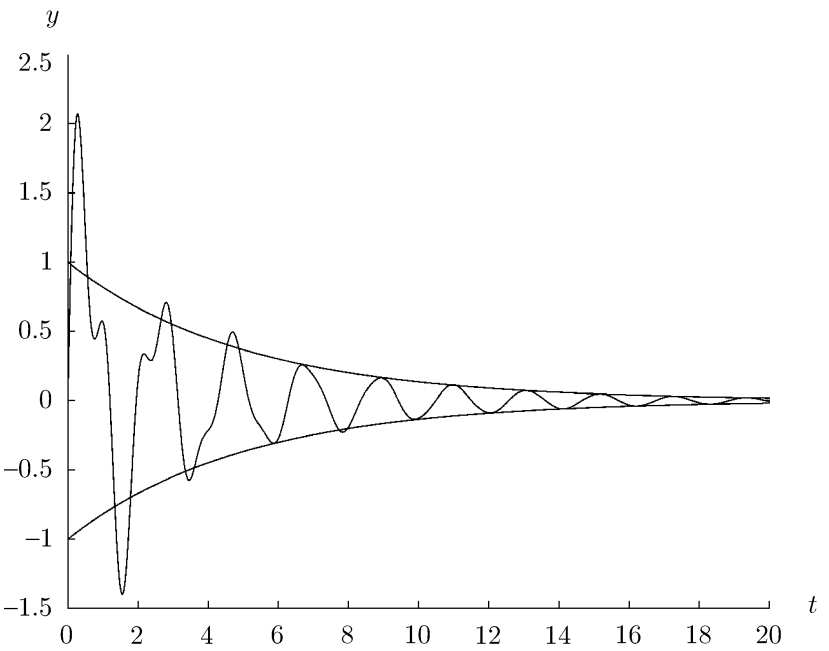


Fig. 7.1. A possible output of a system with roots as in (7.10); the asymptotic enveloping exponential curves correspond to $\pm\exp(-0.2t)$

Sometimes it is possible to translate Γ-stability into asymptotic stability by algebraic transformation (Sondergeld, 1983). This is so, for example, when

Γ is the part Γ_δ of the complex plane located on the left of the vertical line $\mathrm{Re}(s) = -\delta$. A system Σ is said to be *δ-stable* if and only if it is Γ_δ-stable, *i.e.*, all its roots are in Γ_δ. For instance, if the roots of Σ are given by (7.10), then Σ is 0.1-stable but 0.3-unstable. To check the δ-stability of $P(s)$, it suffices to check the stability of the polynomial

$$\begin{aligned} Q_\delta(s) &= P(s-\delta) \\ &= (s-\delta)^n + a_{n-1}(s-\delta)^{n-1} + \ldots + a_1(s-\delta) + a_0 \\ &= s^n + b_{n-1}(\delta)s^{n-1} + \ldots + b_1(\delta)s + b_0(\delta), \end{aligned} \tag{7.14}$$

as stated by the following theorem.

Theorem 7.1 *$P(s)$ is δ-stable if and only if $P(s-\delta)$ is stable.* ■

Proof. The proposition "$Q_\delta(s)$ is stable" is equivalent to the implication

$$\forall s \in \mathbb{C},\ Q_\delta(s) = 0 \Rightarrow \mathrm{Re}(s) < 0, \tag{7.15}$$

i.e., to

$$\forall s \in \mathbb{C},\ P(s-\delta) = 0 \Rightarrow \mathrm{Re}(s) < 0, \tag{7.16}$$

which is equivalent to the implication

$$\forall s \in \mathbb{C},\ P(s) = 0 \Rightarrow \mathrm{Re}(s+\delta) < 0. \tag{7.17}$$

This means that all the roots of $P(s)$ are in Γ_δ. ■

Now, the stability of $Q_\delta(s)$ can be tested using the Routh criterion. Define the δ-Routh vector $\mathbf{r}(\delta)$ as the Routh vector associated with $Q_\delta(s)$. Then

$$P(s) \text{ is } \delta\text{-stable} \quad \Leftrightarrow \quad Q_\delta(s) \text{ is stable} \quad \Leftrightarrow \quad \mathbf{r}(\delta) > \mathbf{0}. \tag{7.18}$$

The *stability degree* (or *decay rate*) of $P(s)$ is

$$\delta_M \triangleq \sup_{\mathbf{r}(\delta)>\mathbf{0}} \delta = \max_{\mathbf{r}(\delta)\geqslant\mathbf{0}} \delta. \tag{7.19}$$

By extension, the stability degree of Σ is that of its characteristic polynomial. The larger the stability degree of Σ is, the faster the state of Σ will return to equilibrium in the absence of inputs. Stability degree is thus an important factor to be taken into account when designing a controller. Figure 7.2 illustrates the significance of the stability degree δ_M for the root configuration of (7.10); δ_M is found here to be equal to 0.2. This means that asymptotically the dominant component of $y_i(t)$ is $\alpha_1 \sin(3t+\phi_1)\exp(-0.2t)$ (see Figure 7.1).

Often, however, the equations describing Σ are imprecise, and one would like to take the uncertainty of the model into account when testing Σ for stability, in order to reach a conclusion that is robust to this uncertainty. Methods to do so are presented in the next section.

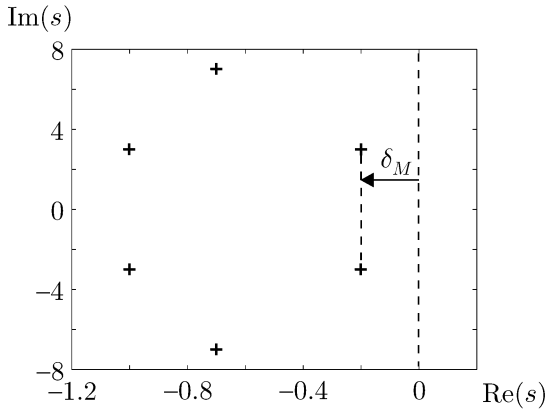


Fig. 7.2. Stability degree δ_M; the crosses correspond to the roots of the characteristic polynomial

7.3 Basic Tests for Robust Stability

Assume now that the state-space representation of the system Σ depends on some n_{p}-dimensional time-invariant parameter vector $\mathbf{p}$:

$$\Sigma(\mathbf{p}) : \begin{cases} \dot{\mathbf{x}}(t) = \mathbf{A}(\mathbf{p})\mathbf{x}(t) + \mathbf{B}(\mathbf{p})\mathbf{u}(t), \\ \mathbf{y}(t) = \mathbf{C}(\mathbf{p})\mathbf{x}(t), \end{cases} \tag{7.20}$$

where $\mathbf{p}$ is known to belong to the box $[\mathbf{p}]$. Let $\Sigma([\mathbf{p}])$ be the set of all the systems $\Sigma(\mathbf{p})$ such that $\mathbf{p}$ belongs to $[\mathbf{p}]$. $\Sigma([\mathbf{p}])$ is said to be *robustly stable* if and only if $\Sigma(\mathbf{p})$ is stable for any $\mathbf{p}$ in $[\mathbf{p}]$. Proving the robust stability of $\Sigma([\mathbf{p}])$ is one of the fundamental topics of robust control theory. The set of all the characteristic polynomials associated with $\Sigma([\mathbf{p}])$ is defined by

$$P(s, [\mathbf{p}]) \triangleq \left\{ a_n(\mathbf{p})s^n + a_{n-1}(\mathbf{p})s^{n-1} + \cdots + a_0(\mathbf{p}) \mid \mathbf{p} \in [\mathbf{p}] \right\}. \tag{7.21}$$

Define the *coefficient function* by

$$\mathbf{a}(\mathbf{p}) \triangleq (a_n(\mathbf{p}), a_{n-1}(\mathbf{p}), \cdots, a_1(\mathbf{p}), a_0(\mathbf{p}))^{\mathrm{T}}, \tag{7.22}$$

and the *coefficient set* by

$$\mathbb{A} \triangleq \{\mathbf{a}(\mathbf{p}) \mid \mathbf{p} \in [\mathbf{p}]\} = \mathbf{a}([\mathbf{p}]). \tag{7.23}$$

A polynomial $P(s, \mathbf{p})$ is entirely specified by its coefficient function $\mathbf{a}(\mathbf{p})$. This is why, for the sake of brevity, we shall also use $\mathbf{a}(\mathbf{p})$ to designate the polynomial $P(s, \mathbf{p})$ and $\mathbb{A}$ to designate the corresponding set of polynomials. Depending on the context, $\mathbf{a}$ may thus be a coefficient function or a polynomial, and $\mathbb{A}$ a set of coefficient functions or a family of polynomials.

Consider a family $\mathbb{A}$ of polynomials, and assume that the degree of each of them is equal to n (a_n is thus never equal to zero). $\mathbb{A}$ is said to be robustly

stable if and only if all polynomials in $\mathbb{A}$ are stable, and *robustly unstable* if and only if all polynomials in $\mathbb{A}$ are unstable. The problem to be considered now is the test of robust stability (or instability) for different types of $\mathbb{A}$.

- **Case 1**: The coefficient set $\mathbb{A}$ is a box

$$\mathbb{A} = [a_n] \times [a_{n-1}] \times \cdots \times [a_1] \times [a_0]\,. \tag{7.24}$$

The corresponding family of polynomials is classically called an *interval polynomial*. It can be written as

$$\mathbb{A} = [a_n]\, s^n + [a_{n-1}]\, s^{n-1} + \cdots + [a_1]\, s + [a_0]\,. \tag{7.25}$$

Note that when $\mathbb{A}$ is given under the form (7.23), *i.e.*, $\mathbb{A} = \mathbf{a}([\mathbf{p}])$, $\mathbf{a}$ may not be the identity function and may even be non-linear in $\mathbf{p}$. If, for instance, $\mathbf{a}$ is a function from $\mathbb{R}^2$ to $\mathbb{R}^2$ defined by

$$\mathbf{a}\,(\mathbf{p}) = (\sin\,(p_1)\,, \exp\,(p_2))^{\mathrm{T}}\,, \tag{7.26}$$

then the image by $\mathbf{a}$ of any box $[\mathbf{p}]$ is a box and $\mathbb{A}$ is thus a box or equivalently an interval polynomial.
- **Case 2**: $\mathbb{A}$ is a polytope, or equivalently designates a *polytope polynomial*.
- **Case 3**: $\mathbb{A}$ is the image of a box $[\mathbf{p}]$ by a function $\mathbf{a}\,(.)$. $\mathbb{A}$ then designates an *image-set polynomial*, which can be written under the form (7.23).

Example 7.1 *Since the entries of* $\mathbf{p}$ *appear independently in the coefficients, the family of polynomials*

$$\mathbb{A} = \left\{(p_1 + \sin^2(p_2))s^2 + \exp(\sqrt{p_3})s + p_5 \ln(1 + p_6) \mid \mathbf{p} \in [\mathbf{p}]\right\} \tag{7.27}$$

is an interval polynomial and thus a polytope polynomial and an image-set polynomial. It can be rewritten indifferently as

$$\mathbb{A} = ([p_1] + \sin^2([p_2]))s^2 + \exp(\sqrt{[p_3]})s + [p_5] \ln(1 + [p_6]), \tag{7.28}$$

or as

$$\mathbb{A} = ([p_1] + \sin^2([p_2])) \times \exp(\sqrt{[p_3]}) \times [p_5] \ln(1 + [p_6]). \tag{7.29}$$

The family of polynomials

$$\mathbb{A} = \left\{(\cos(p_1) + \sin^2(p_2))s^2 + 3\cos(p_1)s + p_3 + \sin^2(p_2) \mid \mathbf{p} \in [\mathbf{p}]\right\} \tag{7.30}$$

is not an interval polynomial, but is a polytope polynomial. For instance, if $[\mathbf{p}] = [0, \pi/2]^{\times 3}$*, then* $\cos([p_1]) = \sin^2([p_2]) = [0, 1]$*.* $\mathbb{A}$ *thus can be rewritten as*

$$\mathbb{A} = \left\{(p'_1 + p'_2)s^2 + 3p'_1 s + p'_3 + p'_2 \mid \mathbf{p}' \in [0, 1] \times [0, 1] \times [0, \pi/2]\right\}. \tag{7.31}$$

Because of the dependency of the coefficients of the polynomial, no form similar to (7.28) can be given. ■

7.3.1 Interval polynomials

The Kharitonov theorem (Kharitonov, 1978) provides necessary and sufficient conditions for the robust stability of interval polynomials; see also Bialas (1983), Barmish (1984) and countless publications in the 1980s. This very important result is at the origin of the extreme-point approach to testing uncertain systems for robust stability (Bartlett et al., 1988).

Theorem 7.2 *(Kharitonov) The interval polynomial*

$$[\mathbf{a}] = [\underline{a}_n, \overline{a}_n] \times \cdots \times [\underline{a}_1, \overline{a}_1] \times [\underline{a}_0, \overline{a}_0]$$

is robustly stable if and only if the four polynomials $K_1(s)$, $K_2(s)$, $K_3(s)$ and $K_4(s)$ respectively given by

$$\begin{aligned}
&\underline{a}_n s^n + \underline{a}_{n-1} s^{n-1} + \overline{a}_{n-2} s^{n-2} + \overline{a}_{n-3} s^{n-3} + \underline{a}_{n-4} s^{n-4} + \underline{a}_{n-5} s^{n-5} \ldots, \\
&\overline{a}_n s^n + \underline{a}_{n-1} s^{n-1} + \underline{a}_{n-2} s^{n-2} + \overline{a}_{n-3} s^{n-3} + \overline{a}_{n-4} s^{n-4} + \underline{a}_{n-5} s^{n-5} \ldots, \\
&\overline{a}_n s^n + \overline{a}_{n-1} s^{n-1} + \underline{a}_{n-2} s^{n-2} + \underline{a}_{n-3} s^{n-3} + \overline{a}_{n-4} s^{n-4} + \overline{a}_{n-5} s^{n-5} \ldots, \\
&\underline{a}_n s^n + \overline{a}_{n-1} s^{n-1} + \overline{a}_{n-2} s^{n-2} + \underline{a}_{n-3} s^{n-3} + \underline{a}_{n-4} s^{n-4} + \overline{a}_{n-5} s^{n-5} \ldots
\end{aligned}$$

are stable. ■

Studying the robust stability of a non-denumerable set of polynomials thus boils down to testing at most four of them for stability, independently of the value of n. (When $n \leqslant 5$, it is possible to prove robust stability with even less computation, see Anderson et al. (1987).) If the family $\mathbb{A}$ is not an interval polynomial, the Kharitonov theorem can still be used by wrapping $\mathbb{A}$ into an interval polynomial $[\mathbf{a}]$. If $[\mathbf{a}]$ is robustly stable, then $\mathbb{A}$ is also robustly stable, but the condition becomes only sufficient, of course. This is illustrated by the following example.

Example 7.2 *The family of polynomials*

$$\begin{aligned}
\mathbb{A} = \{&p_5 s^4 + (p_4 + \cos^2(p_3)) s^3 + 2p_1 s^2 + p_2 \sqrt{p_4} s + p_1 \mid \\
&p_1 \in [5,7],\ p_2 \in [3,4],\ p_3 \in [-\pi/4, \pi/4],\ p_4 \in [1,2],\ p_5 \in [1,2]\}
\end{aligned}$$

is not an interval polynomial, but it is a subset of the interval polynomial

$$[\mathbf{a}] = [1,2]s^4 + [3/2,3]s^3 + [10,14]s^2 + [3,4\sqrt{2}]s + [5,7]. \tag{7.32}$$

The Kharitonov polynomials associated with $[\mathbf{a}]$ *are*

$$\begin{aligned}
K_1(s) &= s^4 + \tfrac{3}{2}s^3 + 14s^2 + 4\sqrt{2}s + 5, \\
K_2(s) &= 2s^4 + \tfrac{3}{2}s^3 + 10s^2 + 4\sqrt{2}s + 7, \\
K_3(s) &= 2s^4 + 3s^3 + 10s^2 + 3s + 7, \\
K_4(s) &= s^4 + 3s^3 + 14s^2 + 3s + 5.
\end{aligned} \tag{7.33}$$

Each of them is easily proved stable, e.g., by the Routh criterion. Therefore $[\mathbf{a}]$ *is robustly stable, and so is* $\mathbb{A}$. *Assume now (Wei and Yedavalli, 1989) that*

$$p_1 \in [1.5, 4],\ p_2 = 1,\ p_3 = \pi/2,\ p_4 = 1,\ p_5 = 1. \tag{7.34}$$

Then

$$[\mathbf{a}] = s^4 + s^3 + [3, 8]s^2 + s + [1.5, 4]. \tag{7.35}$$

The corresponding Kharitonov polynomial $K_2(s) = s^4 + s^3 + 3s^2 + s + 4$ *is unstable, which implies that* $[\mathbf{a}]$ *is not robustly stable. This does not imply that* $\mathbb{A}$ *is not robustly stable. It is actually easy to show that* $\mathbb{A}$ *is robustly stable, for instance by building a formal Routh table depending on* p_1. ■

In practice, $\mathbb{A}$ is seldom an interval polynomial. For instance, if the entries of $\mathbf{p}$ appear independently in the drift matrix $\mathbf{A}(\mathbf{p})$, then the coefficient function $\mathbf{a}(\mathbf{p})$ is multilinear and the coefficient set $\mathbb{A}$ is not even a polytope.

7.3.2 Polytope polynomials

The main result regarding the robust stability of polytope polynomials is the *edge theorem* (Bartlett et al., 1988).

Theorem 7.3 *(edge theorem) The polytope polynomial* $\mathbb{A}$ *is robustly stable if all its edges are robustly stable.* ■

Recall that the number of edges in a box increases exponentially with its dimension. A 10-dimensional box has 5120 edges. Therefore, the edge theorem may require the study of the robust stability of many edge polynomials of the form

$$P_{1,2}(s, \lambda) = \lambda P_1(s) + (1 - \lambda)P_2(s), \quad \text{with } \lambda \in [0, 1]. \tag{7.36}$$

Now, this is not an easy problem. There are examples where the two vertices $P_1(s)$ and $P_2(s)$ of the edge are stable whereas the edge $P_{1,2}(s, \lambda)$ itself is not robustly stable (Bialas and Garloff, 1985). The robust stability of an edge can be established using the Bialas algebraic condition (Bialas, 1985), based on the Hurwitz criterion. Another approach is to treat polytope polynomials as special cases of image-set polynomials, as presented in the following section.

7.3.3 Image-set polynomials

Assume that the characteristic polynomial associated with $\Sigma(\mathbf{p})$ is

$$P(s, \mathbf{p}) = s^n + a_{n-1}(\mathbf{p})s^{n-1} + \ldots + a_1(\mathbf{p})s + a_0(\mathbf{p}). \tag{7.37}$$

The Routh vector of $P(s, \mathbf{p})$ is a function of $\mathbf{p}$, called the *Routh function* (Didrit, 1997) and denoted by $\mathbf{r}(\mathbf{p})$. From the Routh criterion, the following equivalence holds:

$$P(s,\mathbf{p}) \text{ is stable } \Leftrightarrow \mathbf{r}(\mathbf{p}) > \mathbf{0}. \tag{7.38}$$

Thus, $\Sigma([\mathbf{p}])$ is robustly stable if and only if $\mathbf{r}(\mathbf{p}) > \mathbf{0}$ for any $\mathbf{p}$ in $[\mathbf{p}]$. Let $[\mathbf{r}]$ be an inclusion function for the Routh function $\mathbf{r}$. If all the components $[r_i]([\mathbf{p}])$ of $[\mathbf{r}]([\mathbf{p}])$ are positive (*i.e.*, if their lower bounds are strictly positive) then $\Sigma([\mathbf{p}])$ is robustly stable. On the other hand, if there exists a component r_i of $\mathbf{r}$ such that $r_i([\mathbf{p}]) \leqslant 0$ then $\Sigma([\mathbf{p}])$ is robustly unstable ($\Sigma(\mathbf{p})$ is not asymptotically stable for any $\mathbf{p}$ in $[\mathbf{p}]$).

An efficient way to prove the robust stability or instability of $\Sigma([\mathbf{p}])$ is to use contractors (see Chapter 4). Proving robust instability amounts to proving that $\mathbf{r}(\mathbf{p}) > \mathbf{0}$ admits no solution in $[\mathbf{p}]$. This is equivalent to saying that the solution set of the CSP

$$\mathcal{H} : \left(\mathbf{r}(\mathbf{p}) - \mathbf{y} = \mathbf{0}, \mathbf{p} \in [\mathbf{p}], \mathbf{y} \in [0,\infty[^{\times n}\right) \tag{7.39}$$

is empty. Note that the edge approach is unable to prove robust instability, even for interval polynomials. Proving robust stability amounts to proving that,

$$\begin{aligned}
&\forall \mathbf{p} \in [\mathbf{p}],\ \mathbf{r}(\mathbf{p}) > 0\\
\Leftrightarrow\ &\forall \mathbf{p} \in [\mathbf{p}], \forall i \in \{1,\ldots,n\},\ r_i(\mathbf{p}) > 0\\
\Leftrightarrow\ &\forall i \in \{1,\ldots,n\}, \forall \mathbf{p} \in [\mathbf{p}],\ r_i(\mathbf{p}) > 0\\
\Leftrightarrow\ &\forall i \in \{1,\ldots,n\}, \forall \mathbf{p} \in [\mathbf{p}],\ \neg\left(r_i(\mathbf{p}) \leqslant 0\right)\\
\Leftrightarrow\ &\forall i \in \{1,\ldots,n\}, \neg\left(\exists \mathbf{p} \in [\mathbf{p}] \mid r_i(\mathbf{p}) \leqslant 0\right),
\end{aligned} \tag{7.40}$$

where $\neg A$ is equivalent to (A is false). Now, saying that $\neg(\exists \mathbf{p} \in [\mathbf{p}] \,|\, r_i(\mathbf{p}) \leqslant 0)$ is equivalent to saying that the CSP

$$\mathcal{H}_i : (r_i(\mathbf{p}) + y = 0, \mathbf{p} \in [\mathbf{p}], y \in]0,\infty[) \tag{7.41}$$

has an empty solution set. Therefore, $\Sigma([\mathbf{p}])$ is robustly stable if and only if all such $\mathcal{H}_i$'s have empty solution sets. All the contractors presented in Chapter 4 can thus be used to check both robust stability and robust instability. Note, however, that checking robust stability with this approach requires the contraction of n CSPs instead of one for robust instability.

Remark 7.1 *The type of reasoning followed here to prove robust stability can also be used to check that a given box is inside a set* $\mathbb{S}$ *defined by nonlinear inequalities. Contractors could thus be used in an algorithm such as* SIVIA*, not only to check that a box is outside* $\mathbb{S}$*, but also to check that a box is inside* $\mathbb{S}$*.* ■

Remark 7.2 *When the coefficient function* $\mathbf{a}(\mathbf{p})$ *is polynomial in* $\mathbf{p}$*, so is* $\mathbf{r}(\mathbf{p})$*. Bernstein polynomials may then be used for computing an outer approximation of* $\mathbf{r}([\mathbf{p}])$ *and thus for checking the signs of the entries of* $\mathbf{r}([\mathbf{p}])$ *(Vicino et al., 1990; Milanese et al., 1991; Garloff, 2000).* ■

7.3.4 Conclusion

The complexity of the test of the stability of $\Sigma([\mathbf{p}])$ drastically depends on the nature of the coefficient set $\mathbb{A}$. When $\mathbb{A}$ is a box, the Kharitonov theorem reduces the task to that of checking at most four polynomials for stability, whatever the degree of the uncertain polynomial. When $\mathbb{A}$ is a polytope, the edge theorem reduces the task to that of checking the edges of this polytope, which may nevertheless reveal quite demanding. When $\mathbb{A}$ is a general image-set polynomial, the problem becomes even more complicated (Nemirovskii, 1993; Poljak and Rohn, 1993; Blondel and Tsitsiklis, 1995). The next section presents numerical methods based on interval analysis that can be used even in this most complicated case. Moreover, these methods will make it possible to characterize the part of $[\mathbf{p}]$ that corresponds to stable systems, when $\Sigma([\mathbf{p}])$ is neither robustly stable nor robustly unstable, as well as to characterize level sets of the stability degree.

7.4 Robust Stability Analysis

The basic tests presented in Section 7.3 will now be used for the analysis of the robust stability of $\Sigma([\mathbf{p}])$.

7.4.1 Stability domains

The stability domain $\mathbb{S}_{\mathrm{p}}$ of the polynomial

$$P(s,\mathbf{p}) = s^n + a_{n-1}(\mathbf{p})s^{n-1} + \ldots + a_1(\mathbf{p})s + a_0(\mathbf{p}) \tag{7.42}$$

is the set of all the parameter vectors $\mathbf{p}$ such that $P(s,\mathbf{p})$ is stable. It can be defined as

$$\mathbb{S}_{\mathrm{p}} \triangleq \{\mathbf{p} \in \mathbb{R}^{n_{\mathrm{p}}} \mid \mathbf{r}(\mathbf{p}) > \mathbf{0}\} = \mathbf{r}^{-1}\left(]0,+\infty[^{\times n}\right). \tag{7.43}$$

The characterization of $\mathbb{S}_{\mathrm{p}}$ can thus be cast into the framework of set inversion and performed by the algorithm SIVIA (Walter and Jaulin, 1994).

Example 7.3 *Consider the multi-affine polynomial (Ackermann et al., 1990; Ackermann, 1992)*

$$\begin{aligned} P(s,\mathbf{p}) = {} & s^3 + (p_1 + p_2 + 2)s^2 + (p_1 + p_2 + 2)s \\ & + 2p_1p_2 + 6p_1 + 6p_2 + 2 + \sigma^2, \end{aligned} \tag{7.44}$$

where the coefficient σ corresponds to the radius of an unstable disk centred at $\mathbf{p}_c = (1,1)^T$ inside a stable region. The construction of the Routh table leads to the Routh function

$$\mathbf{r}(\mathbf{p}) = \begin{pmatrix} p_1 + p_2 + 2 \\ (p_1 - 1)^2 + (p_2 - 1)^2 - \sigma^2 \\ 2(p_1 + 3)(p_2 + 3) - 16 + \sigma^2 \end{pmatrix}. \tag{7.45}$$

Note that if $\sigma = 0$ *and* p_1 *and* p_2 *are positive, then* $P(s, \mathbf{p})$ *is always stable, except at* $\mathbf{p}_c$. *For* $[\mathbf{p}] = [-3, 7] \times [-3, 7]$, $\varepsilon = 0.05$ *and* $\sigma = 0.5$, SIVIA *computed the paving of Figure 7.3 in* 0.3 *s on a* PENTIUM 90 *(Didrit, 1997). Since all the components of*

$$[\mathbf{r}]([2, 7], [2, 7]) = \begin{pmatrix} [6, 16] \\ [\frac{3}{4}, \frac{287}{4}] \\ [\frac{135}{4}, \frac{735}{4}] \end{pmatrix} \tag{7.46}$$

are positive, the box $[2, 7] \times [2, 7]$ *was proved to be stable in a single iteration (see Figure 7.3).*

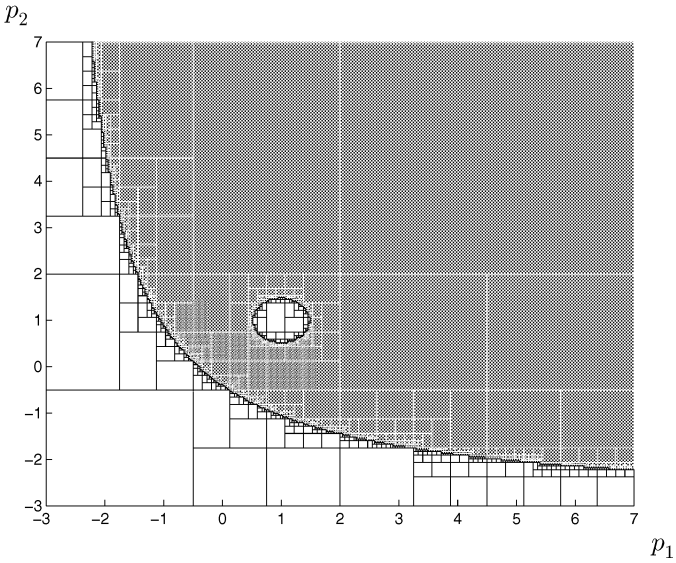


Fig. 7.3. Characterization of the stability domain of Example 7.3 for $\sigma = 0.5$; grey boxes are stable and white boxes unstable

On the other hand, since

$$[\mathbf{r}]([-3, -\frac{1}{2}], [-3, -\frac{1}{2}]) = \begin{pmatrix} [-4, 1] \\ [\frac{17}{4}, \frac{127}{4}] \\ [-\frac{63}{4}, -\frac{13}{4}] \end{pmatrix}, \tag{7.47}$$

the box $[-3, -\frac{1}{2}] \times [-3, -\frac{1}{2}]$ *was proved unstable in a single iteration. Figure 7.4 has been generated in* 0.2 *s for* $\sigma = 0$. SIVIA *was unable to prove that* $\mathbf{p} = (1, 1)^T$ *was unstable but a small indeterminate box (too small to be visible) was generated around this unstable point.* ■

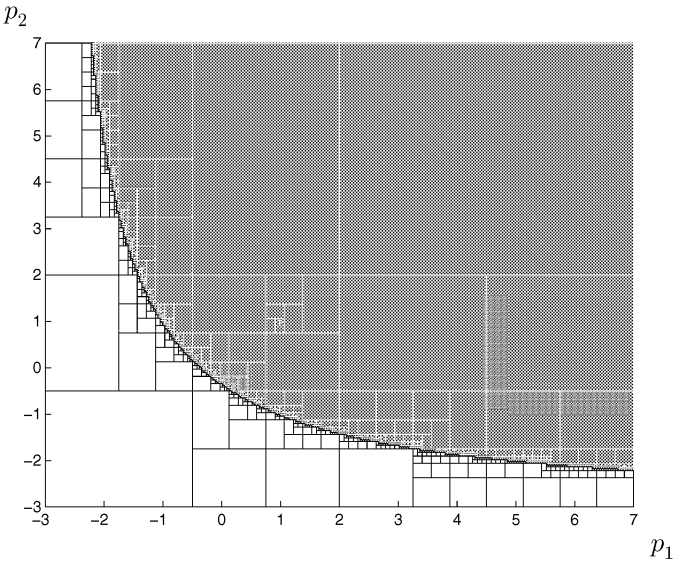


Fig. 7.4. Characterization of the stability domain of Example 7.3 for $\sigma = 0$; grey boxes are stable and white boxes unstable; the unstable point (1, 1) is surrounded by an indeterminate box, too small to be visible

Example 7.3 has been used by Ackermann to illustrate the limits of the idea of studying edges: for $\sigma = 0.5$, and $[\mathbf{p}] = [0, 2] \times [0, 2]$ all the edges of $[\mathbf{p}]$ are stable, but $[\mathbf{p}]$ nevertheless contains an unstable region. This example was treated in Murdock et al. (1991) with an approach based on a genetic algorithm. Since this approach amounts to random search, its efficiency decreases with σ and no guarantee on its result can be provided. Kiendl and Michalske (1992) have studied the same example with a partition method. Their approach is valid only in the case of polytope polynomials, which implies here a pessimistic reparametrization of the model, with detrimental consequences on the quality of the results (see the figures in Kiendl and Michalske, 1992).

The following example shows how SIVIA can be used to characterize the root locus of an uncertain polynomial.

Example 7.4 *We already know that the uncertain polynomial* $P(s, [\mathbf{p}])$ *of Example 7.3 is robustly stable for* $\sigma = 0$ *and* $\mathbf{p} \in [\mathbf{p}] = [2, 7] \times [2, 7]$. *Define its* root locus $\mathcal{R}([\mathbf{p}])$ *as the set of all the roots of* $P(s, \mathbf{p})$ *for* $\mathbf{p} \in [\mathbf{p}]$, i.e.,

$$\mathcal{R}([\mathbf{p}]) \triangleq \{s \in \mathbb{C} \mid P(s, \mathbf{p}) = 0, \mathbf{p} \in [\mathbf{p}]\}\,. \tag{7.48}$$

$\mathcal{R}([\mathbf{p}])$ *is thus the projection onto the complex plane of the set*

$$\{(s, \mathbf{p}) \in \mathbb{C}\times[\mathbf{p}] \mid P(s, \mathbf{p}) = 0\}\,, \tag{7.49}$$

which can be characterized by SIVIA *of Section 3.4.1, page 55. The associated subpaving, depicted on Figure 7.5, intersects the imaginary axis, so stability is not proved. Nevertheless, this subpaving provides important information about the uncertain model. For instance, it shows that, even if the model is robustly stable on* $[\mathbf{p}]$*, a small non-parametric perturbation may push some roots across the imaginary axis.* ■

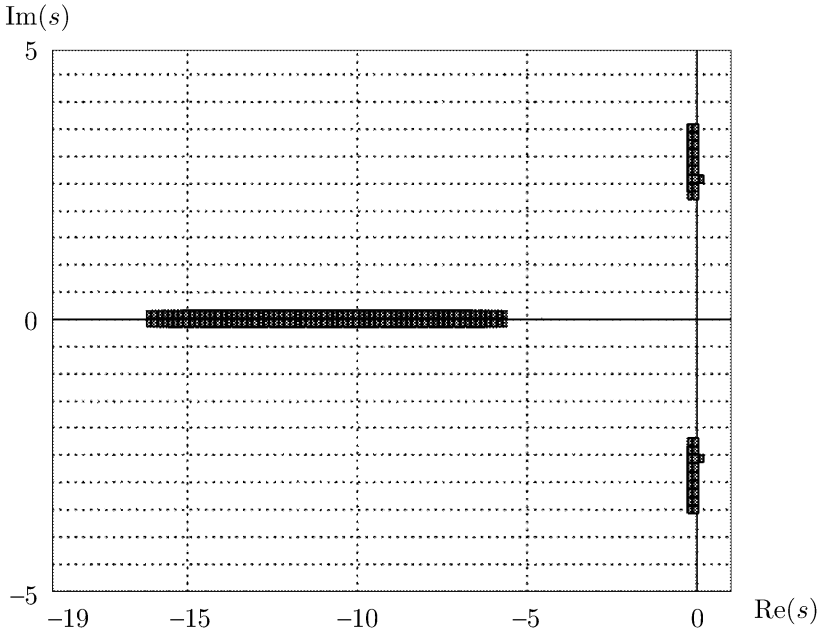


Fig. 7.5. Outer approximation of the root locus for the uncertain polynomial $P(s, [\mathbf{p}])$ of Example 7.4

7.4.2 Stability degree

Level sets. When Σ depends on a vector $\mathbf{p}$ of parameters, so does its δ-Routh vector, now written $\mathbf{r}(\mathbf{p}, \delta)$ (see (7.18), page 192). The stability degree of Σ is now defined as

$$\delta_M(\mathbf{p}) \triangleq \sup_{\mathbf{r}(\mathbf{p},\delta)>\mathbf{0}} \delta = \max_{\mathbf{r}(\mathbf{p},\delta)\geqslant\mathbf{0}} \delta. \tag{7.50}$$

ISOCRIT presented in Chapter 5, page 135, for the characterization of level sets will now be applied to the function $\delta_M(\mathbf{p})$. Recall that the level set of $\delta_M(\mathbf{p})$ associated with δ_1 is the set of all $\mathbf{p}$s in the region of interest of parameter space that are such that $\delta_M(\mathbf{p}) = \delta_1$. The levels of interest are denoted by $\delta_1, \delta_2, \ldots, \delta_m$. It is assumed that the functions $\mathbf{a}(\mathbf{p})$ and thus $\delta_M(\mathbf{p})$ are continuous.

ISOCRIT requires an inclusion function for $\delta_M(\mathbf{p})$, which can be evaluated for any given box $[\mathbf{p}]$ by minimizing and maximizing $\delta_M(\mathbf{p})$ over $[\mathbf{p}]$ with OPTIMIZE presented in Chapter 5, page 119.

Example 7.5 *For the multi-affine polynomial of Example 7.3 for* $[\mathbf{p}] = [-3,7] \times [-3,7]$, $\delta_1 = 0.1$, $\delta_2 = 0$, $\varepsilon = 0.05$ *and* $\sigma = 0.5$, ISOCRIT *computed the paving of Figure 7.6 in* 3.6 *s on a* PENTIUM 90 *(Didrit, 1997). The level set associated with* $\delta_2 = 0$ *is consistent with the result of Example 7.3 (see Figure 7.3).* ■

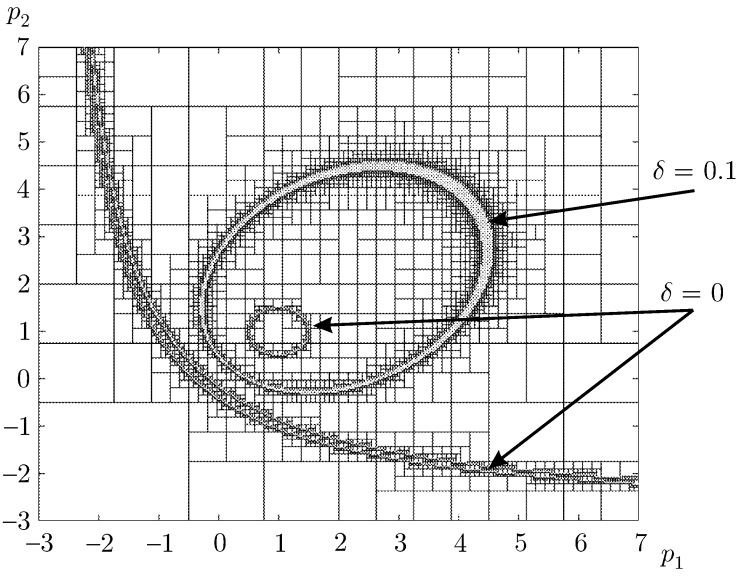


Fig. 7.6. Level sets of the stability degree for $\sigma = 0.5$

Example 7.6 *Consider now the uncertain system (Kokame and Mori, 1992)*

$$\dot{\mathbf{x}} = \begin{pmatrix} 0 & 1 & -p_1 \\ 1 & 0 & -p_2 \\ p_1 & p_2 & 1 \end{pmatrix} \mathbf{x}, \; p_1 \in [-7, 1.3], \; p_2 \in [-1, 2.5]. \tag{7.51}$$

Its characteristic polynomial is

$$P(s, \mathbf{p}) = s^3 + s^2 + (p_1^2 + p_2^2 + 1)s + 1. \tag{7.52}$$

From the Routh table of $P(s - \delta, \mathbf{p})$*, one can easily prove that for* $\delta \geqslant \frac{1}{3}$ *the system is* δ*-unstable for any* $\mathbf{p}$ *in* $[\mathbf{p}]$*, and that for* $0 \leqslant \delta < \frac{1}{3}$ *the system is* δ*-stable if and only if*

$$\begin{cases} p_1^2 + p_2^2 > \dfrac{4\delta(2\delta^2 - 2\delta + 1)}{-2\delta + 1} \triangleq \underline{\sigma}^2(\delta), \\ \\ p_1^2 + p_2^2 < \dfrac{-\delta^3 + \delta^2 - \delta + 1}{\delta} \triangleq \overline{\sigma}^2(\delta). \end{cases} \tag{7.53}$$

Therefore, $\Sigma(\mathbf{p})$ *is* δ*-stable for all* $\mathbf{p}$*s located between the circles centred at zero and with radii equal to* $\underline{\sigma}(\delta)$ *and* $\overline{\sigma}(\delta)$*. Moreover, the only unstable point is* $\mathbf{0}$*. For* $\delta_1 = 0.2$*,* $\delta_2 = 0.1$*,* $\delta_3 = 0.05$ *and* $\delta_4 = 0$*, and for* $[\mathbf{p}] = [-7, 1.3] \times [-1, 2.5]$ *and* $\varepsilon = 0.1$*,* ISOCRIT *computed the paving of Figure 7.7 in* 0.3 *s on a* PENTIUM 90 *(Didrit, 1997).* ■

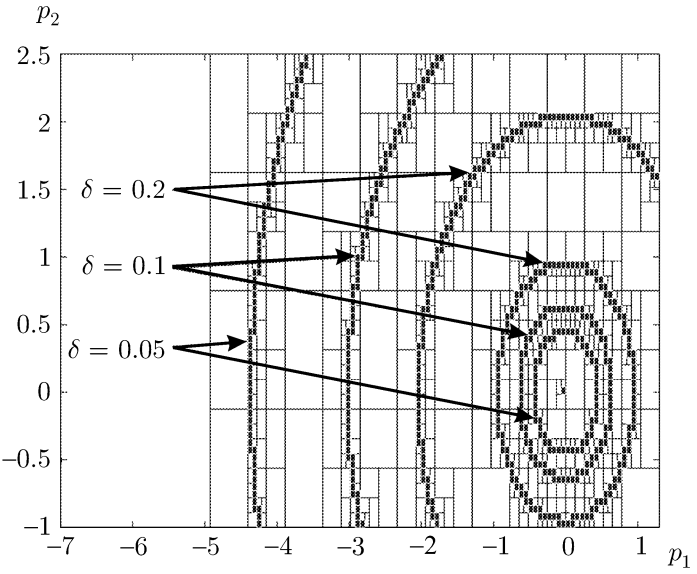


Fig. 7.7. Characterization of level sets of the stability degree

This example has also been studied by Kokame and Mori (1992). Their method applies when the drift matrix is affine in the parameters and only makes it possible to find a point in parameter space that is δ-stable for a given value of δ.

Robust stability degree. Define the *robust stability degree* $\delta_M([\mathbf{p}])$ of $\Sigma(\mathbf{p})$ for $\mathbf{p}$ in $[\mathbf{p}]$ as the stability degree in the worst case:

$$\delta_M([\mathbf{p}]) = \min_{\mathbf{p}\in[\mathbf{p}]} \max_{\mathbf{r}(\mathbf{p},\delta)\geqslant\mathbf{0}} \delta. \tag{7.54}$$

If $\delta_M([\mathbf{p}]) > 0$, then all the roots of $\Sigma(\mathbf{p})$ are in $\mathbb{C}^-$ and the uncertain system $\Sigma([\mathbf{p}])$ is robustly stable (*i.e.*, it is stable for any $\mathbf{p}$ in $[\mathbf{p}]$), as illustrated by Figure 7.8 (left). If $\delta_M([\mathbf{p}]) \leqslant 0$, then there exists some $\mathbf{p}$ in $[\mathbf{p}]$ such that $\Sigma(\mathbf{p})$ is unstable, as illustrated by Figure 7.8 (right).

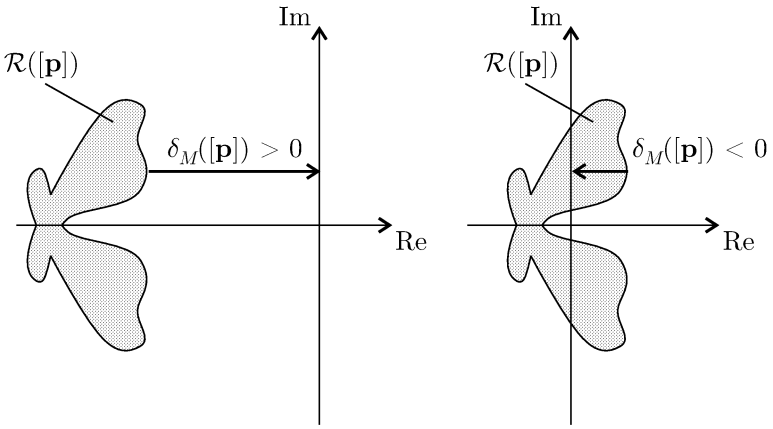


Fig. 7.8. Left: the root locus of $\Sigma([\mathbf{p}])$ is in $\mathbb{C}^-$, $\delta_M([\mathbf{p}])$ is thus positive and $\Sigma([\mathbf{p}])$ is robustly stable; right: $\Sigma([\mathbf{p}])$ is not in $\mathbb{C}^-$, $\delta_M([\mathbf{p}])$ is thus negative and $\Sigma(\mathbf{p})$ is unstable for some $\mathbf{p}$ in $[\mathbf{p}]$

The algorithm MINIMAX, presented in Section 5.6, page 133, can be used to compute $\delta_M([\mathbf{p}])$ in a guaranteed way, as illustrated by the two following examples.

Example 7.7 *Consider the uncertain system (Balakrishnan et al., 1991a)*

$$\dot{\mathbf{x}} = \begin{bmatrix} \dfrac{p_2}{1+p_2} & 2 \\ \dfrac{p_2}{1+p_1} & \dfrac{p_1}{1+p_2^2} \end{bmatrix} \mathbf{x}. \tag{7.55}$$

For $[\mathbf{p}] = [1,2] \times [0,0.5]$, *with* $\varepsilon = 0.001$, *on a* PENTIUM 90, MINIMAX *finds in* 2 *s and* 237 *iterations that the robust stability degree satisfies*

$$-2.01590 \leqslant \delta_M([\mathbf{p}]) \leqslant -2.01451.$$

The system is thus not robustly stable. ■

Example 7.8 *Consider the uncertain system (Balakrishnan et al., 1991a, 1991b)*

$$\dot{\mathbf{x}} = \begin{bmatrix} \dfrac{1}{(p_1+3.5)^2+(p_2+1)^2+\frac{1}{0.9}} & 0 \\ 0 & \dfrac{1}{p_1^4+p_2^4+1} \end{bmatrix} \mathbf{x}. \tag{7.56}$$

For $[\mathbf{p}] = [-4,0]\times[-4,4]$, *with* $\varepsilon = 0.01$, *again on a* PENTIUM 90, MINIMAX *needs only* 20 *iterations to prove that the robust stability degree satisfies*

$$-1.0048866272 \leqslant \delta_M([\mathbf{p}]) \leqslant -0.9999999781. \tag{7.57}$$

This system is thus not robustly stable. ■

7.4.3 Value-set approach

The concept of *value set* (Saeki, 1986; Barmish, 1988), allows a simple geometrical interpretation of robust stability in the complex plane. Recall that the uncertain system Σ with characteristic polynomial

$$P(s,\mathbf{p}) = a_n(\mathbf{p})s^n + a_{n-1}(\mathbf{p})s^{n-1} + \ldots + a_1(\mathbf{p})s + a_0(\mathbf{p}) \tag{7.58}$$

is robustly stable in $[\mathbf{p}]$ if the CSP

$$\mathcal{H} : (P(s,\mathbf{p}) = 0, \mathbf{p} \in [\mathbf{p}], \mathrm{Re}(s) \geqslant 0) \tag{7.59}$$

has no solution for s and $\mathbf{p}$. This can be checked easily with SIVIAX, presented in Section 5.2, page 104. Since s is complex, the dimension of the search space is dim $\mathbf{p}+2$. Let us first show that it is often possible to reduce this dimension to dim $\mathbf{p}+1$ by taking advantage of the continuity of the roots of the characteristic polynomial with respect to its coefficients. Assume that $[\mathbf{p}]$ contains a stable vector $\mathbf{p}_0$ and an unstable vector $\mathbf{p}_1$. The roots associated with $\mathbf{p}_0$ all have negative real parts and at least one of the roots associated with $\mathbf{p}_1$ has a positive real part. This is illustrated by Figure 7.9. Assume also that the coefficients of $P(s,\mathbf{p})$ are continuous in $\mathbf{p}$ and that the leading coefficient $a_n(\mathbf{p})$ never vanishes. When $\mathbf{p}$ moves from $\mathbf{p}_0$ to $\mathbf{p}_1$ in $[\mathbf{p}]$, at least one of the roots crosses the imaginary axis (see Figure 7.9), *i.e.*, there exist $\mathbf{p}$ in $[\mathbf{p}]$ and ω in $\mathbb{R}$ such that $P(j\omega,\mathbf{p}) = 0$. This leads to the following theorem.

Theorem 7.4 *If*

1. *the coefficients of* $P(s,\mathbf{p})$ *are continuous functions of* $\mathbf{p}$,
2. *the leading coefficient* $a_n(\mathbf{p})$ *never vanishes over* $[\mathbf{p}]$,
3. *there exists* $\mathbf{p}_0$ *in* $[\mathbf{p}]$ *such that* $P(s,\mathbf{p}_0)$ *is stable,*

then $P(s,[\mathbf{p}])$ *is robustly stable if and only if the CSP*

$$\mathcal{H} : (P(j\omega,\mathbf{p}) = 0, \mathbf{p} \in [\mathbf{p}],\ \omega \in \mathbb{R}) \tag{7.60}$$

has no solution. ■

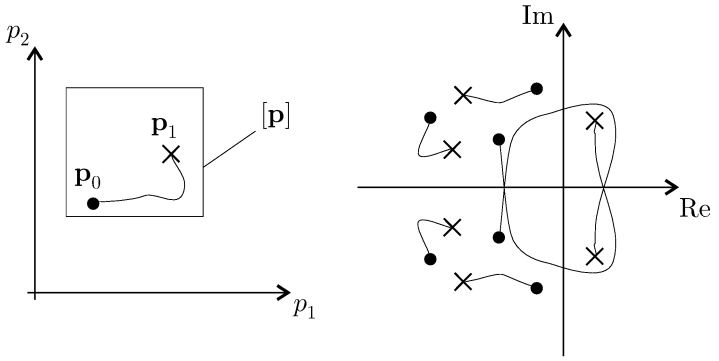


Fig. 7.9. Along the path from a stable point $\mathbf{p}_0$ to an unstable point $\mathbf{p}_1$, at least one of the roots crosses the imaginary axis

The stability of $P(s, \mathbf{p}_0)$ can be checked with the Routh criterion, for instance. The domain for ω can be restricted to $\omega \geqslant 0$, because if $(\mathbf{p}, \omega)$ is a solution of (7.60), so is $(\mathbf{p}, -\omega)$. The dimension of search space is now dim $\mathbf{p} + 1$ instead of dim $\mathbf{p} + 2$. To apply SIVIAX to prove that (7.60) has no solution, it is important to bound the domain for ω. As the module of $P(j\omega, \mathbf{p})$ tends to infinity with ω, there exists an angular frequency ω_c (*cutoff frequency*) beyond which $P(j\omega, \mathbf{p})$ will never be equal to zero for any $\mathbf{p}$ in $[\mathbf{p}]$. The following theorem (Marden, 1966) provides a mean for computing an upper bound for ω_c.

Theorem 7.5 *All the roots of* $P(s) = a_n s^n + \cdots + a_1 s + a_0$, *with* $a_n \neq 0$, *are inside the disk with centre zero and radius*

$$\beta = 1 + \frac{\max\{|a_0|, |a_1|, \ldots, |a_{n-1}|\}}{|a_n|}. \tag{7.61}$$

■

Proof. First, let us prove the result for the monic polynomial $P_1(s) = s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0$. Since

$$P_1(s) = s(s(s(s(\ldots) + a_4) + a_3) + a_2) + a_1) + a_0, \tag{7.62}$$

$P_1(s)$ can be obtained by the following sequence:

$$\begin{aligned} &Q_0(s) = 1, \\ &Q_i(s) = sQ_{i-1}(s) + a_{n-i},\ i = 1, \ldots, n, \\ &P_1(s) = Q_n(s). \end{aligned} \tag{7.63}$$

Assume that $|s| \geqslant \beta$ and that $|Q_{i-1}(s)| \geqslant 1$ (this holds true for $i = 1$). $|Q_i(s)| = |sQ_{i-1}(s) + a_{n-i}|$. Since $|s| \geqslant \beta$, $|Q_{i-1}(s)| \geqslant 1$ and (7.61) implies that $|a_{n-i}| \leqslant \beta - 1$. The property $|\rho_1 e^{j\theta_1} + \rho_2 e^{j\theta_2}| \geqslant |\rho_1 - \rho_2|$ then implies

that $|Q_i(s)| \geqslant \beta - (\beta - 1) = 1$. Therefore, $|s| \geqslant \beta$ implies $|P_1(s)| \geqslant 1$ and thus $P_1(s) \neq 0$. Theorem 7.5 is thus valid for monic polynomials. Now, the roots of $P(s)$ are those of the monic polynomial

$$s^n + \frac{a_{n-1}}{a_n} s^{n-1} + \ldots + \frac{a_1}{a_n} s + \frac{a_0}{a_n}, \tag{7.64}$$

and are therefore located inside the disk with centre 0 and radius

$$\beta = 1 + \max(|\frac{a_0}{a_n}|, \ldots, |\frac{a_{n-1}}{a_n}|) = 1 + \frac{\max(|a_0|, \ldots, |a_{n-1}|)}{|a_n|}. \tag{7.65}$$

■

When $P(s)$ depends on $\mathbf{p}$, β becomes a function of $\mathbf{p}$. An inclusion function for $\beta(\mathbf{p})$ is thus

$$[\beta]([\mathbf{p}]) = 1 + \frac{\max(|[a_0]([\mathbf{p}])|, \ldots, |[a_{n-1}]([\mathbf{p}])|)}{|[a_n]([\mathbf{p}])|}. \tag{7.66}$$

If $\overline{\beta}$ is the upper bound of the interval $[\beta]([\mathbf{p}])$, the uncertain polynomial $P(s, [\mathbf{p}])$ has all its roots inside the disk with centre 0 and radius $\overline{\beta}$. $\overline{\beta}$ is thus an upper bound of the cutoff frequency ω_c. The domain for ω is now taken as $[0, \overline{\beta}]$, which is finite. For some types of coefficient functions, it has been shown (Sideris, 1991; Ferreres and Magni, 1996) that the study can be limited to a finite number of frequencies.

Example 7.9 *Consider the polynomial (Barmish and Tempo, 1995)*

$$P(s, \mathbf{p}) = s^3 + a_2(\mathbf{p})s^2 + a_1(\mathbf{p})s + a_0(\mathbf{p}), \tag{7.67}$$

with

$$\begin{aligned} a_0(\mathbf{p}) &= (p_3 + 2)p_3^2 + p_1(\cos 2p_3 - p_2(p_4 - 0.5)^2) + 5, \\ a_1(\mathbf{p}) &= p_2(2\cos 2p_3 + p_1 \cos p_4) + 20, \\ a_2(\mathbf{p}) &= 4p_3 + p_2(1 + 2p_1) + 0.5, \end{aligned} \tag{7.68}$$

and take $\mathbf{p} \in [\mathbf{p}] = [0, 1]^{\times 4}$. *Equation 7.66 yields* $\overline{\beta} = 24\,rad/s$. SIVIAX *proves in* $0.005\,s$ *on a* PENTIUM 233 *that* $P(s, [\mathbf{p}])$ *is robustly stable (see Exercise 11.24, page 331).* ■

In the literature on robust control, Theorem 7.4 is generally presented by introducing the *value set*

$$P(j\omega, [\mathbf{p}]) = \{P(j\omega, \mathbf{p}) \mid \mathbf{p} \in [\mathbf{p}]\}, \tag{7.69}$$

considered as a function of ω. IMAGESP of Chapter 3, page 59, can be used to compute such value sets.

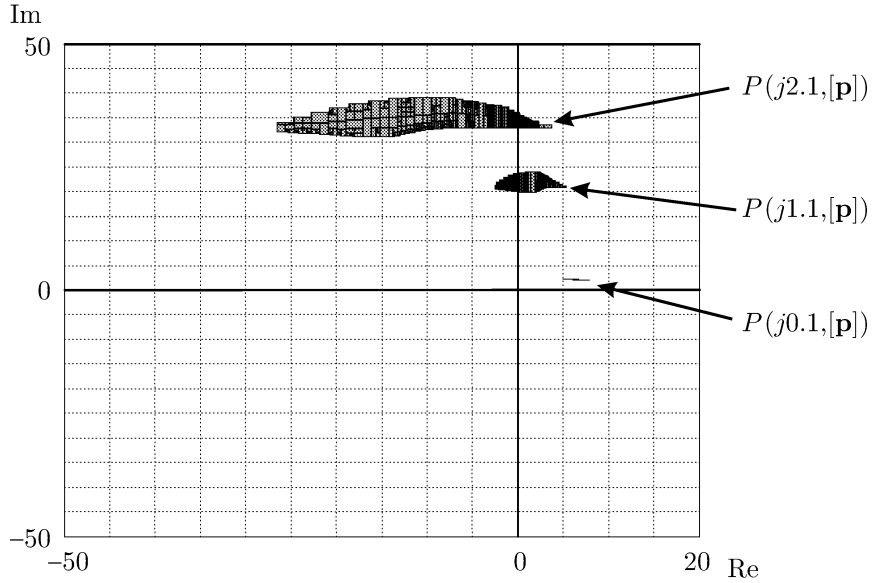


Fig. 7.10. Three outer approximations of value sets for Example 7.9

Example 7.10 *Consider again the problem of Example 7.9. Outer approximations of the value sets* $P(j\omega, [\mathbf{p}])$ *obtained by* ImageSp *for* $\omega = 0.1$ *rad/s,* $\omega = 1.1$ *rad/s and* $\omega = 2.1$ *rad/s are presented in Figure 7.10.* ■

Another formulation of Theorem 7.4 is the *zero-exclusion condition* stated in Frazer and Duncan (1929).

Theorem 7.6 *(zero-exclusion condition) If the coefficients of the polynomial* $P(s, \mathbf{p})$*, are continuous functions of* $\mathbf{p}$ *and if there exists* $\mathbf{p}_0$ *in* $[\mathbf{p}]$ *such that* $P(s, \mathbf{p}_0)$ *is stable, then* $P(s, [\mathbf{p}])$ *is robustly stable if and only if for any* $\omega \geqslant 0$*,* $\mathbf{0} \notin P(j\omega, [\mathbf{p}])$*.* ■

Example 7.11 *Consider again the problem of Examples 7.9 and 7.10.* ImageSp *can be used to compute an outer approximation of the set*

$$P(j\,[0, 24]\,, [\mathbf{p}]) = \{P(j\omega, \mathbf{p}) \mid \omega \in [0, 24]\,, \mathbf{p} \in [\mathbf{p}]\}. \tag{7.70}$$

Computing time can be reduced considerably by avoiding the bisection of any box the image of which does not contain $\mathbf{0}$ *(Adrot, 2000). The resulting subpaving, computed in less than* 2 *s on a* Pentium 233*, is presented in Figure 7.11. It contains the three value sets of Figure 7.10. Since* $P(j\,[0, 24]\,, [\mathbf{p}])$ *does not contain* $\mathbf{0}$*, the zero-exclusion condition implies that* $P(s, [\mathbf{p}])$ *is robustly stable, as it is easy to show that* $P(s, \mathrm{mid}[\mathbf{p}])$ *is stable.* ■

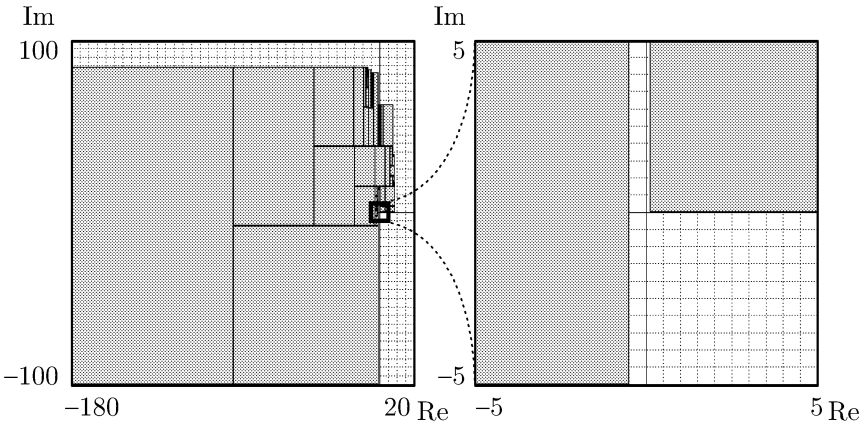


Fig. 7.11. Intersections of the outer approximation of the set of all value sets with the frames $[-180, 20] \times [-100, 100]$ and $[-5, 5] \times [-5, 5]$; $\mathbf{0}$ is excluded

Remark 7.3 *The approach presented here can be used even if* $P(s, \mathbf{p})$ *is not a polynomial. This allows consideration of systems with delays (Barmish, 1994), thus making it possible to deal with a much larger class of problems than the approach based on the Routh criterion.* ■

In order to cast the problem into the framework of optimization (Didrit, 1997), one may transform the zero-exclusion condition $(\forall \omega \geqslant 0, \mathbf{0} \notin P(j\omega, [\mathbf{p}]))$ into the equivalent condition

$$\eta([\mathbf{p}]) \triangleq \min_{\mathbf{p} \in [\mathbf{p}], \omega \geqslant 0} |P(j\omega, \mathbf{p})|^2 > 0. \tag{7.71}$$

The dimension of search space is again $\dim \mathbf{p} + 1$. The domain $[0, \overline{\beta}]$ for ω is obtained by computing an upper bound of the cutoff frequency as in (7.66). Note that $\eta([\mathbf{p}])$ can be viewed as a stability margin.

Example 7.12 *Consider again the problem of Examples 7.9, 7.10 and 7.11. Barmish and Tempo (1995) show how to construct an outer approximation of the value set* $P(j\omega, [\mathbf{p}])$ *for a given value of* ω*. From ten such approximations obtained for ten values of* ω*, they conclude that the family of polynomials* $P(s, [\mathbf{p}])$ *should be robustly stable, without proving it rigorously. Recall that an upper bound of the cutoff frequency is* $\overline{\beta} = 24$ *rad/s. The search domain is* $[\mathbf{x}] = [0, 24] \times [0, 1]^{\times 4}$*. Since the minimum of the cost function* $c(\omega, \mathbf{p}) = |P(j\omega, \mathbf{p})|^2$ *is independent of* p_4*, no bisection of* $[p_4]$ *is allowed (Didrit, 1997). After* 275 *bisections and in* 5.16 *s on a* PENTIUM 90*, Hansen's algorithm, presented in Section 5.5.2, page 121, isolates four solution boxes. Each of them contains one* $\mathbf{p}$ *of the form* $\mathbf{p} = (1, 0, 0, p_4)^T$*. This may indicate that the solution is on the boundary of the domain of interest, which poses no problem to the algorithm. The associated frequency domain for* ω *is* $[4.4609, 4.4611]$*;*

$\eta([\mathbf{p}])$ *is proved to belong to* $[15.799, 15.802]$. *We have thus not only* proved *that* $P(s, \mathbf{p})$ *is robustly stable, but also quantified a stability margin.* ■

The next example shows how Γ-stability can be studied when the region Γ of interest is not a half-plane.

Example 7.13 *For the polynomial*

$$P(s, \mathbf{p}) = s^3 + a_2(\mathbf{p})s^2 + a_1(\mathbf{p})s + a_0(\mathbf{p}), \tag{7.72}$$

where

$$\begin{aligned} a_0(\mathbf{p}) &= \sin(p_2)e^{p_2} + p_1 p_2 - 1, \\ a_1(\mathbf{p}) &= 2p_1 + 0.2p_1 e^{p2}, \\ a_2(\mathbf{p}) &= p_1 + p_2 + 4, \end{aligned} \tag{7.73}$$

two of the coefficient functions are neither linear nor polynomial. Using an affine outer approximation, Amato et al. (1995) have proved that this polynomial is stable for all parameter vectors in $[\mathbf{p}] = [1, 1.5]^{\times 2}$. *We shall now study the robust* Γ*-stability of* $P(s, [\mathbf{p}])$, *where* Γ *is a cone symmetrical with respect to the real axis, with vertex* $\mathbf{0}$ *and half angle* $\phi = \frac{\pi}{3}$. *The reasoning to be followed is depicted in Figure 7.12.* Γ *is the intersection of the half-planes* $\mathbb{D}^{-\frac{\pi}{3}}$ *and* $\mathbb{D}^{\frac{\pi}{3}}$. *Since* $P(s, \mathbf{p})$ *has real coefficients, its roots are symmetrical with respect to the real axis and a polynomial is* Γ*-stable if and only if it is* $\mathbb{D}^{-\frac{\pi}{3}}$*-stable, in which case it is also* $\mathbb{D}^{\frac{\pi}{3}}$*-stable. We shall thus consider only the problem of* $\mathbb{D}^{-\frac{\pi}{3}}$*-stability. The polynomial is* $\mathbb{D}^{-\frac{\pi}{3}}$*-stable if and only if*

$$P(s) = 0, s \notin \mathbb{D}^{-\frac{\pi}{3}} \tag{7.74}$$

has no solution for s. *Set* $z = se^{-j\frac{\pi}{6}}$; *(7.74) becomes equivalent to*

$$P(ze^{j\frac{\pi}{6}}) = 0, z \notin \mathbb{C}^-, \tag{7.75}$$

where $\mathbb{C}^-$ *is the set of all complex numbers with strictly negative real parts, and the polynomial is* $\mathbb{D}^{-\frac{\pi}{3}}$*-stable if and only (7.75) has no solution for* z. *Set* $\tilde{P}(z) = P(ze^{j\frac{\pi}{6}})$; *proving the* Γ*-stability of* $\Sigma([\mathbf{p}])$ *then amounts to proving the stability of* $\tilde{P}(z)$. *This can be done by checking that for a given* $\mathbf{p}$ *in* $[\mathbf{p}]$, $\tilde{P}(s, \mathbf{p})$ *is stable (trivial) and that*

$$\min_{\mathbf{p} \in [\mathbf{p}],\ \omega \in \mathbb{R}} |\tilde{P}(j\omega, [\mathbf{p}])|^2 > 0, \tag{7.76}$$

see (7.71). The latter condition can be checked using Hansen's optimization algorithm to minimize the cost function

$$c(\omega, \mathbf{p}) = |\tilde{P}(j\omega, [\mathbf{p}])|^2. \tag{7.77}$$

For $\varepsilon_p = 10^{-3}$ *and* $\varepsilon_c = 10^{-5}$, *after* 230 *iterations executed in* 2.1 *s on a* PENTIUM 90, *one solution box is generated, the width of which is smaller than* ε_p. *This solution box contains the point with coordinates* $p_1 = 1.5$, $p_2 = 1$ *and* $\omega = 0.678$ *rad/s. The algorithm also returns a small interval that contains*

the minimum of the cost function $c(.)$, the width of which is less than ε_c. This minimum, found to be approximately equal to 0.217*, is guaranteed to be strictly positive. The robust stability of $\tilde{P}(z, [\mathbf{p}])$ and thus the Γ-stability of $P(s, [\mathbf{p}])$ are therefore established.* ■

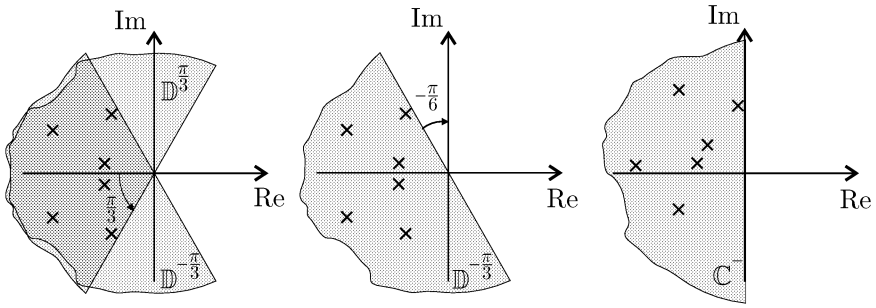


Fig. 7.12. Transformation of a Γ-stability problem into a problem of stability for a polynomial with complex coefficients

Casting the zero-exclusion condition into such an optimization framework does not make it possible to benefit from the graphical interpretation of the value sets, but it allows consideration of any type of parametric dependency. An approach similar to the one followed here can also be used to detect whether the behaviour of the uncertain system is acceptable in the sense of many robust performance criteria.

7.4.4 Robust stability margins

To illustrate the notion of robust stability margin, consider a system with a single input and a single output, defined by its transfer function

$$G(s) = \frac{N(s)}{D(s)} = \frac{s+1}{s^2 + 0.4s + 1}. \tag{7.78}$$

The unit step response of this system is presented in Figure 7.13. (The unit step response is the output of the system when $u = 0$ for $t < 0$ and $u = 1$ for $t \geqslant 0$.)

Put this system inside a negative feedback loop as indicated on Figure 7.14. Such feedback loops are commonly used to counteract the effect of external perturbations by adapting the input of G based on the deviation between what is achieved (y) and what is desirable (u). Let $H(s)$ be the transfer function of the resulting closed-loop system. For zero initial condition, the Laplace transform $y(s)$ of the system output $y(t)$ satisfies $y(s) = G(s)(u(s) - y(s))$, or equivalently

$$y(s) = \frac{G(s)}{1 + G(s)} u(s). \tag{7.79}$$

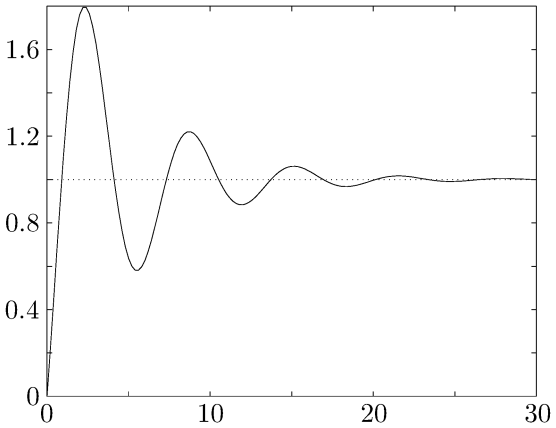


Fig. 7.13. Unit step response of the open-loop system defined by (7.78)

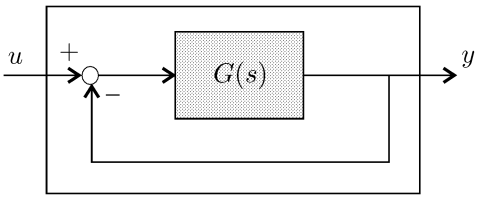


Fig. 7.14. Closed-loop system

The transfer function for the closed-loop system can thus be written as

$$H(s) = \frac{G(s)}{1 + G(s)} = \frac{N(s)}{N(s) + D(s)} = \frac{s + 1}{s^2 + 1.4s + 2}. \tag{7.80}$$

The unit step response of this closed-loop system is presented in Figure 7.15. Here $H(s)$ is stable, but if the coefficients of $G(s)$ are moved continuously, $H(s)$ may become unstable. It will do so when any of the roots of its denominator crosses the imaginary axis, *i.e.*, when $\exists\omega \mid N(j\omega) + D(j\omega) = 0$, or equivalently when

$$\exists\omega \mid G(j\omega) = \frac{N(j\omega)}{D(j\omega)} = -1. \tag{7.81}$$

This amounts to saying that $H(s)$ becomes unstable when the set

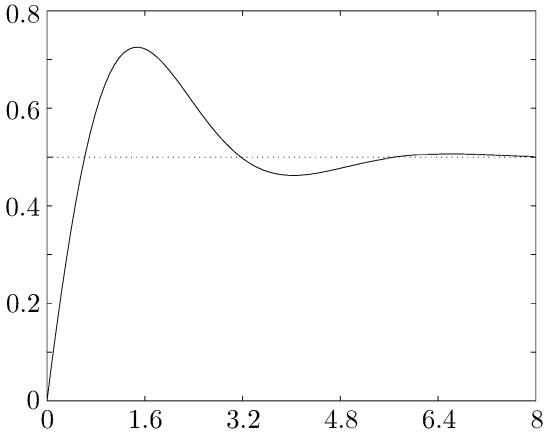


Fig. 7.15. Unit step response of the closed-loop system

$$G(j\mathbb{R}) = \{G(j\omega) \mid \omega \in \mathbb{R}\} \tag{7.82}$$

crosses the critical point -1. It is thus possible to study the stability of the closed-loop system $H(s)$ by analyzing features of the open-loop system $G(s)$. Of special importance is the location of the graph of the set $G(j\mathbb{R})$, called the *Nyquist plot* of G, relative to the critical point. Note that this critical point has a modulus equal to 1 and a phase angle equal to $-\pi$. The Nyquist plot of the system defined by (7.78) is given in Figure 7.16.

Assume that the nominal model $G(s)$ is such that $H(s)$ is stable. To evaluate the robustness of the stability of $H(s)$ to a modification of $G(s)$, perturb the open-loop system until it becomes unstable by multiplying $G(s)$ by a complex coefficient $\rho e^{j\theta}$, with $\rho > 0$. This is illustrated by Figure 7.17. The open-loop transfer function is now $\tilde{G}(s) = \rho e^{j\theta} G(s)$, and the corresponding closed-loop transfer function will be denoted by $\tilde{H}(s)$. For $\rho = 1$ and $\theta = 0$, the system is not perturbed, *i.e.*, $\tilde{G}(s) = G(s)$. The gain ρ is often measured in dB, with $\rho_{\mathrm{dB}} = 20 \log_{10}(\rho)$. When $\rho = 0, 1$ or ∞, $\rho_{\mathrm{dB}} = -\infty, 0$ or ∞, respectively, so $\rho_{\mathrm{dB}} = 0$ when the gain is not perturbed. The perturbed open-loop transfer function can be written as

$$\tilde{G}(s) = 10^{\frac{\rho_{\mathrm{dB}}}{20}} e^{j\theta} G(s). \tag{7.83}$$

If ρ_{dB} or θ depart from their nominal zero values, $\tilde{H}$ becomes unstable when $\tilde{G}(j\mathbb{R})$ crosses the critical point -1, *i.e.*, when

$$\exists \omega \in \mathbb{R} \mid 10^{\frac{\rho_{\mathrm{dB}}}{20}} e^{j\theta} G(j\omega) = -1. \tag{7.84}$$

Take first $\theta = 0$, and define the *gain margin* m_{G} as the smallest value of $|\rho_{\mathrm{dB}}|$ such that $\tilde{H}(s)$ becomes unstable. This means that if the gain of the open-loop system is modified in such a way that the absolute value of

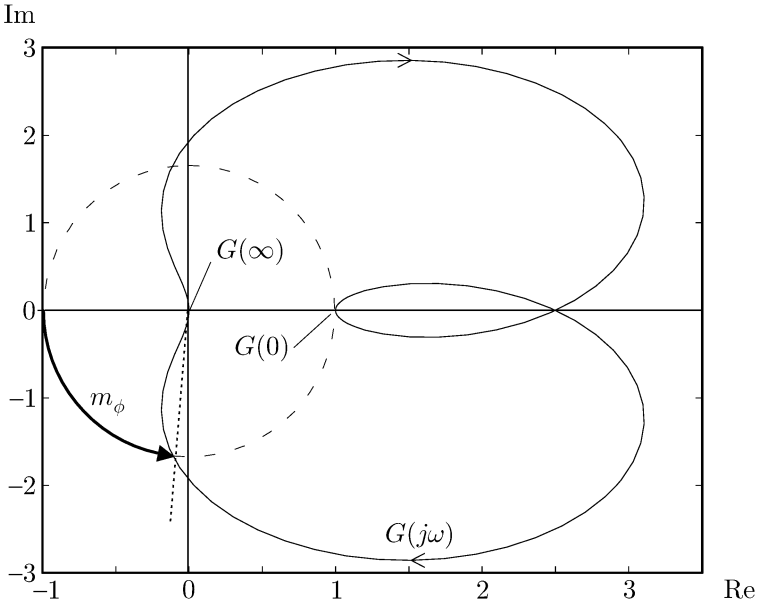


Fig. 7.16. Nyquist plot of the open-loop system $G(s)$; its location with respect to the critical point -1 provides information on the stability of the closed-loop system $H(s)$; the angle m_ϕ represents the phase margin of $G(s)$

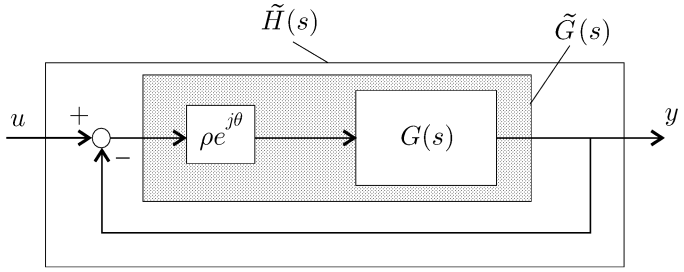


Fig. 7.17. Perturbed system

ρ_{dB} remains less than m_{G}, then the closed-loop system remains stable. From (7.84),

$$m_{\mathrm{G}} \triangleq \min \left\{ |\rho_{\mathrm{dB}}| \mid \exists \omega \in \mathbb{R},\ 10^{\frac{\rho_{\mathrm{dB}}}{20}} G(j\omega) = -1 \right\}. \tag{7.85}$$

Since $10^{\frac{\rho_{\mathrm{dB}}}{20}} G(j\omega) = -1 \Leftrightarrow \rho_{\mathrm{dB}} = 20 \log_{10} \frac{-1}{G(j\omega)}$, the gain margin may equivalently be computed as

$$\begin{cases} m_{\mathrm{G}} = \min \left| 20 \log_{10} \dfrac{-1}{G(j\omega, \mathbf{p})} \right|, \\ \text{subject to } 10^{\frac{\rho_{\mathrm{dB}}}{20}} G(j\omega) = -1. \end{cases} \tag{7.86}$$

Now, in (7.86) the real number ρ_{dB} is a free variable and the constraint $10^{\frac{\rho_{\mathrm{dB}}}{20}} G(j\omega) = -1$ is thus equivalent to stating that $G(j\omega)$ is real and strictly negative, *i.e.*, that $\mathrm{Im}(G(j\omega)) = 0$ and $\mathrm{Re}(G(j\omega)) < 0$. Therefore

$$\begin{cases} m_{\mathrm{G}} = \min \left| 20 \log_{10} \dfrac{-1}{G(j\omega, \mathbf{p})} \right|, \\ \text{subject to } \left(\mathrm{Im}(G(j\omega)) = 0\right) \wedge \left(\mathrm{Re}(G(j\omega)) < 0\right). \end{cases} \tag{7.87}$$

Moreover, since $G(j\omega) = G(-j\omega)$ when the constraint $\mathrm{Im}(G(j\omega)) = 0$ is satisfied, the search can be limited to $\omega \geqslant 0$. Finding m_{G} thus amounts to solving the constrained minimization problem

$$\begin{cases} m_{\mathrm{G}} = \min \left| 20 \log_{10} \dfrac{-1}{G(j\omega, \mathbf{p})} \right|, \\ \text{subject to } \left(\mathrm{Im}(G(j\omega)) = 0\right) \wedge \left(\mathrm{Re}(G(j\omega)) < 0\right) \wedge \left(\omega \geqslant 0\right). \end{cases} \tag{7.88}$$

Take now $\rho_{\mathrm{dB}} = 0$, and define the *phase margin* m_ϕ as the smallest value of $|\theta|$ such that $\tilde{H}(s)$ becomes unstable. From (7.84),

$$m_\phi \triangleq \min \left\{ |\theta| \mid \exists \omega \in \mathbb{R},\ e^{j\theta} G(j\omega) = -1 \right\}. \tag{7.89}$$

Since $G(j\omega)$ and $G(-j\omega)$ are conjugate, θ can be taken as positive,

$$m_\phi = \min_{\theta \geqslant 0} \left\{ \theta \mid \exists \omega \in \mathbb{R},\ e^{j\theta} G(j\omega) = -1 \right\}. \tag{7.90}$$

Finding m_ϕ thus amounts to solving the constrained minimization problem

$$\begin{cases} m_\phi = \min \theta, \\ \text{subject to } \left(\theta \geqslant 0\right) \wedge \left(e^{j\theta} G(j\omega) = -1\right). \end{cases} \tag{7.91}$$

Assume now that the transfer function of the system to be put in the loop is $G(s, \mathbf{p})$, where $\mathbf{p}$ is an uncertain parameter vector. Assume that for any $\mathbf{p}$ in $[\mathbf{p}]$ the closed-loop system is stable, which can readily be checked with the techniques described in Section 7.2.2, page 189. The notions of gain and phase margins can be extended to this context by considering the worst case, *i.e.*, the value of $\mathbf{p}$ in $[\mathbf{p}]$ such that the margin under consideration is the smallest. This leads to defining the *robust gain margin* as

$$m_{\mathrm{G}}([\mathbf{p}]) = \min_{\mathbf{p} \in [\mathbf{p}]} \ \min_{\substack{\mathrm{Im}(G(j\omega, \mathbf{p})) = 0 \\ \mathrm{Re}(G(j\omega, \mathbf{p})) < 0 \\ \omega \geqslant 0}} \left| 20 \log_{10} \frac{-1}{G(j\omega, \mathbf{p})} \right|, \tag{7.92}$$

and the *robust phase margin* as

$$m_\phi([\mathbf{p}]) = \min_{\mathbf{p}\in[\mathbf{p}]} \min_{\substack{\theta \geqslant 0 \\ e^{j\theta}G(j\omega,\mathbf{p}) = -1}} \theta. \tag{7.93}$$

These quantities can be computed with OPTIMIZE, presented in Chapter 5, page 119.

Example 7.14 *Consider the system*

$$G(s,\mathbf{p}) = \frac{(2s+1)\left((1+p_1^2)s + e^{-p_2}\right)}{D(s,\mathbf{p})}, \tag{7.94}$$

where

$$\begin{aligned} D(s,\mathbf{p}) = {} & s\,(s+5)\,(\frac{s}{1+p_2^2} + 1 + \cos^2 5p_1) \\ & *(s^2 + \sqrt{p_2}(3 + 2\sin 3p_1)s + p_2^2 - 2p_2 + 2). \end{aligned} \tag{7.95}$$

For $\mathbf{p} \in [-1,1] \times [0.3, 1.5]$ *and* $\varepsilon_\mathrm{p} = \varepsilon_\mathrm{c} = 0.005$, OPTIMIZE *finds that*

$$\begin{aligned} &16.405\ dB \leqslant m_\mathrm{G}([\mathbf{p}]) \leqslant 16.891\ dB \\ &1.537\ rad \leqslant m_\phi([\mathbf{p}]) \leqslant 1.544\ rad. \end{aligned} \tag{7.96}$$

On a PENTIUM 90, *it takes* 23.8 *s and* 1495 *iterations to generate* 56 *solution boxes for the gain margin, and* 72.7 *s and* 6349 *iterations to generate* 1384 *solution boxes for the phase margin (Didrit, 1997).* ■

7.4.5 Stability radius

To characterize the stability margin of $\Sigma(\mathbf{p})$ with respect to the uncertainty on $\mathbf{p}$ around some nominal value $\mathbf{p}^0$, Safonov and Athans (1981) and Doyle (1982) have independently defined the notion of *stability radius*.

Definition 7.1 *The stability radius of* $\Sigma(\mathbf{p})$ *at* $\mathbf{p}^0$ *is*

$$\begin{aligned} \rho &\triangleq \sup\{\eta \geqslant 0 \mid \Sigma(\mathbf{p}) \text{ is stable for all } \mathbf{p} \in [\mathbf{p}](\eta)\} \\ &= \min\{\eta \geqslant 0 \mid \Sigma(\mathbf{p}) \text{ is unstable for one } \mathbf{p} \in [\mathbf{p}](\eta)\}, \end{aligned} \tag{7.97}$$

where $[\mathbf{p}](\eta)$ *is the box with centre* $\mathbf{p}^0$ *such that the width of its* j*th component satisfies* $w([\underline{p}_j, \overline{p}_j]) = 2\eta w_j$ *for some prespecified positive number* w_j. ■

The quantity η is thus the radius of the hypercube $[\mathbf{p}](\eta)$ in the L_∞ norm weighted by the w_js. Figure 7.18 illustrates this notion for dim $\mathbf{p} = 2$ and $w_1 = w_2 = 1$. Now, since

$$\mathbf{p} \in [\mathbf{p}](\eta) \Leftrightarrow \forall j \in \{1,\ldots,n_\mathrm{p}\}, (p_j^0 - \eta w_j \leqslant p_j \leqslant p_j^0 + \eta w_j) \tag{7.98}$$

and since

$$\begin{aligned}\Sigma(\mathbf{p}) \text{ is unstable } &\Leftrightarrow \exists i \text{ such that } r_i(\mathbf{p}) \leqslant 0 \\ &\Leftrightarrow (r_1(\mathbf{p}) \leqslant 0) \vee \cdots \vee (r_n(\mathbf{p}) \leqslant 0)\,,\end{aligned} \tag{7.99}$$

where $\vee$ stands for the Boolean operator OR, the stability radius can also be defined as

$$\left\{\begin{array}{l} \rho = \min\limits_{\eta \geqslant 0} \eta, \\ \text{subject to } \left\{\begin{array}{l} ((r_1(\mathbf{p}) \leqslant 0) \vee \cdots \vee (r_n(\mathbf{p}) \leqslant 0)) \\ \quad \wedge \left(\forall j \in \{1, \ldots, n_\mathrm{p}\}, \left\{\begin{array}{l} p_j^0 - \eta w_j - p_j \leqslant 0 \\ -p_j^0 - \eta w_j + p_j \leqslant 0 \end{array}\right.\right) \end{array}\right. \end{array}\right., \tag{7.100}$$

where $\wedge$ stands for the Boolean operator AND. Because of the operator $\vee$ involved in the constraints, OPTIMIZE cannot be applied directly.

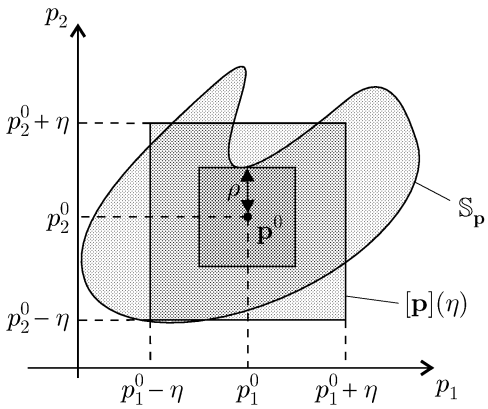


Fig. 7.18. Stability radius ρ at $\mathbf{p}^0$; $\mathbb{S}_\mathrm{p}$ is the stability domain.

An equivalent definition of ρ is

$$\rho = \min\{\rho_1, \ldots, \rho_n\}, \tag{7.101}$$

with

$$\left\{\begin{array}{l} \rho_i = \min\limits_{\eta \geqslant 0} \eta, \\ \text{subject to } \left\{\begin{array}{l} (r_i(\mathbf{p}) \leqslant 0) \\ \wedge \left(\forall j \in \{1, \ldots, n_\mathrm{p}\}, \left\{\begin{array}{l} p_j^0 - \eta w_j - p_j \leqslant 0 \\ -p_j^0 - \eta w_j + p_j \leqslant 0 \end{array}\right.\right) \end{array}\right. \end{array}\right. . \tag{7.102}$$

Now, the ρ_is can be computed using OPTIMIZE, because the associated constraints are related with $\wedge$ operators. The stability radius ρ is then obtained by taking the smallest of the ρ_is.

The number of minimizations to be performed is equal to the number of inequalities to be checked in the stability test. If the Routh criterion is used, this number is equal to n. The Routh–Hurwitz criterion can reduce this number to three, as stated by the following theorem (Kolev, 1993b; Malan et al., 1997).

Theorem 7.7 *The family of polynomials*

$$P(s,[\mathbf{p}]) = a_n([\mathbf{p}])s^n + a_{n-1}([\mathbf{p}])s^{n-1} + \cdots + a_1([\mathbf{p}])s + a_0([\mathbf{p}]) \tag{7.103}$$

is robustly stable if $\Sigma(\mathrm{mid}([\mathbf{p}]))$ *is stable and if for any* $\mathbf{p}$ *in* $[\mathbf{p}]$

$$\begin{cases} q_1(\mathbf{p}) = \quad a_0(\mathbf{p}) > 0, \\ q_2(\mathbf{p}) = \quad a_n(\mathbf{p}) > 0, \\ q_3(\mathbf{p}) = D_{n-1}(\mathbf{p}) > 0, \end{cases} \tag{7.104}$$

where $D_{n-1}(\mathbf{p})$ *is the* $(n-1)$*th Hurwitz determinant associated with* $P(s,\mathbf{p})$*:*

$$D_{n-1}(\mathbf{p}) = \begin{vmatrix} a_1(\mathbf{p}) & a_3(\mathbf{p}) & a_5(\mathbf{p}) & \cdots & a_{2n-1}(\mathbf{p}) \\ a_0(\mathbf{p}) & a_2(\mathbf{p}) & a_4(\mathbf{p}) & \cdots & a_{2n-2}(\mathbf{p}) \\ 0 & a_1(\mathbf{p}) & a_3(\mathbf{p}) & \cdots & a_{2n-3}(\mathbf{p}) \\ 0 & a_0 & a_2(\mathbf{p}) & \cdots & a_{2n-4}(\mathbf{p}) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & a_{n-1}(\mathbf{p}) \end{vmatrix}. \tag{7.105}$$

■

The stability radius ρ satisfies

$$\begin{cases} \rho = \min\limits_{\eta \geqslant 0} \eta, \\ \text{subject to } \begin{cases} ((q_1(\mathbf{p}) \leqslant 0) \vee (q_2(\mathbf{p}) \leqslant 0) \vee (q_3(\mathbf{p}) \leqslant 0)) \\ \quad \wedge \left(\forall j \begin{cases} p_j^0 - \eta w_j - p_j \leqslant 0 \\ -p_j^0 - \eta w_j + p_j \leqslant 0 \end{cases} \right), \end{cases} \end{cases} \tag{7.106}$$

or equivalently

$$\begin{cases} \rho = \min\{\rho_1, \rho_2, \rho_3\}, \\ \rho_i = \min\limits_{\eta \geqslant 0} \eta, \\ \text{subject to } (q_i(\mathbf{p}) \leqslant 0) \wedge \left(\forall j \begin{cases} p_j^0 - \eta w_j - p_j \leqslant 0 \\ -p_j^0 - \eta w_j + p_j \leqslant 0 \end{cases} \right). \end{cases} \tag{7.107}$$

The number of minimizations to be performed is now equal to three instead of n.

When the coefficient function is affine, the problem can be solved using linear programming (Tesi and Vicino, 1989). When this function is polynomial, methods based on generalized geometric programming, also called *signomial programming,* have been used (Vicino et al., 1990). The problem of computing the stability radius has been proved to be NP-hard in Braatz et al. (1994).

The efficiency of interval techniques for computing stability radii will now be demonstrated on two examples.

Example 7.15 *The polynomial*

$$P(s,\mathbf{p}) = s^3 + (p_1+p_2+2)s^2 + (p_1+p_2+2)s \\ +2p_1p_2 + 6p_1 + 6p_2 + 2 + \sigma^2,$$

was considered in Examples 7.3 and 7.5, pages 198 and 202. When p_1 and p_2 are positive, this polynomial is stable for all parameter vectors outside the disk with centre $\mathbf{p}^0 = (1,1)^T$ and radius σ. Hansen's optimization algorithm is now used to compute the stability radius for different values of σ based on (7.107). As in Murdock et al. (1991), Psarris and Floudas (1995) and Malan et al. (1997), the nominal value for $\mathbf{p}$ is taken as $\mathbf{p}^0 = (1.4, 0.85)^T$, and the weights are $w_1 = 1.1$ and $w_2 = 0.85$. The box $[\mathbf{p}](\eta)$ is thus defined by

$$\begin{aligned} 1.4 - 1.1\eta \ &\leqslant p_1 \leqslant \ 1.4 + 1.1\eta, \\ 0.85 - 0.85\eta &\leqslant p_2 \leqslant 0.85 + 0.85\eta. \end{aligned} \tag{7.108}$$

The results, obtained on a PENTIUM 90 *for different values of σ are given in Table 7.2. They are similar to those of Malan et al. (1997), but our approach can deal with a general non-linear parametric dependency, as illustrated by the next example.* ■

Table 7.2. Stability radii for various values of σ (Example 7.15)

σ	10^{-1}	10^{-3}	10^{-5}	10^{-7}
ε_{p} and ε_{c}	10^{-5}	10^{-5}	10^{-5}	10^{-7}
Number of iterations	66	113	55	63
Computing time (s)	0.44	0.55	0.44	0.49
Number of solution boxes	1	5	1	2
Stability radius	0.2727	0.3627	0.3636	0.3636

Example 7.16 *Consider the polynomial*

$$P(s,\mathbf{p}) = s^3 + a_2(\mathbf{p})s^2 + a_1(\mathbf{p})s + a_0(\mathbf{p}), \tag{7.109}$$

with

$$\begin{aligned} a_0(\mathbf{p}) &= \sin(p_2)e^{p_2} + p_1p_2 - 1, \\ a_1(\mathbf{p}) &= 2p_1 + 0.2p_1e^{p_2}, \\ a_2(\mathbf{p}) &= p_1 + p_2 + 4. \end{aligned} \tag{7.110}$$

The coefficient function $\mathbf{a}(\mathbf{p})$ *is neither linear nor polynomial. A computation of the stability radius based on (7.107), using Hansen's optimization algorithm for* $\varepsilon_{\mathrm{p}} = \varepsilon_{\mathrm{c}} = 10^{-5}$*, finds it to be approximatively equal to* 2.025 *at* $\mathbf{p}^0 = (1.5, 1.5)^T$ *(Didrit, 1997). This is consistent with the fact that* $P(s,\mathbf{p})$ *is stable for any* $\mathbf{p}$ *in* $[\mathbf{p}] = [1,2]^{\times 2}$ *(Amato et al., 1995).* ■

7.5 Controller Design

Although relatively few papers have been dedicated to controller design via interval analysis (Kolev et al., 1988; Kearfott, 1989b; Khlebalin, 1992; Kolev, 1993a; Jaulin and Walter, 1996; Malan et al., 1997), interest is growing, as illustrated by a recent special issue of *Reliable Computing* (Garloff and Walter, 2000). In this section, the application of the interval solvers presented in Chapter 5 to the tuning of the parameters of controllers will be demonstrated.

Consider a linear system to be controlled. Assume first that the system is perfectly known and that no uncertain parameters are involved. The parameter vector $\mathbf{c}$ of the controller can be chosen arbitrarily in a box $[\mathbf{c}]$. Denote by $\mathbf{r}(\mathbf{c},\delta)$ the δ-Routh function associated with the controlled system. The controller that maximizes the stability degree is given by

$$\mathbf{c} = \arg\max_{\mathbf{c}\in[\mathbf{c}]} \max_{\mathbf{r}(\mathbf{c},\delta)\geqslant\mathbf{0}} \delta. \tag{7.111}$$

Example 7.17 *Consider again the system* $\Sigma(\mathbf{p})$ *of Example 7.6, page 202. Assume now that* $\mathbf{p}$ *can be tuned so as to maximize the stability degree, so* $\mathbf{p}$ *plays here the role of* $\mathbf{c}$*. The characteristic polynomial of* $\Sigma(\mathbf{p})$ *is*

$$P(s,\mathbf{p}) = s^3 + s^2 + (p_1^2 + p_2^2 + 1)s + 1, \tag{7.112}$$

where the parameters p_1 *and* p_2 *can be chosen arbitrarily in* $[\mathbf{p}] = [-7, 1.3] \times [-1, 2.5]$*. Recall that for any* $\mathbf{p}$*,* $\Sigma(\mathbf{p})$ *is* δ*-unstable for* $\delta \geqslant \frac{1}{3}$*. For* $0 < \delta < \frac{1}{3}$*,* $P(s,\mathbf{p})$ *is* δ*-stable for all vectors* $\mathbf{p}$ *inside the region located between the circles with radii*

$$\underline{\sigma}(\delta) = \sqrt{\frac{4\delta(2\delta^2 - 2\delta + 1)}{-2\delta + 1}} \text{ and } \overline{\sigma}(\delta) = \sqrt{\frac{-\delta^3 + \delta^2 - \delta + 1}{\delta}} \tag{7.113}$$

The maximal stability degree is obtained when the inner and outer circles merge. This happens for $\delta = \frac{1}{3}$*. Then* $\underline{\sigma}(\delta) = \overline{\sigma}(\delta) = \frac{2\sqrt{5}}{3}$*. There are thus*

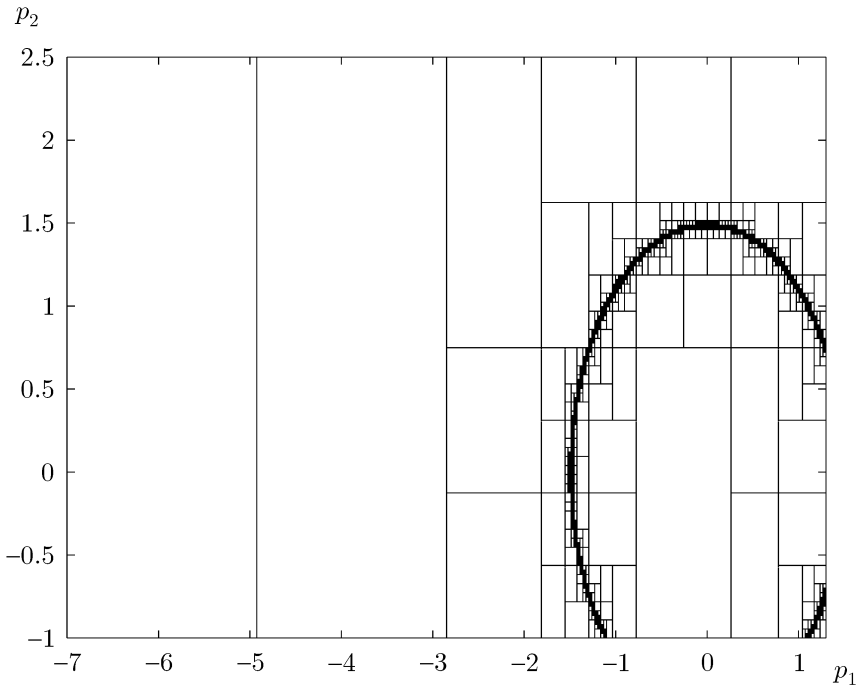


Fig. 7.19. Paving generated to characterize the set of all the maximizers of the stability degree of Example 7.17; all maximizers are in the black subpaving

infinitely many values of $\mathbf{p}$ *that maximize the stability degree of the closed-loop system. This is due to the fact the system is overparametrized. For* $\varepsilon_c = 0.05$ *and* $\varepsilon_\delta = 0.001$*, after* 588 *iterations in* 2.5 *s on a* PENTIUM 90*, Hansen's optimization algorithm yields* $\delta_M \in [0.3333, 0.3339]$*. All maximizers are in the black subpaving of Figure 7.19.* ■

Assume now that the model of the system to be controlled depends on a vector $\mathbf{p}$ of uncertain parameters. Two types of parameters have then to be dealt with, namely $\mathbf{p}$ and the tuning parameters $\mathbf{c}$ of the controller.

Consider a closed-loop system $\Sigma(\mathbf{p}, \mathbf{c})$, the forward path of which consists of a controller $C(s, \mathbf{c})$ cascaded with an uncertain parametric model $G(s, \mathbf{p})$, $\mathbf{p} \in [\mathbf{p}]$ (see Figure 7.20). The problem to be studied is the computation of the set $\mathbb{S}_c$ of the vectors $\mathbf{c}$ that maximize the stability degree in the worst case. This set satisfies

$$\mathbb{S}_c = \arg \max_{\mathbf{c}\in[\mathbf{c}]} \min_{\mathbf{p}\in[\mathbf{p}]} \max_{\mathbf{r}(\mathbf{p},\mathbf{c},\delta)\geqslant\mathbf{0}} \delta. \tag{7.114}$$

The rightmost *max* corresponds to the definition of the stability degree. The *min* ensures the worst-case conditions. The leftmost *max* corresponds to the optimality requirement. If one chooses an element $\mathbf{c}$ in $\mathbb{S}_c$ for the controller,

then one is certain that the stability degree of the controlled system is at least equal to

$$\delta_M^c = \max_{\mathbf{c}\in[\mathbf{c}]} \min_{\mathbf{p}\in[\mathbf{p}]} \max_{\mathbf{r}(\mathbf{p},\mathbf{c},\delta)\geqslant\mathbf{0}} \delta, \tag{7.115}$$

and δ_M^c is the *optimal robust stability degree.*

Example 7.18 (Jaulin, 1994; Jaulin and Walter, 1996; Didrit, 1997). *For the closed-loop system* $\Sigma(\mathbf{p},\mathbf{c})$ *of Figure 7.20,* MINIMAX *gives the results of Table 7.3. In this table,* $\#\overline{\mathbb{S}}_c$ *is the number of boxes in the subpaving* $\overline{\mathbb{S}}_c$ *containing all the values of* $\mathbf{c}$ *corresponding to globally optimal robust controllers,* $[\overline{\mathbb{S}}_c]$ *is the interval hull of* $\overline{\mathbb{S}}_c$*, and* $[\delta_M^c]$ *is an interval guaranteed to contain the associated optimal robust stability degree. The times are indicated for a* PENTIUM 90*, and the order of magnitude of* ε *is* 10^{-3}. ■

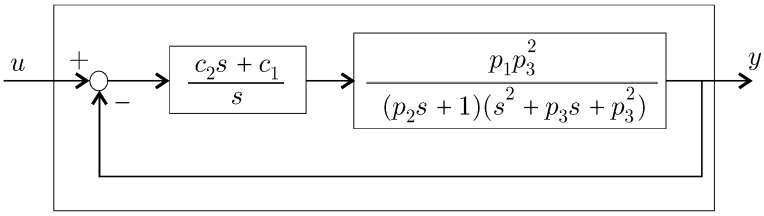


Fig. 7.20. Uncertain system with a PI controller

Table 7.3. Results obtained by MINIMAX for the optimal robust controller

$[\mathbf{p}]$	Time (s)	$\#\overline{\mathbb{S}}_c$	$[\overline{\mathbb{S}}_c]$	$[\delta_M^c]$
$(1,1,1)^T$	5.5	66	$[0.257, 0.273] \times [0.305, 0.354]$	$[0.300, 0.326]$
$[0.99, 1.01]^{\times 3}$	85	69	$[0.239, 0.274] \times [0.264, 0.382]$	$[0.288, 0.299]$
$[0.95, 1.05]^{\times 3}$	339	37	$[0.207, 0.277] \times [0.179, 0.437]$	$[0.261, 0.282]$
$[0.9, 1.1]^{\times 3}$	345	17	$[0.207, 0.254] \times [0.191, 0.367]$	$[0.230, 0.246]$

Remark 7.4 *In practice, it is often sufficient to find one* $\mathbf{c}$ *such that the robust stability degree is higher than some prespecified value* $\underline{\delta}$*. The problem can then be formulated as*

$$\textit{find one } \mathbf{c} \in [\mathbf{c}] \mid \forall \mathbf{p} \in [\mathbf{p}], \mathbf{r}(\mathbf{p},\mathbf{c},\underline{\delta}) > \mathbf{0}. \tag{7.116}$$

This is much simpler than finding the set of all optimal robust controllers, and OPTIMIZE *can be adapted to solve this problem in a more efficient way (Jaulin and Walter, 1996).* ■

7.6 Conclusions

Robust control provides a mine of opportunities for applying interval analysis. Almost any question of interest in this field can be cast into the framework of set inversion, minimax optimization or constrained optimization, and we hope that the examples considered in this chapter have convinced the reader that interval analysis is well equipped to provide pertinent answers.

Of course, interval analysis in its present state cannot handle all problems of robust control, if only because of the curse of dimensionality. Four factors, however, contribute to making problems of practical interest tractable. First, one is in general interested in controllers with only a few tuning parameters (about three for the ubiquitous PID controller). Secondly, parametric uncertainty in the model of the process to be controlled can often be limited to a few dominant factors, frequently connected to physically meaningful parameters. Thirdly, it is usually easy to express the open-loop or closed-loop transfer matrix of the system to be controlled as an explicit function of the uncertain and tuning parameters, which facilitates the derivation of efficient inclusion functions. Lastly, what is needed in general is a vector of satisfactory tuning parameters rather than a characterization of the set of all such vectors.

Assessing how far the complexity barrier can be pushed back in actual controller design is an exciting challenge, which will require the cooperation of control engineers and interval analysts.

8. Robotics

8.1 Introduction

Robots are mechanical systems that are controlled to achieve specific tasks, deemed too repetitive, too dangerous or too difficult for human beings. As a result, robotics is a vast interdisciplinary field, which draws on mathematics, mechanics, control theory, artificial intelligence, ergonomics... This chapter cannot, of course, pretend to exhaustiveness, and will limit itself to illustrating how interval analysis can contribute to the solution of three difficult problems.

The first one, presented in Section 8.2, is the evaluation of all possible configurations of a parallel robot, known as a Stewart–Gough platform, given the lengths of its limbs. This has become a classical benchmark for computer algebra, because it involves solving a rather complicated set of non-linear equations (Raghavan and Roth, 1995). We shall show that interval analysis makes it possible to deal with this type of problem on a personal computer even in the most complex and most general case, and shall stress the advantages of the resulting solution compared to those based on more classical symbolic manipulations.

The second problem, described in Section 8.3, is the planning of a collision-free path for a rigid object in a known environment. It will be solved by combining interval and graph-theoretical tools. This will be illustrated by a planar test case where the object is a non-convex polygon and the obstacles consist of line segments.

The last problem, considered in Section 8.4, is the localization and tracking of a robot from on-board distance measurements in a partially known environment. We shall see how this can be cast into the framework of bounded-error parameter and state estimation described in Chapter 6, and how sensor failures, partially outdated maps and ambiguities due to symmetries in the environment can be taken into account.

L. Jaulin et al., *Applied Interval Analysis*

8.2 Forward Kinematics Problem for Stewart–Gough Platforms

8.2.1 Stewart–Gough platforms

A Stewart–Gough platform consists of a rigid mobile plate linked to a rigid base by six rectilinear limbs, the lengths of which can be controlled (see Figure 8.1). Because the limbs act in parallel on the position and orientation of the mobile plate with respect to the base, this platform is an example of what is known as a parallel robot (as opposed to an articulated arm where the effectors attached to the articulations act in series). This mechanism was proposed by Gough in 1949 for a tyre-testing bed (Gough, 1956), and used by Stewart (1965) to design a flight simulator. Stewart–Gough platforms have now found many other applications in tasks where force and precision are required (Merlet, 1990).

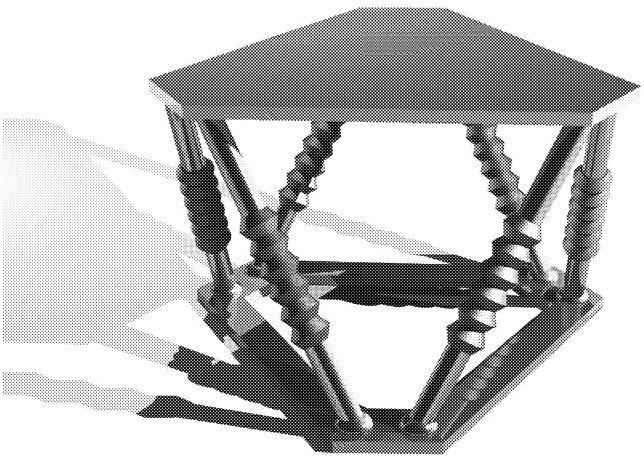


Fig. 8.1. Symbolic rendition of a Stewart–Gough platform

Let $\mathbf{a}(i)$ and $\mathbf{b}(i)$ be the extremities of the ith limb attached to the base and mobile plate, respectively. The *forward kinematic problem* consists in computing *all* possible configurations of the platform, given (1) the coordinates of the $\mathbf{a}(i)$s in a frame attached to the base, (2) the coordinates of the $\mathbf{b}(i)$s in a frame attached to the mobile plate and (3) the lengths y_i of the limbs. Solving this problem is difficult, to the point that it has

become a benchmark for symbolical and numerical computations (Nanua et al., 1990; Lazard, 1992; Mourrain, 1993; Wang and Chen, 1993).

Several methods are available for translating the problem into a set of non-linear equations to be solved. An approach based on the Euler angles will be presented in Sections 8.2.2 and 8.2.3. The solution of the resulting equations by an interval solver will be described in Section 8.2.4. More details can be found in Didrit et al. (1998).

8.2.2 From the frame of the mobile plate to that of the base

Let $\mathcal{R}_0$ be a frame attached to the base, and $\mathcal{R}_4$ be a frame attached to the mobile plate (the reason for such an indexation will become clear shortly). The configuration (position and orientation) of the mobile plate with respect to the base can be represented by the coordinates (c_{x0}, c_{y0}, c_{z0}) in $\mathcal{R}_0$ of a given point $\mathbf{c}$ of the mobile plate together with the three Euler angles ψ, θ and φ, as illustrated by Figure 8.2. The coordinates of the $\mathbf{a}(i)$s are known in $\mathcal{R}_0$ and those of the $\mathbf{b}(i)$s are known in $\mathcal{R}_4$. These coordinates do not depend on the configuration of the platform, but the transformation from $\mathcal{R}_0$ to $\mathcal{R}_4$ does. To compute the lengths of the limbs as functions of the configuration, one must express the $\mathbf{a}(i)$s and $\mathbf{b}(i)$s in the same frame, say $\mathcal{R}_0$. A transformation is thus needed to compute the coordinates of the $\mathbf{b}(i)$s in $\mathcal{R}_0$ from those in $\mathcal{R}_4$. This section is devoted to building this transformation.

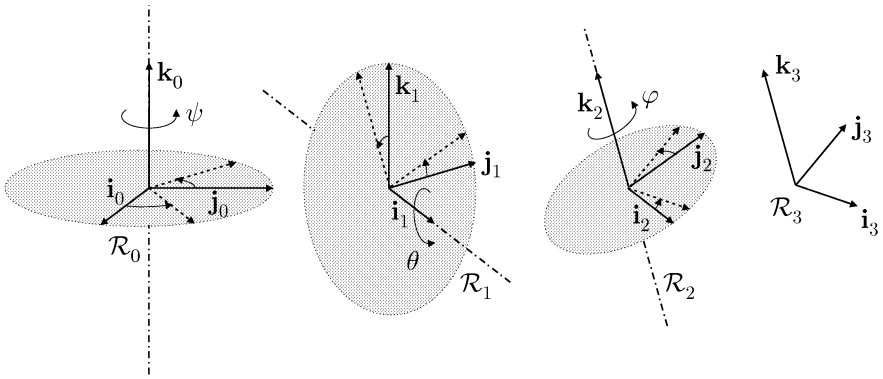


Fig. 8.2. Frame transformation using the Euler angles ψ, θ and φ

Let $\mathcal{R}_1$ be the frame obtained after rotating $\mathcal{R}_0$ around $\mathbf{k}_0$ by an angle ψ (see Figure 8.2). The basis vectors of $\mathcal{R}_1$ can be expressed with respect to those of $\mathcal{R}_0$ as

$$\begin{aligned}
\mathbf{i}_1 &= \cos\psi\ \mathbf{i}_0 + \sin\psi\ \mathbf{j}_0,\\
\mathbf{j}_1 &= -\sin\psi\ \mathbf{i}_0 + \cos\psi\ \mathbf{j}_0,\\
\mathbf{k}_1 &= \mathbf{k}_0,
\end{aligned} \tag{8.1}$$

or in matrix form as

$$\begin{pmatrix} \mathbf{i}_1 \\ \mathbf{j}_1 \\ \mathbf{k}_1 \end{pmatrix} = \mathbf{P}_1 \begin{pmatrix} \mathbf{i}_0 \\ \mathbf{j}_0 \\ \mathbf{k}_0 \end{pmatrix}, \text{ where } \mathbf{P}_1 = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.2}$$

Let $\mathcal{R}_2$ be the frame obtained after rotating $\mathcal{R}_1$ around $\mathbf{i}_1$ by an angle θ. The basis vectors of $\mathcal{R}_2$ satisfy

$$\begin{pmatrix} \mathbf{i}_2 \\ \mathbf{j}_2 \\ \mathbf{k}_2 \end{pmatrix} = \mathbf{P}_2 \begin{pmatrix} \mathbf{i}_1 \\ \mathbf{j}_1 \\ \mathbf{k}_1 \end{pmatrix}, \text{ where } \mathbf{P}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}. \tag{8.3}$$

For $\mathcal{R}_3$, obtained after rotating $\mathcal{R}_2$ around $\mathbf{k}_2$ by an angle φ, we have

$$\begin{pmatrix} \mathbf{i}_3 \\ \mathbf{j}_3 \\ \mathbf{k}_3 \end{pmatrix} = \mathbf{P}_3 \begin{pmatrix} \mathbf{i}_2 \\ \mathbf{j}_2 \\ \mathbf{k}_2 \end{pmatrix}, \text{ where } \mathbf{P}_3 = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.4}$$

Combining (8.2), (8.3) and (8.4) yields

$$\begin{pmatrix} \mathbf{i}_3 \\ \mathbf{j}_3 \\ \mathbf{k}_3 \end{pmatrix} = \mathbf{P}_3\mathbf{P}_2\mathbf{P}_1 \begin{pmatrix} \mathbf{i}_0 \\ \mathbf{j}_0 \\ \mathbf{k}_0 \end{pmatrix}. \tag{8.5}$$

Consider now a vector $\mathbf{v}$ with coordinates (x_0, y_0, z_0) in $\mathcal{R}_0$ and (x_3, y_3, z_3) in $\mathcal{R}_3$. It satisfies

$$\mathbf{v} = x_0\mathbf{i}_0 + y_0\mathbf{j}_0 + z_0\mathbf{k}_0 = x_3\mathbf{i}_3 + y_3\mathbf{j}_3 + z_3\mathbf{k}_3, \tag{8.6}$$

or, in vector form,

$$(x_0\ y_0\ z_0) \begin{pmatrix} \mathbf{i}_0 \\ \mathbf{j}_0 \\ \mathbf{k}_0 \end{pmatrix} = (x_3\ y_3\ z_3) \begin{pmatrix} \mathbf{i}_3 \\ \mathbf{j}_3 \\ \mathbf{k}_3 \end{pmatrix}. \tag{8.7}$$

From (8.5), this implies that

$$(x_0\ y_0\ z_0) \begin{pmatrix} \mathbf{i}_0 \\ \mathbf{j}_0 \\ \mathbf{k}_0 \end{pmatrix} = (x_3\ y_3\ z_3)\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1 \begin{pmatrix} \mathbf{i}_0 \\ \mathbf{j}_0 \\ \mathbf{k}_0 \end{pmatrix}. \tag{8.8}$$

Since $(\mathbf{i}_0, \mathbf{j}_0, \mathbf{k}_0)$ is a basis of $\mathbb{R}^3$, (8.8) is equivalent to

$$(x_0\ y_0\ z_0) = (x_3\ y_3\ z_3)\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1, \tag{8.9}$$

or to

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \mathbf{P}^{\mathrm{T}} \begin{pmatrix} x_3 \\ y_3 \\ z_3 \end{pmatrix}, \tag{8.10}$$

where

$$\mathbf{P}^{\mathrm{T}} = \mathbf{P}_1^{\mathrm{T}}\mathbf{P}_2^{\mathrm{T}}\mathbf{P}_3^{\mathrm{T}}, \tag{8.11}$$

with $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ given by (8.2) to (8.4). Finally, let $\mathcal{R}_4$ be the frame obtained after translating $\mathcal{R}_3$ by the vector $\mathbf{c}$ with coordinates c_{x0}, c_{y0} and c_{z0} in $\mathcal{R}_0$. The coordinates m_{x0}, m_{y0} and m_{z0} of a point $\mathbf{m}$ in $\mathcal{R}_0$ can be obtained from its coordinates m_{x4}, m_{y4} and m_{z4} in $\mathcal{R}_4$ as

$$\begin{pmatrix} m_{x0} \\ m_{y0} \\ m_{z0} \end{pmatrix} = \begin{pmatrix} c_{x0} \\ c_{y0} \\ c_{z0} \end{pmatrix} + \mathbf{P}^{\mathrm{T}} \begin{pmatrix} m_{x4} \\ m_{y4} \\ m_{z4} \end{pmatrix}. \tag{8.12}$$

8.2.3 Equations to be solved

The length y_i of the ith limb of the platform is the Euclidean distance between $\mathbf{a}(i)$ and $\mathbf{b}(i)$. Now, the coordinates of $\mathbf{a}(i)$ are given in $\mathcal{R}_0$, but those of $\mathbf{b}(i)$ are given in $\mathcal{R}_4$. To compute this distance, the coordinates of these points must be expressed in the same frame, e.g., $\mathcal{R}_0$. The coordinates of $\mathbf{b}(i)$ in $\mathcal{R}_0$ can be computed by (8.12). The length y_i can then be obtained as

$$\begin{aligned} y_i &= ||\mathbf{a}(i) - \mathbf{b}(i)||_2 \\ &= \sqrt{(a_{x0}(i) - b_{x0}(i))^2 + (a_{y0}(i) - b_{y0}(i))^2 + (a_{z0}(i) - b_{z0}(i))^2}. \end{aligned} \tag{8.13}$$

The procedure for computing the lengths of the six limbs as functions of the configuration $\mathbf{x} = (c_{x0}, c_{y0}, c_{z0}, \psi, \theta, \varphi)^{\mathrm{T}}$ of the platform is summarized in Table 8.1. The coefficients r_{ij} correspond to the entries of the matrix $\mathbf{P}^{\mathrm{T}}$ in (8.11).

Finding all possible configurations of a Stewart–Gough platform from the knowledge of the lengths of its limbs thus amounts to solving the equation

$$\mathbf{f}(\mathbf{x}) - \mathbf{y} = \mathbf{0} \tag{8.14}$$

for $\mathbf{x}$, where $\mathbf{x}$ is the configuration vector, $\mathbf{f}(\mathbf{x})$ is the vector of the lengths of the limbs as predicted by the algorithm of Table 8.1 and $\mathbf{y}$ is the vector containing the actual numerical values of the lengths $y_i, i = 1, \ldots, 6$, of the limbs. Interval solvers such as those presented in Chapter 5 can be used to solve this problem, as shown in the next section.

Table 8.1. Algorithm for computing the vector $\mathbf{y}_{\mathrm{m}}$ of lengths of the limbs for a given configuration of the platform

Algorithm SGSIMULATOR (in: $c_{x0}, c_{y0}, c_{z0}, \psi, \theta, \varphi$; out $\mathbf{y}_{\mathrm{m}}$)	
1	$r_{11} := \cos\psi\cos\varphi - \sin\psi\cos\theta\sin\varphi;$
2	$r_{12} := -\cos\psi\sin\varphi - \sin\psi\cos\theta\cos\varphi;$
3	$r_{13} := \sin\psi\sin\theta;$
4	$r_{21} := \sin\psi\cos\varphi + \cos\psi\cos\theta\sin\varphi;$
5	$r_{22} := -\sin\psi\sin\varphi + \cos\psi\cos\theta\cos\varphi;$
6	$r_{23} := -\cos\psi\sin\theta;$
7	$r_{31} := \sin\theta\sin\varphi;$
8	$r_{32} := \sin\theta\cos\varphi;$
9	$r_{33} := \cos\theta;$
10	for $i := 1$ to 6
11	$\quad b_{x0}(i) := c_{x0} + r_{11}b_{x4}(i) + r_{12}b_{y4}(i) + r_{13}b_{z4}(i);$
12	$\quad b_{y0}(i) := c_{y0} + r_{21}b_{x4}(i) + r_{22}b_{y4}(i) + r_{23}b_{z4}(i);$
13	$\quad b_{z0}(i) := c_{z0} + r_{31}b_{x4}(i) + r_{32}b_{y4}(i) + r_{33}b_{z4}(i);$
14	$\quad y_i := \sqrt{(a_{x0}(i) - b_{x0}(i))^2 + (a_{y0}(i) - b_{y0}(i))^2 + (a_{z0}(i) - b_{z0}(i))^2};$
15	$\mathbf{y}_{\mathrm{m}} := (y_1, \ldots, y_6)^{\mathrm{T}}.$

8.2.4 Solution

No analytic solution is available for (8.14) in the general case. It has been shown (Lazard, 1993; Mourrain, 1993; Raghavan, 1993; Wampler, 1996), that there are at most 40 complex solutions in the general case, a bound that can be decreased for special configurations (Lee and Roth, 1993; Faugère and Lazard, 1995). Of course, only the *real* solutions are interesting in practice. Since (8.14) involves trigonometric functions, formal elimination methods for the solution of sets of polynomial equations, such as those based on the construction of a Gröbner basis (Lazard, 1992, 1993), do not apply directly. However, (8.14) can be transformed into a set of nine polynomial equations, at the cost of increasing the number of unknowns from six to nine. For special types of platforms, the problem can be simplified. For instance, when the $\mathbf{a}_i$s are coplanar, $\mathbf{b}_1 = \mathbf{b}_2, \mathbf{b}_3 = \mathbf{b}_4, \mathbf{b}_5 = \mathbf{b}_6$ and $(\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_5)$ form an isosceles triangle, it is possible to cast the problem into that of solving a single polynomial equation in one indeterminate with a degree equal to 16 (Merlet, 1990; Nanua et al., 1990; Innocenti and Parenti-Castelli, 1991). Husty (1996) has shown how to perform a similar transformation in the general non-planar case, but this involves complicated algebraic manipulations that are only possible in practice when the geometrical parameters defining the problem take small integer values. Many formal methods based on elimination theory suffer from the same limitation. Moreover, these methods must use numerical algorithms to find the solutions of the high-degree polynomials that they generate, and

the accuracy of the numerical results may be questionable, unless guaranteed methods such as those based on interval analysis are used. By contrast, the approach used in this section is able to isolate all real solutions of (8.14) in the most general non-planar case with realistic coefficients for the geometrical parameters.

Before running an interval solver, one must specify a prior box guaranteed to contain all solutions of interest. The naive initial domain

$$[\mathbf{x}]_0 = \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times [-\pi, \pi] \times [-\pi, \pi] \times [-\pi, \pi] \tag{8.15}$$

can be reduced by taking into account the geometry of the problem. Since

$$\mathbf{P}^{\mathrm{T}}(\psi + \pi, -\theta, \varphi + \pi) = \mathbf{P}^{\mathrm{T}}(\psi, \theta, \varphi), \tag{8.16}$$

where $\mathbf{P}^{\mathrm{T}}$ is given by (8.11)[1], the configuration of the platform for $(\psi + \pi, -\theta, \varphi + \pi)$ is the same as for (ψ, θ, φ). The prior domain can therefore be reduced to

$$[\mathbf{x}]_0 = \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times [-\pi, \pi] \times [0, \pi] \times [-\pi, \pi]. \tag{8.17}$$

To bound the domains associated with the first three components of $\mathbf{x}$, notice that

$$\mathbf{c} = \mathbf{a}(i) + (\mathbf{b}(i) - \mathbf{a}(i)) + (\mathbf{c} - \mathbf{b}(i)). \tag{8.18}$$

In $\mathcal{R}_0$, this equation becomes

$$\begin{pmatrix} c_{x0} \\ c_{y0} \\ c_{z0} \end{pmatrix} = \begin{pmatrix} a_{x0}(i) \\ a_{y0}(i) \\ a_{z0}(i) \end{pmatrix} + (\mathbf{b}(i) - \mathbf{a}(i))_{\mathcal{R}_0} + (\mathbf{c} - \mathbf{b}(i))_{\mathcal{R}_0}. \tag{8.19}$$

Now, since y_i is the distance between $\mathbf{a}(i)$ and $\mathbf{b}(i)$, each component of $(\mathbf{b}(i) - \mathbf{a}(i))_{\mathcal{R}_0}$ belongs to the interval $[-y_i, y_i]$ and since the distance $d(\mathbf{c}, \mathbf{b}(i))$ between $\mathbf{c}$ and $\mathbf{b}(i)$ is known, each component of $(\mathbf{c} - \mathbf{b}(i))_{\mathcal{R}_0}$ belongs to the interval $[-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))]$. Thus, for any i,

$$\begin{pmatrix} c_{x0} \\ c_{y0} \\ c_{z0} \end{pmatrix} \in \begin{pmatrix} a_{x0}(i) \\ a_{y0}(i) \\ a_{z0}(i) \end{pmatrix} + \begin{pmatrix} [-y_i, y_i] \\ [-y_i, y_i] \\ [-y_i, y_i] \end{pmatrix} + \begin{pmatrix} [-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))] \\ [-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))] \\ [-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))] \end{pmatrix}, \tag{8.20}$$

and $\mathbf{c}$ expressed in $\mathcal{R}_0$ belongs to the box

[1] To check (8.16), it suffices to check it for all entries of the matrices. For instance, for the entry associated with the first row and the first column, we have $\cos(\psi + \pi)\cos(\varphi + \pi) - \sin(\psi + \pi)\cos(-\theta)\sin(\varphi + \pi) = (-\cos\psi)(-\cos\varphi) - (-\sin\psi)\cos\theta(-\sin\varphi) = \cos\psi\cos\varphi - \sin\psi\cos\theta\sin\varphi$.

$$[\mathbf{c}]_0 = \bigcap_{i\in\{1,\ldots,6\}} \begin{pmatrix} a_{x0}(i) + [-y_i, y_i] + [-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))] \\ a_{y0}(i) + [-y_i, y_i] + [-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))] \\ a_{z0}(i) + [-y_i, y_i] + [-d(\mathbf{c}, \mathbf{b}(i)), d(\mathbf{c}, \mathbf{b}(i))] \end{pmatrix}. \tag{8.21}$$

Remark 8.1 *If the base and mobile plate are both planar (which is often the case in practice), and if* $(c_{x0}, c_{y0}, c_{z0}, \psi, \theta, \varphi)$ *is a solution, then the configuration* $(c_{x0}, c_{y0}, -c_{z0}, \psi + \pi, \theta, \varphi + \pi)$*, which is symmetrical with respect to the base, is also a solution. In such a case, the search domain can be limited to positive* c_{z0}*s. The prior knowledge that the mobile plate is above the base would lead to the same decision.* ■

Table 8.2. Data of Example 8.1

i	$a_{x0}(i)$	$a_{y0}(i)$	$a_{z0}(i)$	$b_{x4}(i)$	$b_{y4}(i)$	$b_{z4}(i)$	$(y_i)^2$
1	−3	2	0	−1	1	0	22
2	3	2	0	1	1	0	31
3	4	0	0	2	−1	0	39
4	1	−3	0	1	−2	0	29
5	−1	−3	0	−1	−2	0	22
6	−4	1	0	−2	−1	0	22

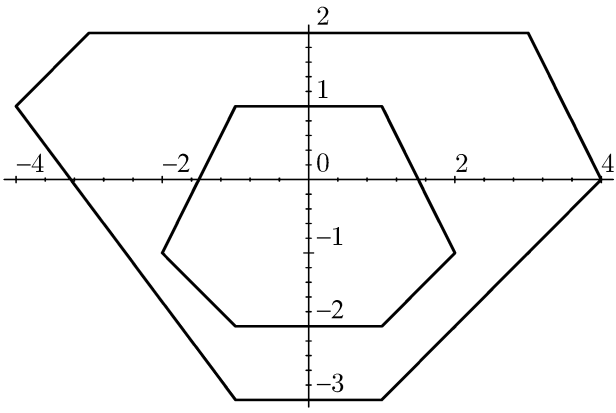


Fig. 8.3. Shapes of the base and mobile plate (Example 8.1)

Example 8.1 (planar case) *The base and mobile plate are planar, as described in Table 8.2. Their shapes are represented on Figure 8.3. Based on (8.17), (8.21) and Remark 8.1, the search box is taken as*

$$[-7.93, 3.99] \times [-4.99, 8.25] \times [0, 6.25] \times [-\pi, \pi] \times [0, \pi] \times [-\pi, \pi].$$

For $\varepsilon_{\mathrm{x}} = \varepsilon_{\mathrm{f}} = 10^{-5}$*, after* 286345 *iterations performed in* 92 *minutes on a* Pentium 90*, Hansen's algorithm for solving equations (Hansen, 1992b) generates four boxes, each of which satisfies the uniqueness condition of page 122. This algorithm is similar to the generic solver* SiviaX *of Section 5.2, page 104, with the contractors described in Section 5.5.2, page 121. Approximations of the four solutions are given in Table 8.3. Figure 8.4 presents the associated configurations.* ■

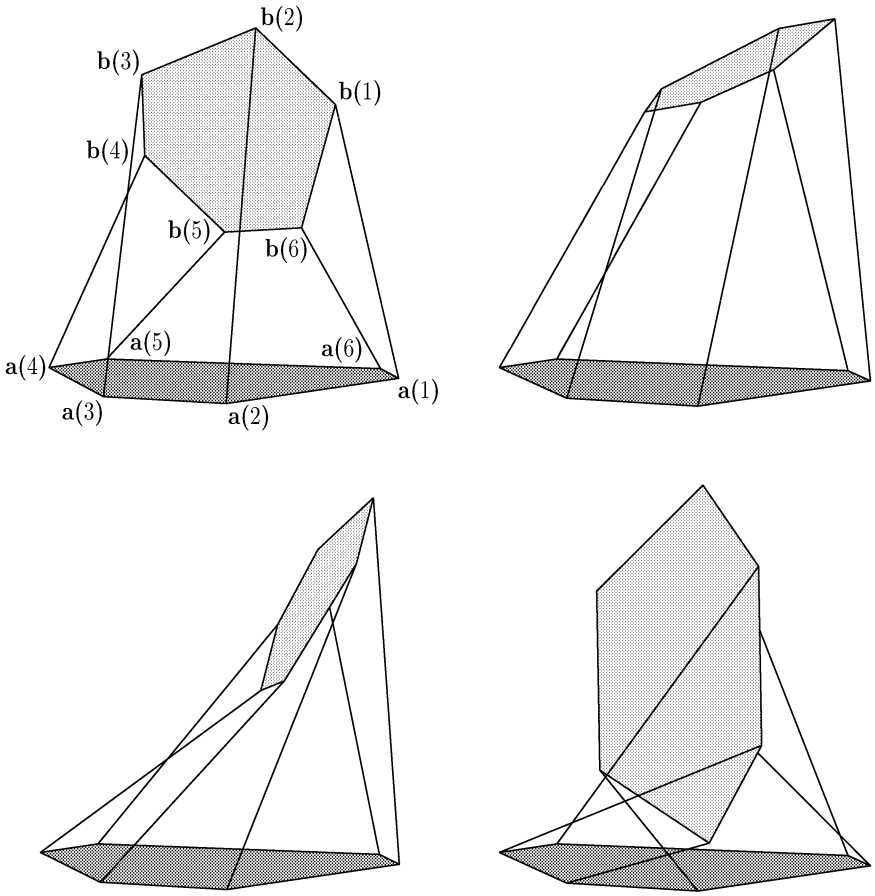


Fig. 8.4. The four configurations associated with Example 8.1

Table 8.3. Configurations obtained for Example 8.1

	c_{x0}	c_{y0}	z_{y0}	ψ	θ	φ
1	0.823	1.359	4.600	−1.482	1.856	−1.726
2	−1.014	1.001	4.976	0.690	0.966	−0.531
3	−2.014	1.222	4.423	0.027	0.519	−0.015
4	−1.772	−1.574	1.921	−0.871	1.297	0.706

Table 8.4. Data of Example 8.2

i	$a_{x0}(i)$	$a_{y0}(i)$	$a_{z0}(i)$	$b_{x4}(i)$	$b_{y4}(i)$	$b_{z4}(i)$	$(y_i)^2$
1	−9.70	9.1	1.0	−3.000	7.300	1.0	426.76
2	9.70	9.1	−1.0	3.000	−7.300	−1.0	576.27
3	12.76	3.9	1.0	7.822	−1.052	1.0	365.86
4	3.00	−13.0	−1.0	4.822	−6.248	−1.0	377.70
5	−3.00	−13.0	1.0	−4.822	−4.822	1.0	381.53
6	−12.76	3.9	−1.0	−7.822	−7.822	−1.0	276.30

Example 8.2 (non-planar case) *The base and mobile plate, as described in Table 8.4, are no longer planar. Again based on (8.17) and (8.21), the prior box is taken as*

$$[-30.37, 18.92] \times [-19.52, 33.1] \times [-25, 23] \times [-\pi, \pi] \times [0, \pi] \times [-\pi, \pi].$$

For $\varepsilon_{\mathrm{x}} = \varepsilon_{\mathrm{f}} = 10^{-5}$*, after* 481800 *iterations performed in* 153 *minutes on a* Pentium 90*, Hansen's algorithm for solving equations finds ten boxes (see Table 8.5). For each of them, the uniqueness condition is satisfied. The associated configurations are depicted on Figure 8.5. For some of them the mobile plate intersects the base, which is not realistic but was not forbidden in the problem as it had been defined.* ■

8.3 Path Planning

This section presents a recent approach to finding a collision-free path for an object in a environment cluttered with known obstacles (Jaulin and Godon, 1999; Jaulin, 2001a). This problem of *path planning in a known environment* has received considerable attention (Nilsson, 1969; Lozano-Pérez, 1981; O'Dunlaing and Yap, 1982; Rimon and Koditschek, 1992). Most

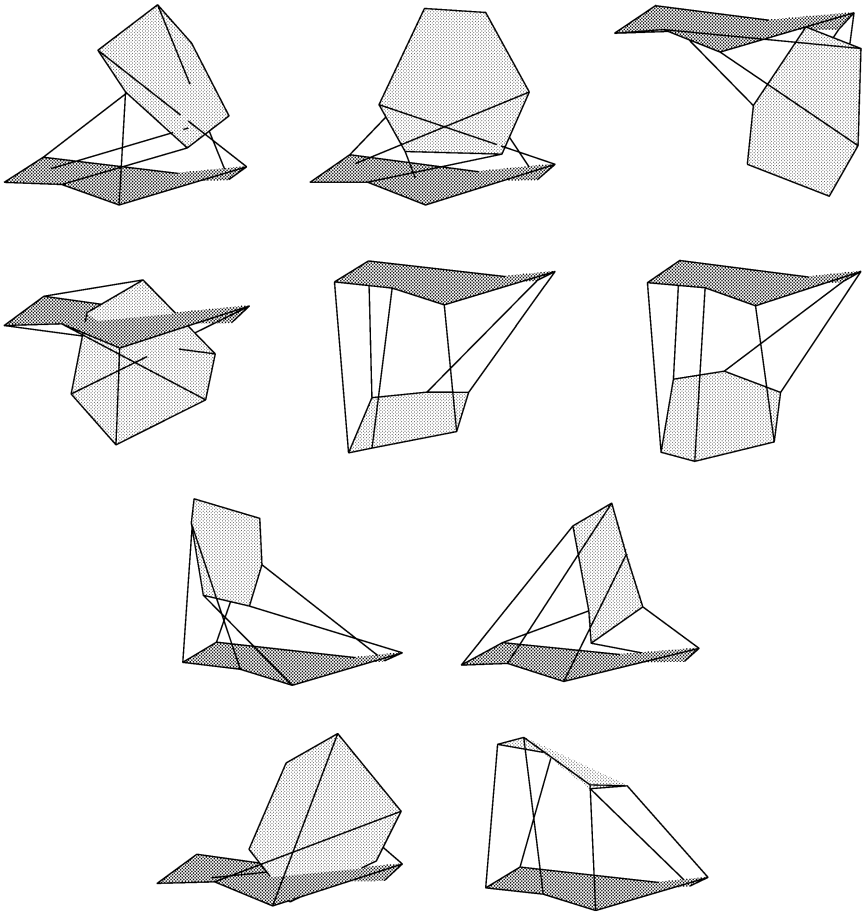


Fig. 8.5. The ten configurations associated with Example 8.2

of the methods available in the literature are based on the concept of *configuration space* (or C-space) (Lozano-Pérez and Wesley, 1979). Each coordinate in C-space represents a degree of freedom of the object. The number of independent parameters needed to specify the configuration of this object corresponds to the dimension of C-space. The initial configuration and desired final configuration of the object become two points **a** and **b** in C-space. Examples of such objects are industrial robots with n degrees of freedom. Their configuration can be characterized by n real numbers, which are the coordinates of an n-dimensional vector in C-space (Lozano-Pérez, 1983).

The *feasible configuration space* $\mathbb{S}$ is a subset of C-space that only contains configuration vectors for which the object does not collide with obstacles. Path planning amounts to finding a path belonging to $\mathbb{S}$ from the

Table 8.5. Configurations obtained for Example 8.2

	c_{x0}	c_{y0}	z_{y0}	ψ	θ	φ
1	4.945	−6.707	9.757	2.076	2.159	2.801
2	4.458	−4.606	7.666	1.126	1.973	2.051
3	−10.406	−6.218	−7.269	0.692	1.860	0.225
4	−2.904	−1.346	−3.538	1.124	1.107	1.756
5	−4.835	5.507	−16.756	−2.790	0.503	3.040
6	−6.980	6.457	−14.916	−2.256	0.740	2.664
7	1.561	6.709	15.219	−0.928	1.227	0.566
8	−11.524	−0.882	10.701	0.172	1.582	−0.504
9	−7.674	−3.113	5.658	−1.612	1.953	−1.828
10	−5.000	5.000	17.000	0.000	0.524	0.000

initial point **a** to the desired point **b**. A number of approaches to solving this problem are based on the use of potential functions (Khatib, 1986). The obstacles to be avoided are then surrounded by a repulsive potential, and the desired final configuration is surrounded by an attractive potential. Driven by the force generated by these potentials, the object is expected to reach the desired configuration without colliding with obstacles (provided that it does not stop at any local minimum). Approaches based on subdivision of C-space have also been considered (Brooks and Lozano-Pérez, 1985; Reboulet, 1988; Pruski, 1996). These approaches partition C-space into three sets of non-overlapping boxes, namely those that have been proved to be inside $\mathbb{S}$, those that have been proved to be outside $\mathbb{S}$, and those for which nothing has been proved. Although this sounds familiar, the methods used so far in the path-planning literature to decide whether a box is inside or outside $\mathbb{S}$ are not based on interval analysis and meet difficulties with orientation parameters. Interval analysis has already been used for parametric path planning in Jaulin and Walter (1996) and Piazzi and Visioli (1998), but this required a parametric model for the path to be available, *i.e.*, the path had to belong to a family parametrized by a vector **p** to be tuned, and the dimension of **p** had to be small. In these papers, the model chosen for the path was a cubic polynomial. By contrast, the approach to be presented now does not require any parametric model of the path.

Section 8.3.1 recalls the basic notions used to build a graph associated with the path-planning problem. In Section 8.3.2, two algorithms for finding a feasible path from **a** to **b** are described. The first one characterizes $\mathbb{S}$ with subpavings before looking for a feasible path. Except for the fact that the tests used to decide on the feasibility of a box are based on interval analysis, this algorithm is rather classical (Brooks and Lozano-Pérez, 1985; Reboulet, 1988). The second algorithm, much more efficient, only investigates regions of C-space that may lead to an interesting path. As an application, Section 8.3.3

considers the planning of the displacement of a non-convex polygonal object in a two-dimensional space cluttered with obstacles represented by line segments.

8.3.1 Graph discretization of configuration space

A guaranteed characterization of the feasible configuration space $\mathbb{S}$ can be obtained using a subdivision algorithm such as SIVIA, presented in Chapter 3. A graph associated with this characterization can then be built. The whole procedure, which we call *graph discretization*, will be used in Section 8.3.2 for path planning. The basic notions needed to understand the principles of graph discretization will now be presented.

Recall that a *paving* of a box $[\mathbf{p}_0]$ is a set of non-overlapping boxes, the union of which is equal to $[\mathbf{p}_0]$. This paving is denoted by $\mathbb{P}$ when it is considered as a set and by $\mathcal{P}$ when it is considered as a list of boxes (see Remark 3.1, page 51). Figure 8.6 describes a paving $\mathcal{P} = \{[\mathbf{p}_1], [\mathbf{p}_2], \ldots, [\mathbf{p}_9]\}$ of the box $[\mathbf{p}_0] = [-2, 10] \times [-2, 6]$.

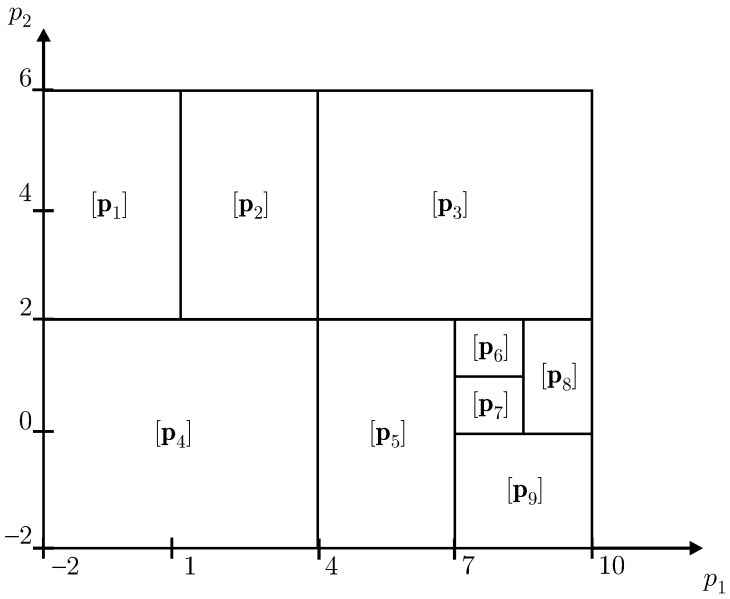


Fig. 8.6. A paving, denoted by $\mathbb{P}$ as a set and by $\mathcal{P}$ as a list of boxes

Two boxes of $\mathbb{IR}^n$ listed in $\mathcal{P}$ are *neighbours* if they share, at least partly, an $(n-1)$-dimensional face. For instance, $[\mathbf{p}_1]$ and $[\mathbf{p}_4]$ are neighbours in the paving of Figure 8.6, but $[\mathbf{p}_2]$ and $[\mathbf{p}_5]$ are not. The subpaving $\mathcal{P}_1$ of a paving $\mathcal{P}$ that contains all the boxes of $\mathcal{P}$ satisfying a given condition is denoted by

$$\mathcal{P}_1 = \text{ subpaving}\,(\mathcal{P}, \text{condition})\,. \tag{8.22}$$

Consider, for example, the test $t(\mathbf{p}) \triangleq (p_1 = 5)$, where p_1 is the first component of $\mathbf{p}$, again in connection with the paving of Figure 8.6. If $[t]\,([\mathbf{p}])$ is the minimal inclusion test for $t(\mathbf{p})$, since $[t]\,([\mathbf{p}_3]) = [t]\,([\mathbf{p}_5]) = [0,1]$, then

$$\text{subpaving}\,(\mathcal{P}, [t]\,([\mathbf{p}]) = 0) = \{[\mathbf{p}_1], [\mathbf{p}_2], [\mathbf{p}_4], [\mathbf{p}_6], [\mathbf{p}_7], [\mathbf{p}_8], [\mathbf{p}_9]\}\,. \tag{8.23}$$

Some basic notions of graph theory are also needed (Deo, 1974). A *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a non-empty set $\mathcal{V}$ of vertices, and of a set $\mathcal{E}$ of unordered pairs of vertices of $\mathcal{V}$ called *edges*. If v_a and v_b are two vertices of the graph, the edge associated with the pair (v_a, v_b) is denoted by $v_a v_b$. A *walk* in $\mathcal{G}$ is a sequence of vertices v_i $(i = 1, \ldots, k)$ such that for any $i \in \{1, \ldots, k-1\}$, the edge $v_i v_{i+1}$ belongs to $\mathcal{E}$. The walk is a *path* if $v_i \neq v_j$ for $i \neq j$. The walk is a *cycle* if $v_k = v_1$. A graph is *connected* if there is a path between any two vertices. Two distinct vertices v_i and v_j of $\mathcal{G}$ are *neighbours* if $\mathcal{E}$ contains the edge $v_i v_j$. A *subgraph* of $\mathcal{G}$ is a graph whose vertices and edges belong to $\mathcal{G}$.

Any paving $\mathcal{P}$ of a box $[\mathbf{p}_0]$ can be represented by a graph $\mathcal{G}$. Each element $[\mathbf{p}_i]$ of $\mathcal{P}$ is associated with a vertex v_i of $\mathcal{G}$. If two boxes $[\mathbf{p}_i]$ and $[\mathbf{p}_j]$ of $\mathcal{P}$ are neighbours, then $\mathcal{G}$ contains the edge $v_i v_j$. For instance, the graph $\mathcal{G}$ associated with the paving of Figure 8.6 is given in Figure 8.7a. Subpavings can also be represented by graphs. For instance, the graph $\mathcal{G}_1$ associated with the subpaving $\mathcal{P}_1$ of (8.23) is given in Figure 8.7b. It is a (disconnected) subgraph of $\mathcal{G}$. The graph associated with a paving (or subpaving) $\mathcal{P}$ is denoted by $\mathcal{G} = \text{graph}(\mathcal{P})$. The construction of the graph associated with a given subpaving requires fast algorithms to find the neighbours of a given box, such as the one developed by Samet (1982).

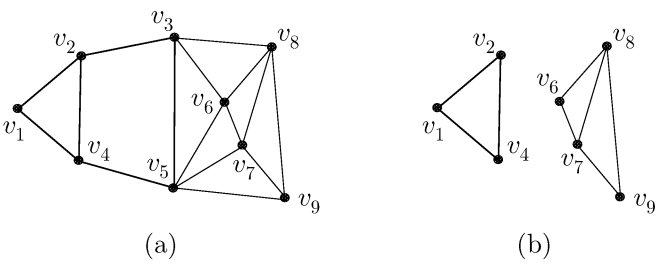


Fig. 8.7. (a) Graph $\mathcal{G}$ associated with the paving $\mathcal{P}$; (b) Graph $\mathcal{G}_1$ associated with the subpaving $\mathcal{P}_1$ of $\mathcal{P}$; $\mathcal{G}_1$ is a disconnected subgraph of $\mathcal{G}$

8.3.2 Algorithms for finding a feasible path

Consider a compact set $\mathbb{S}$ included in a box $[\mathbf{p}_0]$ and two points $\mathbf{a}$ and $\mathbf{b}$ of $\mathbb{S}$. Assume that a thin inclusion test $[t]$ is available to decide whether a box is inside or outside $\mathbb{S}$. A *motion* from the initial point $\mathbf{a}$ to the desired final point $\mathbf{b}$ is a one-to-one continuous function $\mathbf{m} : [0,1] \to \mathbb{R}^n; \tau \mapsto \mathbf{m}(\tau)$, such that $\mathbf{m}(0) = \mathbf{a}$ and $\mathbf{m}(1) = \mathbf{b}$. The associated path is the set

$$\mathbb{L} = \{\mathbf{m}(\tau) \mid \tau \in [0,1]\}. \tag{8.24}$$

The path $\mathbb{L}$ is *feasible* if $\mathbb{L} \subset \mathbb{S}$. In this section, two algorithms FeasiblePath1 and FeasiblePath2 searching for such paths are proposed. When they succeed, both return a *box path*, *i.e.*, a list of adjacent boxes $\{[\mathbf{p}_a], [\mathbf{p}_1], \ldots, [\mathbf{p}_{\ell-1}], [\mathbf{p}_b]\}$, such that $\mathbf{a} \in [\mathbf{p}_a]$ and $\mathbf{b} \in [\mathbf{p}_b]$, and that all these boxes are inside $\mathbb{S}$. It is then still necessary to find a feasible point path $\mathbb{L}$ from $\mathbf{a}$ to $\mathbf{b}$. In general, the choice of this final point path should be based on domain-specific considerations such as kinematic or dynamic characteristics, and not on purely geometric criteria (Laumond, 1986). For instance, a desirable property of the final path is smoothness. Here, for the sake of simplicity, a broken line from $\mathbf{a}$ to $\mathbf{b}$ lying inside the box path will be considered sufficient.

Table 8.6. Algorithm for finding a shortest path in a graph

Algorithm Dijkstra(in: $\mathcal{G}, v_a, v_b$; out: $\mathcal{L}$)	
1	for each vertex $v \in \mathcal{G}$, $d(v) := \infty$;
2	$d(v_a) := 0$; $d_{\min} := 0$;
3	repeat
4	if $\mathcal{G}(d_{\min}) = \varnothing$ then $\mathcal{L} := \varnothing$; return;
5	$d_{\min} := d_{\min} + 1$;
6	for each vertex $v \in \mathcal{G}(d_{\min} - 1)$,
7	for each neighbour w of v in $\mathcal{G}$ with $d(w) = \infty$, $d(w) := d_{\min}$;
8	until $d(v_b) \neq \infty$;
9	$\ell := d(v_b)$; $v_\ell := v_b$;
10	for $i := \ell - 1$ down to 0, select a neighbour v_i of v_{i+1} such that $d(v_i) = i$;
11	$\mathcal{L} := \{v_a, v_1, v_2, \ldots, v_{\ell-1}, v_b\}$.

Among the algorithms that have been proposed for finding the shortest path between two specified vertices v_a and v_b in a graph $\mathcal{G}$, one of the most efficient is due to Dijkstra (1959). It will be called by the two algorithms FeasiblePath1 and FeasiblePath2. Although it has been initially derived for weighted *digraphs* (*i.e.*, graphs with directed edges), we shall use a simplified version for non-directed graphs, presented in Table 8.6. With each vertex v of $\mathcal{G}$ is associated an integer $d(v)$ representing the minimum number of edges

in a path from v_a to v. $\mathcal{G}(i)$, $i \in \mathbb{N}$, denotes the set of all vertices of $\mathcal{G}$ such that $d(v) = i$. If the algorithm DIJKSTRA returns an empty list $\mathcal{L}$, then v_a and v_b are not in the same connected component of $\mathcal{G}$. Otherwise, it returns one of the shortest paths from v_a to v_b, in terms of the number of vertices crossed.

Running DIJKSTRA$(\mathcal{G}, v_1, v_6)$ on the graph of Figure 8.7a, one gets $d(v_1) = 0$, $d(v_2) = d(v_4) = 1$, $d(v_3) = d(v_5) = 2$, $d(v_6) = d(v_7) = d(v_8) = d(v_9) = 3$. DIJKSTRA returns either the path $\{v_1, v_2, v_3, v_6\}$ or the path $\{v_1, v_4, v_5, v_6\}$.

Table 8.7. Basic algorithm for finding a feasible path

Algorithm FEASIBLEPATH1(in: $t(.), \mathbf{a}, \mathbf{b}, [\mathbf{p}_0], \varepsilon$; out: $\underline{\mathcal{L}}$, message)

```
1  if [t](a) ≠ 1 or [t](b) ≠ 1 then
2     L̲ := ∅; message := "error: a and b should be feasible"; return;
3  if a ∉ [p0] or b ∉ [p0] then
4     L̲ := ∅; message := "error: a and b should belong to [p0]"; return;
5  Q := {[p0]}; ΔP := ∅; P̲ := ∅;
6  while Q ≠ ∅;
7     pop a box out of Q into [p];
8     if [t]([p]) = 1 then P̲ := P̲ ∪ {[p]};
9     if [t]([p]) = [0,1] and w([p]) ⩽ ε then ΔP := ΔP ∪ {[p]};
10    if [t]([p]) = [0,1] and w([p]) > ε then
11       bisect([p]) and put the two resulting boxes at the end of Q;
12 end while;
13 P̄ = P̲ ∪ ΔP; Ḡ = graph(P̄); G̲ = graph(P̲);
14 v_a := vertex([p_a]), where [p_a] ∈ P̄ and a ∈ [p_a];
15 v_b := vertex([p_b]), where [p_b] ∈ P̄ and b ∈ [p_b];
16 L̄ := DIJKSTRA(Ḡ, v_a, v_b);
17 if L̄ = ∅ then
18    L̲ := ∅; message := "no path"; return;
19 if v_a ∉ G̲ or v_b ∉ G̲ then
20    L̲ := ∅; message := "no path"; return;
21 L̲ := DIJKSTRA(G̲, v_a, v_b);
22 if L̲ ≠ ∅ then
23    message := "path found";
24 else
25    message := "failure".
```

The first part of the algorithm FEASIBLEPATH1 given in Table 8.7, is the procedure SIVIA (see Chapter 3), which is used to build a paving $\mathbb{P}$ and two subpavings $\underline{\mathbb{P}}$ and $\overline{\mathbb{P}}$ of $\mathbb{P}$ satisfying

$$\underline{\mathbb{P}} \subset \mathbb{S} \subset \overline{\mathbb{P}}. \tag{8.25}$$

A stack of boxes $\mathcal{Q}$ serves to store all the boxes still to be studied. The graphs $\underline{\mathcal{G}}$ and $\overline{\mathcal{G}}$ associated with $\underline{\mathcal{P}}$ and $\overline{\mathcal{P}}$ are then built, and two boxes $[\mathbf{p}_a]$ and $[\mathbf{p}_b]$ of $\overline{\mathcal{P}}$ are selected, such that $\mathbf{a} \in [\mathbf{p}_a]$ and $\mathbf{b} \in [\mathbf{p}_b]$. Several acceptable candidates $[\mathbf{p}_a]$ or $[\mathbf{p}_b]$ may exist if $\mathbf{a}$ or $\mathbf{b}$ is on the boundary of a box of $\overline{\mathcal{P}}$. In such a case, the algorithm selects any of them. Let v_a and v_b be the two vertices of $\overline{\mathcal{G}}$ associated with $[\mathbf{p}_a]$ and $[\mathbf{p}_b]$. FeasiblePath1 calls the procedure Dijkstra to get a path $\overline{\mathcal{L}}$ of $\overline{\mathcal{G}}$ from v_a to v_b. If no such path is found, then $\mathbf{a}$ and $\mathbf{b}$ have been proved to belong to disconnected components[2] of $\mathbb{S}$, and FeasiblePath1 reports that there can be no path. If a non-empty $\overline{\mathcal{L}}$ is found, then Dijkstra is run again to find a path $\underline{\mathcal{L}}$ of $\underline{\mathcal{G}}$ connecting v_a to v_b. If such a path $\underline{\mathcal{L}} = \{v_a, v_1, \ldots, v_{\ell-1}, v_b\}$ is found, then the associated box path in $\underline{\mathbb{P}}$ is included in $\mathbb{S}$ and a point path can thus be generated. If $\underline{\mathcal{L}}$ is empty, the algorithm reports failure because nothing has been proved yet about the existence or inexistence of a feasible path from $\mathbf{a}$ to $\mathbf{b}$. One may then run the algorithm again with a smaller ε.

The rationale for FeasiblePath2 presented in Table 8.8 is that the time spent running Dijkstra is very short compared to that required to build a detailed characterization of the feasible configuration space. During each iteration of FeasiblePath2, Dijkstra is used to locate the regions of configuration space that seem most promising, and the algorithm stops as soon as a feasible path has been found. Let $\mathbb{P}$ be the current paving of the search box $[\mathbf{p}_0]$. As FeasiblePath1, FeasiblePath2 first looks for a shortest path $\overline{\mathcal{L}}$ in the graph associated with an available subpaving $\overline{\mathbb{P}}$ of $\mathbb{P}$, which satisfies $\mathbb{S} \subset \overline{\mathbb{P}}$. If no such path is found, then $\mathbf{a}$ and $\mathbf{b}$ are not in the same connected component of $\mathbb{S}$ and the algorithm reports that there can be no path. If the path exists, then FeasiblePath2 tries to find the shortest path $\underline{\mathcal{L}}$ in the graph $\underline{\mathcal{G}}$ associated with a subpaving $\underline{\mathbb{P}}$ of $\mathbb{P}$, which satisfies $\underline{\mathbb{P}} \subset \mathbb{S}$. If such a path is found, it is returned. Otherwise, since the box path corresponding to $\overline{\mathcal{L}}$ may nevertheless contain a feasible path, all subboxes of this path are bisected and a new paving $\mathbb{P}$ is thus obtained.

8.3.3 Test case

The configuration space of this test case was chosen two-dimensional, so that the feasible configuration set $\mathbb{S}$ can be visualized easily, but it suffices to try to find a solution to realize that the problem is nevertheless quite complicated.

Consider a two-dimensional room that contains $\bar{j}$ segment obstacles. The extreme points of the jth obstacle are denoted by $\mathbf{a}_j$ and $\mathbf{b}_j$ for $j \in \mathcal{J} =$

[2] If $\overline{\mathcal{L}} = \varnothing$, then the vertices v_a and v_b belong to distinct connected components of the graph $\overline{\mathcal{G}}$, *i.e.*, the points $\mathbf{a}$ and $\mathbf{b}$ belong to distinct connected component of $\overline{\mathbb{P}}$. Now, since $\mathbf{a} \in \mathbb{S}, \mathbf{b} \in \mathbb{S}$ and $\mathbb{S} \subset \overline{\mathbb{P}}$, $\mathbf{a}$ and $\mathbf{b}$ belong to distinct connected component of $\mathbb{S}$. Thus, when FeasiblePath1 returns "no path", this conclusion is guaranteed (provided that outward rounding has been implemented).

Table 8.8. Improved algorithm for finding a feasible path

Algorithm FEASIBLEPATH2(in: $[t], \mathbf{a}, \mathbf{b}, [\mathbf{p}_0]$; out: $\underline{\mathcal{L}}$, message)

```
1  if [t](a) ≠ 1 or [t](b) ≠ 1 then
2     L̲ := ∅; message := "error: a and b should be feasible"; return;
3  if a ∉ [p0] or b ∉ [p0] then
4     L̲ := ∅; message := "error: a and b should belong to [p0]"; return;
5  let P be the paving consisting of the single box [p0];
6  repeat
7     P̄ := subpaving(P, 1 ∈ [t]([p])); Ḡ := graph(P̄);
8     v_a := vertex([p_a]), where [p_a] ∈ P̄ and a ∈ [p_a];
9     v_b := vertex([p_b]), where [p_b] ∈ P̄ and b ∈ [p_b];
10    L̄ := DIJKSTRA(Ḡ, v_a, v_b);
11    if L̄ = ∅ then L̲ := ∅; message := "no path"; return;
12    P̲ := subpaving(P, [t]([p]) = 1); G̲ = graph(P̲);
13    if v_a ∈ G̲ and v_b ∈ G̲ then L̲ := DIJKSTRA(G̲, v_a, v_b);
14    if L̲ ≠ ∅ then message := "path found"; return;
15    C := {[p] ∈ P̄ | vertex([p]) ∈ L̄ and [t]([p]) = [0, 1]};
16    bisect all subboxes of C, thus obtaining a new paving P;
17 forever.
```

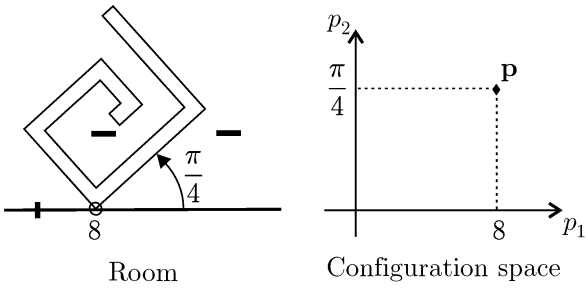


Fig. 8.8. To any given configuration of the object in the room corresponds a single point $\mathbf{p}$ in C-space

$\{1, \ldots, \bar{j}\}$. The object to be moved is a non-convex polygon with $\bar{i}$ vertices, denoted by $\mathbf{s}_i \in \mathbb{R}^2, i \in \mathcal{I} = \{1, \ldots, \bar{i}\}$. In the example to be treated, $\bar{j} = 2$ and $\bar{i} = 14$. The vertex $\mathbf{s}_1$ is constrained to stay on the horizontal axis of the room frame. The configuration of the object is thus represented by a two-dimensional vector $\mathbf{p} = (p_1, p_2)^{\mathrm{T}}$, where p_1 is the coordinate of $\mathbf{s}_1$ along the horizontal axis and p_2 is the heading angle of the object (in radians). Figure 8.8 illustrates the notion of configuration space for this test case. Figure 8.9a presents the initial configuration and Figure 8.9b the desired final configuration.

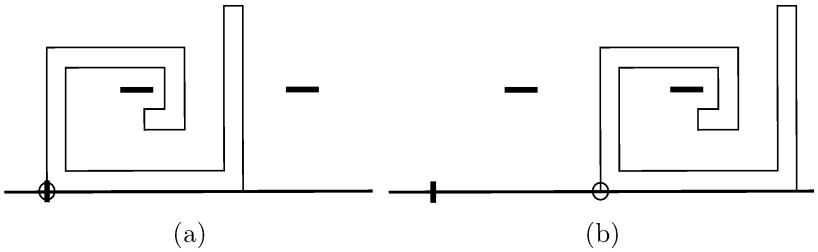


Fig. 8.9. Initial configuration (a) and desired final configuration (b)

A vector $\mathbf{p}$ associated with a given configuration is feasible if and only if none of the edges of the object intersects any of the segment obstacles and the extreme points of each segment obstacle lay outside the object. As illustrated by Figure 8.8, $\mathbf{p} = (8,\ \pi/4)^{\mathrm{T}}$ is feasible. In what follows, $\mathrm{segm}(\mathbf{a}, \mathbf{b})$ denotes the segment with end points $\mathbf{a}$ and $\mathbf{b}$, and $\mathrm{line}(\mathbf{a}, \mathbf{b})$ is the straight line passing through $\mathbf{a}$ and $\mathbf{b}$. Since $[\mathbb{A}]$ denotes the interval hull of $\mathbb{A}$ (*i.e.*, the smallest box containing $\mathbb{A}$), $[\mathbf{a} \cup \mathbf{b}]$ will represent the smallest box that contains $\mathbf{a}$ and $\mathbf{b}$. If $\mathbf{s}_{\bar{i}+1}$ is taken equal to $\mathbf{s}_1$, then

$$(\mathbf{p} \in \mathbb{S}) \Leftrightarrow \begin{pmatrix} \forall i \in \mathcal{I}, \forall j \in \mathcal{J},\ \mathrm{segm}(\mathbf{s}_i, \mathbf{s}_{i+1}) \cap\ \mathrm{segm}(\mathbf{a}_j, \mathbf{b}_j) = \varnothing \\ \text{and } \mathbf{a}_j \text{ and } \mathbf{b}_j \text{ are outside the object} \end{pmatrix}. \tag{8.26}$$

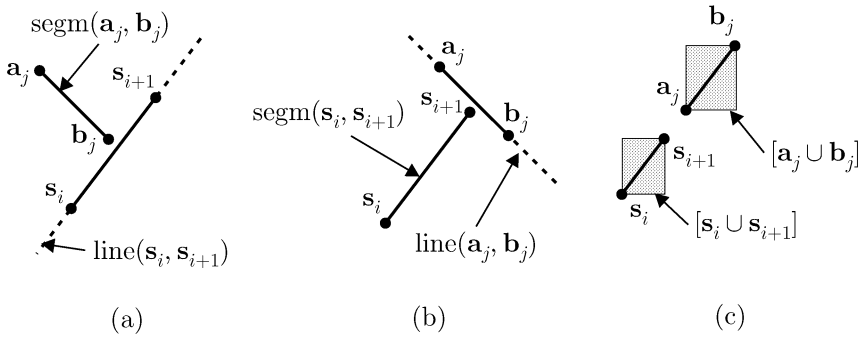


Fig. 8.10. Three configurations where the two segments have an empty intersection

Testing whether $\mathrm{segm}(\mathbf{s}_i, \mathbf{s}_{i+1})$ intersects $\mathrm{segm}(\mathbf{a}_j, \mathbf{b}_j)$ is equivalent to testing whether the three following conditions are simultaneously satisfied:

$$\begin{cases} (i) & \text{line}(\mathbf{s}_i, \mathbf{s}_{i+1}) \cap \text{segm}(\mathbf{a}_j, \mathbf{b}_j) \neq \varnothing, \\ (ii) & \text{segm}(\mathbf{s}_i, \mathbf{s}_{i+1}) \cap \text{line}(\mathbf{a}_j, \mathbf{b}_j) \neq \varnothing, \\ (iii) & [\mathbf{s}_{i+1} \cup \mathbf{s}_i] \cap [\mathbf{a}_j \cup \mathbf{b}_j] \neq \varnothing. \end{cases} \tag{8.27}$$

Condition (i) is false in Figure 8.10a, (ii) is false in Figure 8.10b and (iii) is false in Figure 8.10c. In each of these cases, although the other two conditions hold true, $\text{segm}(\mathbf{s}_i, \mathbf{s}_{i+1}) \cap \text{segm}(\mathbf{a}_j, \mathbf{b}_j) = \varnothing$.

Table 8.9. Algorithm for testing a configuration vector $\mathbf{p}$ for feasibility

Algorithm t(in: $\mathbf{p}$; out: t)	
1	for $j = 1$ to $\overline{j}$,
2	$\tilde{x}_a = (x_a(j) - p_1)\cos p_2 + y_a(j)\sin p_2$;
3	$\tilde{y}_a = -(x_a(j) - p_1)\sin p_2 + y_a(j)\cos p_2$;
4	$\tilde{x}_b = (x_b(j) - p_1)\cos p_2 + y_a(j)\sin p_2$;
5	$\tilde{y}_b = -(x_a(j) - p_1)\sin p_2 + y_a(j)\cos p_2$;
6	$\tilde{\mathbf{a}} = (\tilde{x}_a, \tilde{y}_a)^{\mathrm{T}}$; $\tilde{\mathbf{b}} = (\tilde{x}_b, \tilde{y}_b)^{\mathrm{T}}$;
7	if $\tilde{\mathbf{a}}$ is inside the object then $t := 0$; return;
8	if $\tilde{\mathbf{b}}$ is inside the object then $t := 0$; return;
9	for $i = 1$ to $\overline{i}$,
10	$d_1 = \det\left(\mathbf{s}_i - \tilde{\mathbf{b}}, \mathbf{s}_i - \tilde{\mathbf{a}}\right) * \det\left(\mathbf{s}_{i+1} - \tilde{\mathbf{b}}, \mathbf{s}_{i+1} - \tilde{\mathbf{a}}\right)$;
11	$d_2 = \det(\mathbf{s}_{i+1} - \mathbf{s}_i, \mathbf{s}_i - \tilde{\mathbf{a}}) * \det\left(\mathbf{s}_{i+1} - \mathbf{s}_i, \mathbf{s}_i - \tilde{\mathbf{b}}\right)$;
12	if $(d_1 \leqslant 0)$ and $(d_2 \leqslant 0)$ and $\left([\mathbf{s}_{i+1} \cup \mathbf{s}_i] \cap \left[\tilde{\mathbf{a}} \cup \tilde{\mathbf{b}}\right] \neq \varnothing\right)$ then
13	$t := 0$; return;
14	$t := 1$.

The algorithm of Table 8.9 tests the configuration vector $\mathbf{p}$ for feasibility. For a given segment number j, Steps 2 to 6 compute $\tilde{\mathbf{a}} = (\tilde{x}_a, \tilde{y}_a)^{\mathrm{T}}$ and $\tilde{\mathbf{b}} = (\tilde{x}_b, \tilde{y}_b)^{\mathrm{T}}$, the coordinates of the extreme points of $\text{segm}(\mathbf{a}_j, \mathbf{b}_j)$ in the object frame. To prove that $\tilde{\mathbf{a}}$ is inside the object as required at Step 7, it suffices to check that

$$\sum_{i=1}^{\overline{i}} \arg(\mathbf{s}_i - \tilde{\mathbf{a}}, \mathbf{s}_{i+1} - \tilde{\mathbf{a}}) \neq 0. \tag{8.28}$$

This sum is equal to zero if $\tilde{\mathbf{a}}$ is outside the polygon (Figure 8.11a) and to 2π if $\tilde{\mathbf{a}}$ is inside (Figure 8.11b). The same type of condition is used for $\tilde{\mathbf{b}}$ at Step 8. If $d_1 \leqslant 0$, $d_2 \leqslant 0$ and $[\mathbf{s}_{i+1} \cup \mathbf{s}_i] \cap [\tilde{\mathbf{a}} \cup \tilde{\mathbf{b}}] \neq \varnothing$, then (8.27) indicates that the ith edge of the object intersects the jth segment obstacle; $\mathbf{p}$ is thus unfeasible and the algorithm returns zero at Step 13. An inclusion test $[t]\,([\mathbf{p}])$ for $t(\mathbf{p})$ is given by the algorithm of Table 8.10. To evaluate $[\tilde{x}_a]$, $[\tilde{y}_a]$, $[\tilde{x}_b]$,

$[\tilde{y}_b]$, $[\tilde{x}_b]$, $[d_1]$, $[d_2]$, the centred form has been used with respect to p_1 and p_2.

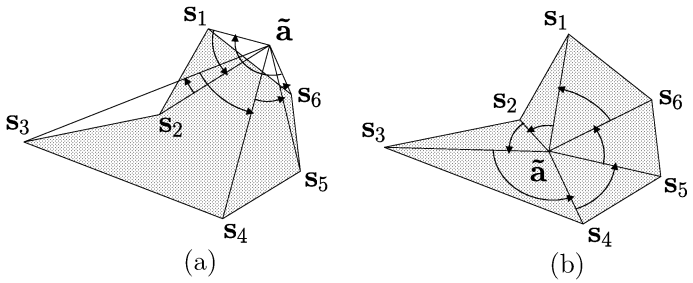


Fig. 8.11. Test to check whether $\tilde{\mathbf{a}}$ is inside or outside a polygon; (a): the sum of the angles is equal to zero; (b): the sum is equal to 2π

Table 8.10. Algorithm for testing a configuration box $[\mathbf{p}]$ for feasibility

Algorithm $[t]$ (in: $[\mathbf{p}]$; out: $[t]$)	
1	$[t] := 1$;
2	for $j := 1$ to $\overline{j}$,
3	$[\tilde{x}_a] := (x_a(j) - p_1)\cos[p_2] + y_a(j)\sin[p_2]$;
4	$[\tilde{y}_a] := -(x_a(j) - p_1)\sin[p_2] + y_a(j)\cos[p_2]$;
5	$[\tilde{x}_b] := (x_b(j) - p_1)\cos[p_2] + y_a(j)\sin[p_2]$;
6	$[\tilde{y}_b] := -(x_a(j) - p_1)\sin[p_2] + y_a(j)\cos[p_2]$;
7	$[\tilde{\mathbf{a}}] := ([\tilde{x}_a], [\tilde{y}_a])^{\mathrm{T}}$; $[\tilde{\mathbf{b}}] = ([\tilde{x}_b], [\tilde{y}_b])^{\mathrm{T}}$;
8	if $[\tilde{\mathbf{a}}]$ is inside the object then $[t] := 0$; return;
9	if $[\tilde{\mathbf{b}}]$ is inside the object then $[t] := 0$; return;
10	for $i := 1$ to $\overline{i}$,
11	$[d_1] := \det\left(\mathbf{s}_i - [\tilde{\mathbf{b}}], \mathbf{s}_i - [\tilde{\mathbf{a}}]\right) * \det\left(\mathbf{s}_{i+1} - [\tilde{\mathbf{b}}], \mathbf{s}_{i+1} - [\tilde{\mathbf{a}}]\right)$;
12	$[d_2] := \det(\mathbf{s}_{i+1} - \mathbf{s}_i, \mathbf{s}_i - [\tilde{\mathbf{a}}]) * \det\left(\mathbf{s}_{i+1} - \mathbf{s}_i, \mathbf{s}_i - [\tilde{\mathbf{b}}]\right)$;
13	if $([d_1] < 0$ and $[d_2] < 0)$ then $[t] := 0$; return;
14	if $(0 \in [d_1]$ or $0 \in [d_2])$ and $\left[[\tilde{\mathbf{a}}] \cup [\tilde{\mathbf{b}}]\right] \cap [\mathbf{s}_{i+1} \cup \mathbf{s}_i] \neq \varnothing$
15	then $[t] := [0, 1]$.

In less than 10 minutes on a PENTIUM 133 and for $\varepsilon = 0.1$, FEASIBLEPATH1 generates the paving presented in Figure 8.12 and the associated graph. The grey boxes are proved feasible, and the black boxes unfeasible. Nothing is known about the white boxes. In less than 0.1 s, FEASIBLEPATH1

Fig. 8.12. Paving and path generated by FeasiblePath1; the frame corresponds to the search box $[\mathbf{p}_0] = [-28, 57] \times [-1.4, 2.7]$; $\mathbf{z}$ corresponds to a dead-end

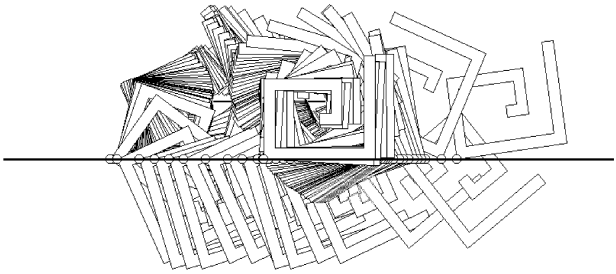


Fig. 8.13. Display of the motion; the two obstacles are still visible

then finds the shortest path in the graph. The corresponding motion is displayed in Figure 8.13. As expected, the two obstacle segments are still visible on the figure. For $\varepsilon = 0.2$, FeasiblePath1 is unable to find a feasible path and reports failure.

In less than 1 minute, FeasiblePath2 finds the path shown in Figure 8.14. The grey boxes are proved to be inside $\mathbb{S}$, the black boxes are outside $\mathbb{S}$ and nothing is known about the white boxes. FeasiblePath2 expends efforts on bisecting and analyzing zones of C-space only when needed, which explains why it is much more efficient than FeasiblePath1.

Fig. 8.14. Paving and path generated by FeasiblePath2. The frame corresponds to the search box $[\mathbf{p}_0] = [-28, 57] \times [-1.4, 2.7]$

Remark 8.2 *The configuration presented in Figure 8.15, usually reached when trying to solve the problem by hand, is a dead-end. It corresponds to* $\mathbf{z}$ *in Figure 8.12.* ■

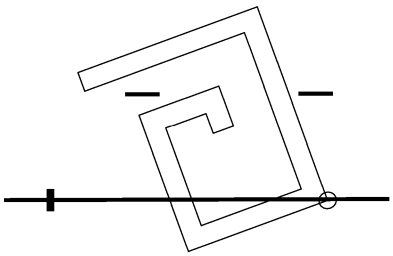


Fig. 8.15. A dead-end

Remark 8.3 *The frame box* $[\mathbf{p}_0] = [-28, 57] \times [-1.4, 2.7]$ *has been chosen small enough to make the small boxes visible, and large enough to include the whole path. The same problem has also been treated for* $[\mathbf{p}_0] = [-100, 100] \times [-10, 10]$. *The computing time with* FeasiblePath1 *is about three times larger than for the former* $[\mathbf{p}_0]$*, whereas the computing time with* FeasiblePath2 *is not changed significantly.* ■

8.4 Localization and Tracking of a Mobile Robot

The autonomous localization of a vehicle in a partially known environment is a key problem of mobile robotics. A variety of sensors may be used, each of them providing uncertain measurements that must be combined, and this localization is archetypal of problems of data fusion (Crowley, 1989; Castellanos et al., 1999). The difficulty is increased by the fact that the maps of the environment provided to the robot are often inaccurate or outdated.

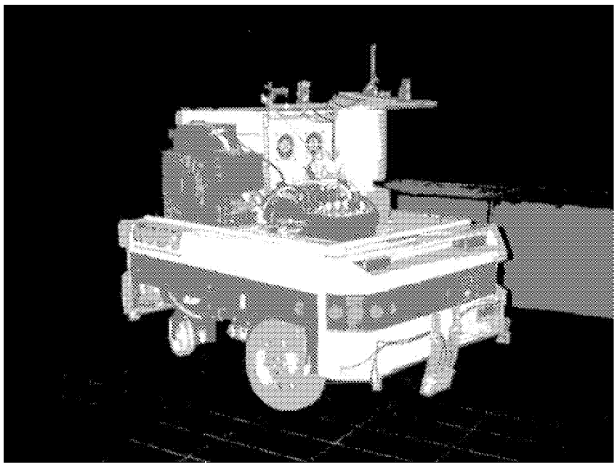


Fig. 8.16. *Robuter* mobile robot by *Robosoft*

To localize itself dynamically, a robot such as that of Figure 8.16 must first estimate its initial position and orientation, and this initialization phase is more complex than tracking proper. Indeed, once a reasonably accurate initialization has been performed, it is often possible to use a well established local tracking technique such as extended Kalman filtering (Leonard and Durrant-Whyte, 1991) or its bounded-error counterpart (Hanebeck and Schmidt, 1996; Meizel et al., 1996). It will be assumed here that initially the robot is only known to be located inside the region described by the map. In this context, initialization pertains to pattern analysis (Drumheller, 1987), and has been handled using data interpretation trees (Grimson and Lozano-Pérez, 1987; Halbwachs and Meizel, 1997), and more recently global maximum-likelihood estimation (Olson, 2000).

The method presented in this chapter makes it possible to solve localization and tracking problems even when the result of the initialization phase is ambiguous or very imprecise. It is based on the hypothesis that the measurement errors and state perturbations belong to known compact sets, but

the estimation method will be made robust to data that would not satisfy this assumption.

The simpler static case where the robot is immobile will be considered first. Robot localization is formulated as a bounded-error parameter estimation problem in Section 8.4.1. Section 8.4.2 describes a model of the measurements provided by ultrasonic sensors (or sonars). The construction of an inclusion function for the output of this model is discussed in Section 8.4.3. This inclusion function allows the techniques presented in Chapter 6 to be used. Section 8.4.4 explains how outliers are dealt with and an example is treated in Section 8.4.5.

The case of the moving robot is considered in a second part, parameter estimation being replaced by state estimation. The tracking methodology, based on the recursive causal state estimator of Section 6.4, is described in Section 8.4.6 and illustrated in Section 8.4.7.

8.4.1 Formulation of the static localization problem

Computation will involve two frames, namely the world frame $\mathcal{W}$ and a frame $\mathcal{R}$ tied to the robot. The origin $\mathbf{c}$ of $\mathcal{R}$ is chosen as the middle of the axis between the driving wheels. Its coordinates in $\mathcal{W}$ are x_c and y_c. The angle θ between $\mathcal{R}$ and $\mathcal{W}$ corresponds to the heading angle of the robot (see Figure 8.17). Points and their coordinates will be denoted by lower-case letters in $\mathcal{W}$ and by tilded lower-case letters in $\mathcal{R}$. Thus, a point $\tilde{\mathbf{m}}$ with coordinates $(\tilde{x}, \tilde{y})$ in $\mathcal{R}$ will be denoted by $\mathbf{m}$ in $\mathcal{W}$, with

$$\mathbf{m} = \begin{pmatrix} x_\mathrm{c} \\ y_\mathrm{c} \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix}. \tag{8.29}$$

Three parameters are to be estimated, namely x_c, y_c and θ. They form the *configuration vector* $\mathbf{p} = (x_\mathrm{c}, y_\mathrm{c}, \theta)^\mathrm{T}$ (Figure 8.17).

The problem is to estimate the value of $\mathbf{p}$, assumed constant for the time being, from a *map* representing the environment of the robot and from *distance measurements* provided by a belt of n_s on-board sonars, some of which are visible on Figure 8.16. Bounds are assumed to be available for the measurement errors (they may have been obtained by separate laboratory experiments), and the resulting interval distances are stored in the interval vector $[\mathbf{d}] = ([d_1], \ldots, [d_{n_\mathrm{s}}])^\mathrm{T}$.

Provided that a model is available to compute the vector $\mathbf{d}_\mathrm{m}(\mathbf{p})$ of the distances that can be expected when the robot configuration is $\mathbf{p}$, the localization problem becomes a now familiar bounded-error parameter estimation problem, namely that of characterizing the set

$$\mathbb{P} = \{\mathbf{p} \in [\mathbf{p}_0] \mid \mathbf{d}_\mathrm{m}(\mathbf{p}) \in [\mathbf{d}]\}, \tag{8.30}$$

where $[\mathbf{p}_0]$ is an initial search box, assumed to be large enough to contain all the configurations of interest. $\mathbb{P}$ then contains all the configurations vectors that are consistent with the map and measurements.

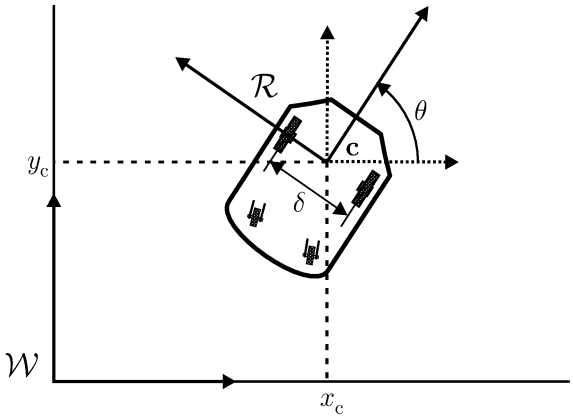


Fig. 8.17. Configuration $\mathbf{p}$ of the robot, $\mathbf{p} = (x_c, y_c, \theta)^T$

In order to be able to build $\mathbf{d}_m(\mathbf{p})$ in Section 8.4.2, we shall first describe how the map is assumed to represent the environment and formulate hypotheses about the measurement process.

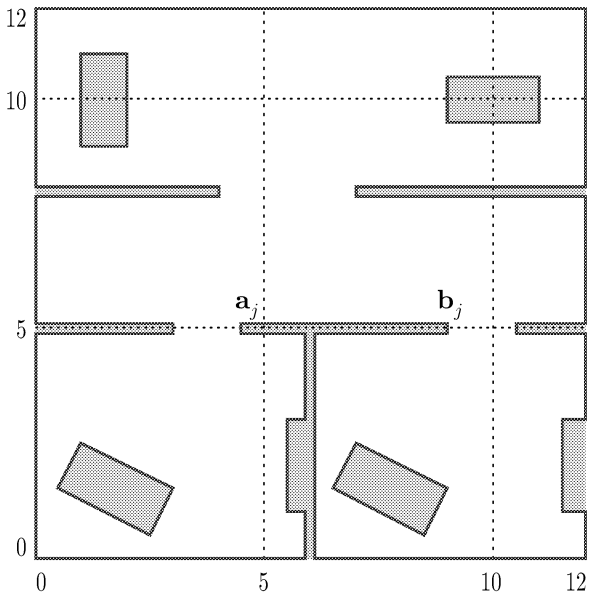


Fig. 8.18. Map of the environment of the robot; distances are in metres

Map. The map $\mathbb{M} = \{[\mathbf{a}_j, \mathbf{b}_j] \mid j = 1, ..., n_{\mathrm{w}}\}$ of the environment is assumed to consist of n_{w} oriented segments $[\mathbf{a}_j, \mathbf{b}_j]$ that describe the landmarks (walls, pillars, etc.). Such a map is represented on Figure 8.18, with the obstacles in grey. By convention, when going from $\mathbf{a}_j$ to $\mathbf{b}_j$, the reflecting face of the segment is on the left. The half-plane $\Delta_{\mathbf{a}_j\mathbf{b}_j}$ located on the reflecting side of the segment $[\mathbf{a}_j, \mathbf{b}_j]$ is therefore characterized by

$$\Delta_{\mathbf{a}_j\mathbf{b}_j} = \left\{ \mathbf{m} \in \mathbb{R}^2 \;\middle|\; \det\left(\overrightarrow{\mathbf{a}_j\mathbf{b}_j}, \overrightarrow{\mathbf{a}_j\mathbf{m}}\right) \geqslant 0 \right\}. \tag{8.31}$$

Remark 8.4 *Such an analytical test of whether a point belongs to a given half-plane will often be used. Another option is to use a scalar product (see Figure 8.19).* ■

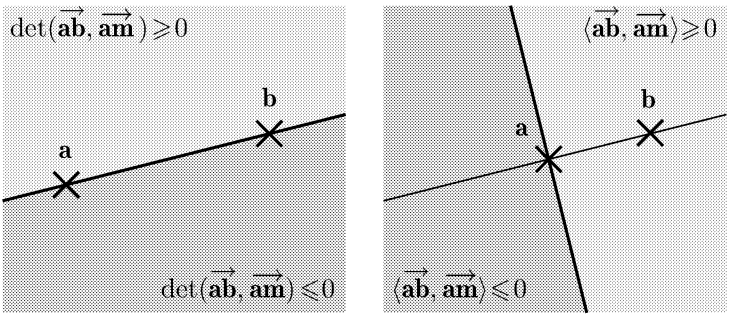


Fig. 8.19. Analytical tests to characterize half-planes; (a) boundary passing through $\mathbf{a}$ and $\mathbf{b}$; (b) boundary orthogonal to $(\mathbf{a}, \mathbf{b})$ and passing through $\mathbf{a}$; cones and strips are obtained as intersections of such half-planes

Measurements. The position of the ith sensor in the robot frame $\mathcal{R}$ is $\tilde{\mathbf{s}}_i = (\tilde{x}_i, \tilde{y}_i)$. This sensor emits an ultrasonic wave assumed to propagate in a cone characterized by its vertex $\tilde{\mathbf{s}}_i$, orientation $\tilde{\theta}_i$ and half-aperture $\tilde{\gamma}_i$ (Figure 8.20). As $\tilde{\gamma}_i$ is frame-independent, $\tilde{\gamma}_i = \gamma_i$. This emission cone will be denoted by $\mathbb{E}(\tilde{\mathbf{s}}_i, \tilde{\theta}_i, \gamma_i)$. The wave emitted will reflect from some landmarks located in the environment of the robot. On landmarks with a rough surface, reflection is both specular and diffuse (Hecht, 1987; Kuc and Siegel, 1987). The angle of incidence on the surface of a given landmark determines whether the reflected wave will hit the sensor. A *limit incidence angle* β_j can be defined, beyond which the reflected wave would not be received if the half-aperture angle γ_i of the emission cone were zero. When γ_i is non-zero, the mean incidence angle of the wave should be less than $\gamma_i + (\beta_j/2)$ for the reflected wave to be received (Kieffer et al., 2001). The value of β_j depends on the roughness of the surface of the obstacle represented by the

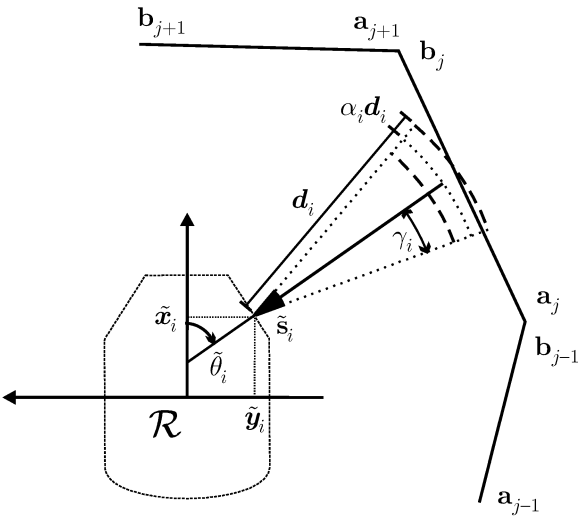


Fig. 8.20. Emission cone

jth segment of the map. It is large for rough surfaces, and very small for smooth surfaces. Two characteristic situations are illustrated by Figure 8.21. When a reflected wave is received by the ith sensor, the time elapsed since

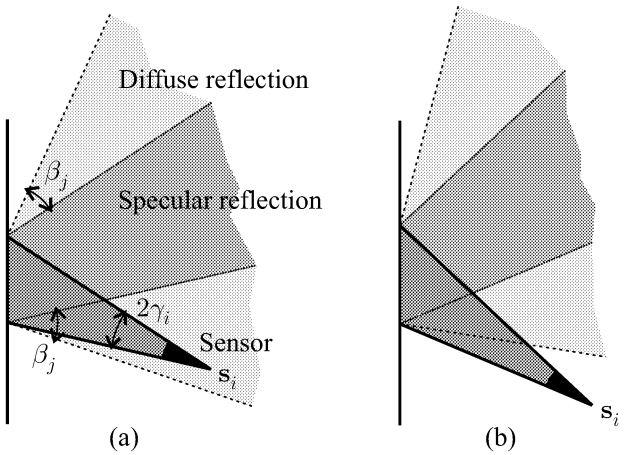


Fig. 8.21. Left: the reflected wave is received if the reflection is diffuse; right: no reflected wave is received

emission is converted into a distance, and interpreted as a measure d_i of the distance between this sensor and the surface of the closed landmark at least

partly located inside the emission cone. Based on laboratory experiments, the relative error on such distance measurements will be assumed to belong to $[-\alpha_i, \alpha_i]$, where α_i is a characteristic of the ith sonar. Thus, the interval $[d_i] = [(1-\alpha_i)\, d_i, (1+\alpha_i)\, d_i]$ is assumed to contain the actual distance to the closest obstacle at least partly located in the emission cone.

Sometimes, however, the wave is only received after several reflections (or even not received at all). A rigorous interpretation of the resulting measure of distance would then become much more complicated, and we shall instead consider such data as outliers, against which the estimation of configuration must be made robust. The almost unavoidable presence of many such outliers makes the problem particularly difficult.

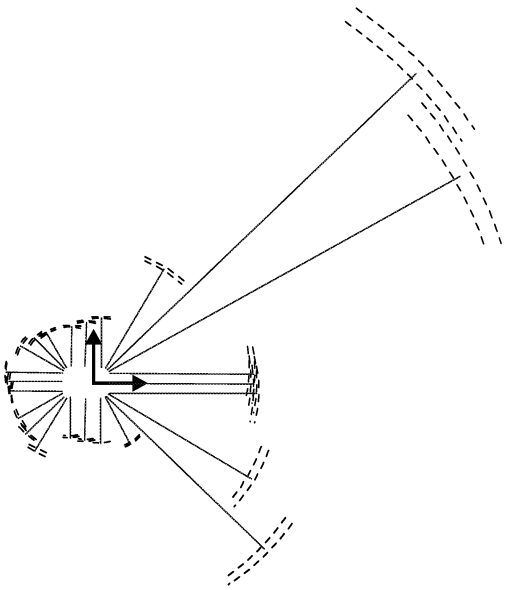


Fig. 8.22. Emission diagram

The n_s measures of distances can be summarized by an *emission diagram* (see Figure 8.22). Each segment in this diagram corresponds to a distance d_i as measured by a sonar. Some obstacle should lay at least in part between the two corresponding arcs of circles, which materialize the relative uncertainty α_i and the half aperture γ_i of the emission cone of this sonar.

8.4.2 Model of the measurement process

For the sake of simplicity, the same model will be used for all sonars and landmarks, so it will be possible to drop the index i from α_i and γ_i and

the index j from β_j. For any given sensor i and configuration vector $\mathbf{p}$, the distance that would be obtained if only one segment of the map were present is computed. This operation is repeated for all the segments of the map, in order to compute the smallest of these distances, which is taken as the ith component of $\mathbf{d}_{\mathrm{m}}(\mathbf{p})$. The details of the operation are given below but may be skipped by proceeding directly to Section 8.4.3.

Consider a sonar, with emission cone $\mathbb{E}(\tilde{\mathbf{s}}, \tilde{\theta}, \gamma)$. For any given configuration $\mathbf{p} = (x_{\mathrm{c}}, y_{\mathrm{c}}, \theta)^{\mathrm{T}}$, $\mathbb{E}$ can be equivalently described in $\mathcal{W}$ by its vertex $\mathbf{s}\,(\mathbf{p})$ and two unit vectors $\overrightarrow{\mathbf{u}_1}(\mathbf{p}, \tilde{\theta}, \gamma)$ and $\overrightarrow{\mathbf{u}_2}(\mathbf{p}, \tilde{\theta}, \gamma)$ corresponding to its edges, given by

$$\overrightarrow{\mathbf{u}_1} = \begin{pmatrix} \cos(\theta + \tilde{\theta} - \gamma) \\ \sin(\theta + \tilde{\theta} - \gamma) \end{pmatrix}, \quad \overrightarrow{\mathbf{u}_2} = \begin{pmatrix} \cos(\theta + \tilde{\theta} + \gamma) \\ \sin(\theta + \tilde{\theta} + \gamma) \end{pmatrix}. \tag{8.32}$$

Omitting the dependency in $\mathbf{p}, \tilde{\theta}$ and γ, one may write $\mathbb{E} = \mathbb{E}\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}\right)$. Since γ is always less than $\pi/2$, the condition for any $\mathbf{m} \in \mathbb{R}^2$ to belong to the emission cone is

$$\mathbf{m} \in \mathbb{E}\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}\right) \Leftrightarrow \left(\det\left(\overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{sm}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{sm}}\right) \leqslant 0\right), \tag{8.33}$$

see Figure 8.19a.

The model of the distance measured by a sonar is based on the following definition, where $[\mathbf{a}, \mathbf{b}]$ is a segment of the map.

Definition 8.1 *The* remoteness *of the emission cone* $\mathbb{E}\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}\right)$ *from the segment* $[\mathbf{a}, \mathbf{b}]$*, denoted by* $r\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}, \mathbf{a}, \mathbf{b}\right)$*, is the smallest distance between the vertex* $\mathbf{s}$ *and the intersection of* $[\mathbf{a}, \mathbf{b}]$ *and* $\mathbb{E}$*, provided that this intersection exists and that* $\mathbf{s}$ *belongs to the half-plane* $\Delta_{\mathbf{ab}}$ *located on the reflecting side of* $[\mathbf{a}, \mathbf{b}]$*; otherwise, the remoteness is infinite.* ■

From (8.31), testing whether $(\mathbf{s} \in \Delta_{\mathbf{ab}})$ is equivalent to testing whether $(\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}}) \geqslant 0)$, and we assume in what follows that this test holds true. The minimum of $||\overrightarrow{\mathbf{sm}}||$ when $\mathbf{m}$ describes $[\mathbf{a}, \mathbf{b}] \cap \mathbb{E}$ should then be evaluated. Let $\mathbf{h}$ be the orthogonal projection of $\mathbf{s}$ onto the line $(\mathbf{a}, \mathbf{b})$. If $\mathbf{h} \in [\mathbf{a}, \mathbf{b}] \cap \mathbb{E}$, then $r\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}, \mathbf{a}, \mathbf{b}\right) = ||\overrightarrow{\mathbf{sh}}||$. Testing whether $\mathbf{h} \in [\mathbf{a}, \mathbf{b}] \cap \mathbb{E}$ amounts to testing whether $\mathbf{h} \in [\mathbf{a}, \mathbf{b}]$ and $\mathbf{h} \in \mathbb{E}$. Testing whether $\mathbf{h} \in [\mathbf{a}, \mathbf{b}]$ is equivalent to testing whether $\mathbf{s}$ belongs to the strip limited by straight lines orthogonal to $[\mathbf{a}, \mathbf{b}]$ and passing through $\mathbf{a}$ and $\mathbf{b}$, so

$$(\mathbf{h} \in [\mathbf{a}, \mathbf{b}]) \Longleftrightarrow \left(\left\langle \overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}} \right\rangle \geqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{ba}}, \overrightarrow{\mathbf{bs}} \right\rangle \geqslant 0\right), \tag{8.34}$$

see Figure 8.19b. Based on (8.33), one can write

$$(\mathbf{h} \in \mathbb{E}) \Longleftrightarrow \left(\det\left(\overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{sh}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{sh}}\right) \leqslant 0\right). \tag{8.35}$$

This test is not so convenient, because the coordinates of $\mathbf{h}$ are difficult to obtain. Since $\mathbf{s} \in \Delta_{\mathbf{ab}}$, $\overrightarrow{\mathbf{sh}}/||\overrightarrow{\mathbf{sh}}||$ is obtained from $\overrightarrow{\mathbf{ab}}/||\overrightarrow{\mathbf{ab}}||$ by a rotation of $-\pi/2$, so (8.35) may be rewritten as

$$(\mathbf{h} \in \mathbb{E}) \Longleftrightarrow \left(\left\langle \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{ab}} \right\rangle \leqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{ab}} \right\rangle \geqslant 0\right). \tag{8.36}$$

Combining (8.34) and (8.36), one obtains

$$\begin{aligned}(\mathbf{h} \in [\mathbf{a},\mathbf{b}] \cap \mathbb{E}) \Longleftrightarrow & \left(\left\langle \overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}} \right\rangle \geqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{ba}}, \overrightarrow{\mathbf{bs}} \right\rangle \geqslant 0\right) \\ \wedge & \left(\left\langle \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{ab}} \right\rangle \leqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{ab}} \right\rangle \geqslant 0\right).\end{aligned} \tag{8.37}$$

If $\mathbf{h} \notin [\mathbf{a},\mathbf{b}] \cap \mathbb{E}$, then $r\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}, \mathbf{a}, \mathbf{b}\right)$ is either infinite (if $[\mathbf{a},\mathbf{b}] \cap \mathbb{E} = \varnothing$) or obtained for one of the extremities of the segment $[\mathbf{a},\mathbf{b}] \cap \mathbb{E}$. These extremities belong to $\mathbb{K} = \{\mathbf{a}, \mathbf{b}, \mathbf{h}_1, \mathbf{h}_2\}$, where $\mathbf{h}_1$ and $\mathbf{h}_2$ are the intersections of the line $(\mathbf{a},\mathbf{b})$ with the lines $\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}\right)$ and $\left(\mathbf{s}, \overrightarrow{\mathbf{u}_2}\right)$. For the example of Figure 8.23, $r\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}, \mathbf{a}, \mathbf{b}\right) = ||\overrightarrow{\mathbf{sb}}||$. A test of whether an element of $\mathbb{K}$ belongs to $[\mathbf{a},\mathbf{b}] \cap \mathbb{E}$ is easily derived from (8.33). For $\mathbf{v} \in \{\mathbf{a},\mathbf{b}\}$

$$(\mathbf{v} \in \mathbb{E}) \Longleftrightarrow \left(\det\left(\overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{sv}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{sv}}\right) \leqslant 0\right). \tag{8.38}$$

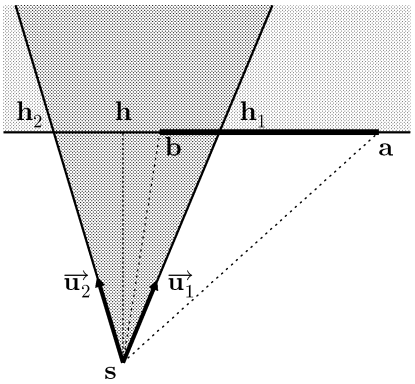


Fig. 8.23. Remoteness of the isolated segment $[\mathbf{a},\mathbf{b}]$ from the sensor $\mathbf{s}$; the half-plane located on the non-reflecting side of $[\mathbf{a},\mathbf{b}]$ is in light grey; the emission cone $\mathbb{E}$ is in dark grey

By construction, $\mathbf{h}_1$ and $\mathbf{h}_2$ belong to $\mathbb{E} \cap (\mathbf{a},\mathbf{b})$; one thus has only to check whether they belong to $[\mathbf{a},\mathbf{b}]$, which is equivalent to checking whether they belong to the cone with vertex $\mathbf{s}$ and edges $\overrightarrow{\mathbf{sa}}$ and $\overrightarrow{\mathbf{sb}}$. From (8.33) one obtains, for $i = 1, 2$,

$$(\mathbf{h}_i \in [\mathbf{a},\mathbf{b}] \cap \mathbb{E}) \Longleftrightarrow \left(\det\left(\overrightarrow{\mathbf{sa}}, \overrightarrow{\mathbf{u}_i}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{sb}}, \overrightarrow{\mathbf{u}_i}\right) \leqslant 0\right). \tag{8.39}$$

Finally, if neither $\mathbf{h}$ nor any element of $\mathbb{K}$ belongs to $[\mathbf{a},\mathbf{b}] \cap \mathbb{E}$, then it has been proved that $[\mathbf{a},\mathbf{b}] \cap \mathbb{E} = \varnothing$, and the remoteness of $\mathbb{E}$ from $[\mathbf{a},\mathbf{b}]$ is taken as infinite.

Table 8.11. Function evaluating the remoteness of a cone from an isolated segment

Algorithm $r\left(\text{in: } \mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}, \mathbf{a}, \mathbf{b};\ \text{out: } r\right)$	
1	if $\left(\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}}) < 0\right)$
2	$r := +\infty$; return;
3	if $\left(\left\langle \overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}} \right\rangle \geqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{ba}}, \overrightarrow{\mathbf{bs}} \right\rangle \geqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{ab}} \right\rangle \leqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{ab}} \right\rangle \geqslant 0\right)$
4	then $r_{\mathbf{h}} := \|\overrightarrow{\mathbf{sh}}\| = \ell\left(\mathbf{s}, (\mathbf{a}, \mathbf{b})\right)$, else $r_{\mathbf{h}} := +\infty$;
5	if $\left(\det\left(\overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{sa}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{sa}}\right) \leqslant 0\right)$
6	then $r_{\mathbf{a}} := \|\overrightarrow{\mathbf{sa}}\|$, else $r_{\mathbf{a}} := +\infty$;
7	if $\left(\det\left(\overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{sb}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{u}_2}, \overrightarrow{\mathbf{sb}}\right) \leqslant 0\right)$
8	then $r_{\mathbf{b}} := \|\overrightarrow{\mathbf{sb}}\|$, else $r_{\mathbf{b}} := +\infty$;
9	for $i := 1$ to 2
10	if $\left(\det\left(\overrightarrow{\mathbf{sa}}, \overrightarrow{\mathbf{u}_i}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{\mathbf{sb}}, \overrightarrow{\mathbf{u}_i}\right) \leqslant 0\right)$
11	then $r_{\mathbf{h}_i} := \|\overrightarrow{\mathbf{sh}_i}\| = \ell_{\overrightarrow{\mathbf{u}_i}}\left(\mathbf{s}, (\mathbf{a}, \mathbf{b})\right)$, else $r_{\mathbf{h}_i} := +\infty$;
12	$r := \min\left(r_{\mathbf{h}}, r_{\mathbf{a}}, r_{\mathbf{b}}, r_{\mathbf{h}_1}, r_{\mathbf{h}_2}\right)$.

Table 8.11 presents a function evaluating $r\left(\mathbf{s}, \overrightarrow{\mathbf{u}_1}, \overrightarrow{\mathbf{u}_2}, \mathbf{a}, \mathbf{b}\right)$ for an isolated segment $[\mathbf{a}, \mathbf{b}]$, which is based on these tests. At Step 4, the distance $\ell\left(\mathbf{s}, (\mathbf{a}, \mathbf{b})\right)$ from $\mathbf{s}$ to the line $(\mathbf{a}, \mathbf{b})$ (Figure 8.24) is given by

$$\ell\left(\mathbf{s}, (\mathbf{a}, \mathbf{b})\right) = \|\overrightarrow{\mathbf{sh}}\| = \frac{|\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}})|}{\|\overrightarrow{\mathbf{ab}}\|}, \tag{8.40}$$

and, at Step 11, the distance $\ell_{\overrightarrow{\mathbf{u}}}\left(\mathbf{s}, (\mathbf{a}, \mathbf{b})\right)$ from $\mathbf{s}$ to the line $(\mathbf{a}, \mathbf{b})$ along the unit vector $\overrightarrow{\mathbf{u}}$ (Figure 8.24) is given by

$$\begin{aligned} \ell_{\overrightarrow{\mathbf{u}}}\left(\mathbf{s}, (\mathbf{a}, \mathbf{b})\right) = \|\overrightarrow{\mathbf{sm}}\| &= \frac{\|\overrightarrow{\mathbf{ah}}\|}{|\sin\theta|} \\ &= \frac{|\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}})|}{\|\overrightarrow{\mathbf{ab}}\|\,|\sin\theta|} = \frac{|\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}})|}{|\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{u}})|}. \end{aligned} \tag{8.41}$$

When n_{w} segments are present, the fact that some of them may not be illuminated, because they lie in the shadow of others that are closer to the sensor, must be taken into account. Let $r_{ij}(\mathbf{p})$ be the remoteness of the jth segment, taken as isolated, from the ith sensor if the configuration is $\mathbf{p}$

$$r_{ij}\left(\mathbf{p}\right) = r\left(\mathbf{s}_i\left(\mathbf{p}\right), \overrightarrow{\mathbf{u}_{1i}}\left(\mathbf{p}, \tilde{\theta}_i, \gamma\right), \overrightarrow{\mathbf{u}_{2i}}\left(\mathbf{p}, \tilde{\theta}_i, \gamma\right), \mathbf{a}_j, \mathbf{b}_j\right). \tag{8.42}$$

The model of the measure that will be performed by the ith sensor if the configuration is $\mathbf{p}$ is then taken as

$$\left(\mathbf{d}_{\mathrm{m}}\right)_i\left(\mathbf{p}\right) = \min_{j=1,\ldots,n_{\mathrm{w}}} r_{ij}\left(\mathbf{p}\right). \tag{8.43}$$

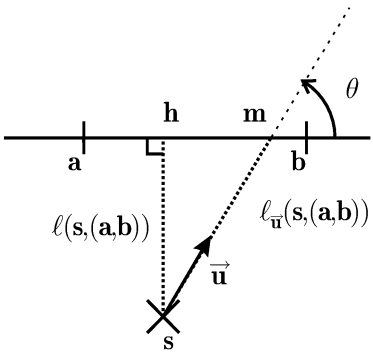


Fig. 8.24. Distances from the point $\mathbf{s}$ to the line $(\mathbf{a}, \mathbf{b})$

By evaluating (8.43) for $i = 1, \ldots, n_\mathrm{s}$, one gets the vector $\mathbf{d}_\mathrm{m}(\mathbf{p})$, to be compared with the interval distance data $[\mathbf{d}]$. The function $\mathbf{d}_\mathrm{m}(\mathbf{p})$ is evaluated by the algorithm of Table 8.12.

Table 8.12. Function computing the distances expected when the configuration is $\mathbf{p}$

Algorithm $\mathbf{d}_\mathrm{m}$ (in: $\mathbf{p}$; out: $\mathbf{d}_\mathrm{m}$)	
1	for $i := 1$ to n_s
2	$\mathbf{s}_i := \begin{pmatrix} x_\mathrm{c} \\ y_\mathrm{c} \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \widetilde{\mathbf{s}}_i$;
3	$\overrightarrow{\mathbf{u}_{1i}} := \begin{pmatrix} \cos(\theta + \tilde{\theta}_i - \gamma) \\ \sin(\theta + \tilde{\theta}_i - \gamma) \end{pmatrix}$; $\overrightarrow{\mathbf{u}_{2i}} := \begin{pmatrix} \cos(\theta + \tilde{\theta}_i + \gamma) \\ \sin(\theta + \tilde{\theta}_i + \gamma) \end{pmatrix}$;
4	$(\mathbf{d}_\mathrm{m})_i(\mathbf{p}) := +\infty$;
5	for $j := 1$ to n_w
6	$(\mathbf{d}_\mathrm{m})_i(\mathbf{p}) := \min\left((\mathbf{d}_\mathrm{m})_i(\mathbf{p}), r\left(\mathbf{s}_i, \overrightarrow{\mathbf{u}_{1i}}, \overrightarrow{\mathbf{u}_{2i}}, \mathbf{a}_j, \mathbf{b}_j\right)\right)$.

Remark 8.5 *The complexity of evaluating* $\mathbf{d}_\mathrm{m}$ *is bilinear in* n_s *and* n_w. ■

8.4.3 Set inversion

The problem of characterizing

$$\begin{aligned} \mathbb{P} &= \{\mathbf{p} \in [\mathbf{p}_0] \mid \mathbf{d}_\mathrm{m}(\mathbf{p}) \in [\mathbf{d}]\} \\ &= [\mathbf{p}_0] \cap (\mathbf{d}_\mathrm{m})^{-1}([\mathbf{d}]), \end{aligned} \tag{8.44}$$

may then be solved using SIVIA, described in Table 3.1, page 56. The only prerequisite is to have an inclusion function $[\mathbf{d}_\mathrm{m}](.)$ for $\mathbf{d}_\mathrm{m}(.)$. The model

$\mathbf{d}_{\mathrm{m}}(.)$ is based on the evaluation of remoteness, which involves a number of conditional branchings. When evaluating $[\mathbf{d}_{\mathrm{m}}]([\mathbf{p}])$ on any given box of parameter space, it is necessary to decide which branch(es) should be executed. A method proposed in Jaulin et al. (2000), based on the notion of χ-function (Kearfott, 1996a), is now presented. If t is the Boolean result of a test and y and z are two real numbers, then

$$\chi(t,y,z)=\begin{cases} y & \text{if } t=1,\\ z & \text{if } t=0. \end{cases} \tag{8.45}$$

The interval counterpart of $\chi(t,y,z)$ is given by

$$[\chi]([t],[y],[z])=\begin{cases} [y] & \text{if } [t]=1,\\ [z] & \text{if } [t]=0,\\ [y]\sqcup[z] & \text{if } [t]=[0,1]. \end{cases} \tag{8.46}$$

The result of the evaluation of a test based on $[\chi]$ is therefore always an interval.

An interval counterpart of Table 8.11 is given by Table 8.13.

Table 8.13. Inclusion function for the remoteness of a cone from an isolated segment

Algorithm $[r]\left(\text{in: } [\mathbf{s}], \overrightarrow{[\mathbf{u}_1]}, \overrightarrow{[\mathbf{u}_2]}, \mathbf{a}, \mathbf{b};\ \text{out: } [r]\right)$	
1	$[t_r] := \left(\det(\overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{as}}) \geqslant 0\right)$;
	if $[t_r]=0$ then $[r]:=+\infty$; return;
2	$[t_{\mathbf{h}}] := \left(\left\langle \overrightarrow{\mathbf{ab}}, \overrightarrow{\mathbf{a}\,[\mathbf{s}]} \right\rangle \geqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{ba}}, \overrightarrow{\mathbf{b}\,[\mathbf{s}]} \right\rangle \geqslant 0\right)$
	$\wedge \left(\left\langle \overrightarrow{\mathbf{ab}}, \overrightarrow{[\mathbf{u}_1]} \right\rangle \leqslant 0\right) \wedge \left(\left\langle \overrightarrow{\mathbf{ab}}, \overrightarrow{[\mathbf{u}_2]} \right\rangle \geqslant 0\right)$;
3	$[r_{\mathbf{h}}] := [\chi]([t_{\mathbf{h}}], [\ell]([\mathbf{s}], (\mathbf{a},\mathbf{b})), +\infty)$;
4	$[t_{\mathbf{a}}] := \left(\det\left(\overrightarrow{[\mathbf{u}_1]}, \overrightarrow{[\mathbf{s}]\,\mathbf{a}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{[\mathbf{u}_2]}, \overrightarrow{[\mathbf{s}]\,\mathbf{a}}\right) \leqslant 0\right)$;
5	$[r_{\mathbf{a}}] := [\chi]\left([t_{\mathbf{a}}], ||\overrightarrow{[\mathbf{s}]\,\mathbf{a}}||, +\infty\right)$;
6	$[t_{\mathbf{b}}] := \left(\det\left(\overrightarrow{[\mathbf{u}_1]}, \overrightarrow{[\mathbf{s}]\,\mathbf{b}}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{[\mathbf{u}_2]}, \overrightarrow{[\mathbf{s}]\,\mathbf{b}}\right) \leqslant 0\right)$;
7	$[r_{\mathbf{b}}] := [\chi]\left([t_{\mathbf{b}}], ||\overrightarrow{[\mathbf{s}]\,\mathbf{b}}||, +\infty\right)$;
8	for $i := 1$ to 2
9	$[t_{\mathbf{h}_i}] := \left(\det\left(\overrightarrow{[\mathbf{s}]\,\mathbf{a}}, \overrightarrow{[\mathbf{u}_i]}\right) \geqslant 0\right) \wedge \left(\det\left(\overrightarrow{[\mathbf{s}]\,\mathbf{b}}, \overrightarrow{[\mathbf{u}_i]}\right) \leqslant 0\right)$;
10	$[r_{\mathbf{h}_i}] := [\chi]\left([t_{\mathbf{h}_i}], \left[\ell_{\overrightarrow{[\mathbf{u}_i]}}\right]([\mathbf{s}], (\mathbf{a},\mathbf{b})), +\infty\right)$;
11	$[r] := \min([r_{\mathbf{h}}], [r_{\mathbf{a}}], [r_{\mathbf{b}}], [r_{\mathbf{h}_1}], [r_{\mathbf{h}_2}])$;
12	$[r] := [\chi]([t_r], [r], +\infty)$.

In this table, $\overrightarrow{\mathbf{a}\,[\mathbf{s}]}$ stands for the set of all the vectors with origin at $\mathbf{a}$ and extremity in the box $[\mathbf{s}]$. The box $[\mathbf{s}]$, guaranteed to contain the location

of the sensor $\mathbf{s}$ for any configuration in $[\mathbf{p}] = ([x_c], [y_c], [\theta])^T$, is evaluated by replacing all the occurrences of the real variables in (8.29) by their interval counterparts. Similarly, the characteristics of the cone (8.32) are evaluated as $[\mathbb{E}] = \mathbb{E}\left([\mathbf{s}], \overrightarrow{[\mathbf{u}_1]}, \overrightarrow{[\mathbf{u}_2]}\right)$. Finally, the minimum of two intervals is defined as

$$\min([a], [b]) = \left[\min(\underline{a}, \underline{b}), \min(\overline{a}, \overline{b})\right], \tag{8.47}$$

and the extension to more than two intervals is straightforward.

Remark 8.6 *The algorithm of Table 8.13 could be accelerated by removing Step* 12. *The price to be paid would be a possible augmentation of the number of outliers, see Section 8.4.4.* ■

It is now trivial to obtain an inclusion function $[\mathbf{d}_m](.)$ for $\mathbf{d}_m(.)$, based on Tables 8.12 and 8.13. This inclusion function is presented in Table 8.14. The complexity of evaluating $[\mathbf{d}_m]$ remains bilinear in n_s and n_w.

Table 8.14. Inclusion function for the measurement model

Algorithm $[\mathbf{d}_m]$ (in: $[\mathbf{p}]$; out: $[\mathbf{d}_m]$)	
1	for $i := 1$ to n_s
2	$[\mathbf{s}_i] := \begin{pmatrix} x_c \\ y_c \end{pmatrix} + \begin{pmatrix} \cos[\theta] & -\sin[\theta] \\ \sin[\theta] & \cos[\theta] \end{pmatrix} \widetilde{\mathbf{s}}_i$;
3	$\overrightarrow{[\mathbf{u}_{1i}]} := \begin{pmatrix} \cos([\theta] + \tilde{\theta}_i - \gamma) \\ \sin([\theta] + \tilde{\theta}_i - \gamma) \end{pmatrix}$; $\overrightarrow{[\mathbf{u}_{2i}]} := \begin{pmatrix} \cos([\theta] + \tilde{\theta}_i + \gamma) \\ \sin([\theta] + \tilde{\theta}_i + \gamma) \end{pmatrix}$;
4	$[\mathbf{d}_m]_i([\mathbf{p}]) := +\infty$;
5	for $j := 1$ to n_w
6	$[\mathbf{d}_m]_i([\mathbf{p}]) := \min\left([\mathbf{d}_m]_i([\mathbf{p}]), [r]\left([\mathbf{s}_i], \overrightarrow{[\mathbf{u}_{1i}]}, \overrightarrow{[\mathbf{u}_{2i}]}, \mathbf{a}_j, \mathbf{b}_j\right)\right)$.

8.4.4 Dealing with outliers

In the context of robot localization, outliers are almost unavoidable. Outliers are data points for which the hypotheses made on the bounds of measurement error are violated. They may correspond to multiple reflections, to the presence of persons or pieces of furniture, to sensor failures, to an outdated map, etc. In the presence of such outliers, the set $\mathbb{P}$, as defined by (8.30), may turn out to be empty. Introducing the relaxing function (see Section 6.3.3, page 160)

$$\lambda(\mathbf{p}) = \frac{\sum_{i=1}^{n_s} \pi_{[d_i]}(\mathbf{p})}{n_s}, \tag{8.48}$$

where

$$\pi_{[d_i]}(\mathbf{p}) = \begin{cases} 1 & \text{if } (\mathbf{d}_{\mathrm{m}})_i(\mathbf{p}) \in [d_i], \\ 0 & \text{if } (\mathbf{d}_{\mathrm{m}})_i(\mathbf{p}) \notin [d_i], \end{cases} \tag{8.49}$$

and characterizing the set

$$\mathbb{P}^q = \left\{ \mathbf{p} \in [\mathbf{p}_0] \mid \lambda(\mathbf{p}) \geqslant 1 - \frac{q}{n_{\mathrm{s}}} \right\}, \tag{8.50}$$

allows up to q outliers to be tolerated. To choose the value of q, one may use GOMNE (Jaulin et al., 1996), the principle of which has been recalled in Section 6.3.3, page 160.

Remark 8.7 *Tests have been proposed in Kieffer et al. (1999, 2000) that make it possible rapidly to eliminate vast domains of the prior search domain during set inversion, thus accelerating localization quite considerably. The next example was treated using these tests.* ■

8.4.5 Static localization example

Although based on simulations, this example is quite realistic, and similar results have been obtained on real data (Lévêque, 1998). The characteristics of the robot are those of the robot of Figure 8.16, which is equipped with $n_{\mathrm{s}} = 24$ sonars. Each of them has been found experimentally to have a half-aperture γ of 0.2 rad and a distance relative inaccuracy α of 2% within its operating range.

Table 8.15. Distances measurements provided by the sonars

Sensor i	1	2	3	4	5	6	7	8
d_i (m)	3.24	3.21	9.02	9.60	2.58	1.11	1.01	0.91

Sensor i	9	10	11	12	13	14	15	16
d_i (m)	0.89	0.95	1.10	1.27	1.21	1.14	1.14	1.21

Sensor i	17	18	19	20	21	22	23	24
d_i (m)	1.39	0.91	0.96	1.02	1.19	4.95	3.71	3.30

The robot is placed in an environment described by the map of Figure 8.18. All the obstacles have a limit incidence angle β of 0.6 rad. The (unknown) configuration of the robot is $(x_{\mathrm{c}}, y_{\mathrm{c}}, \theta) = (8\text{ m}, 3.5\text{ m}, 6\text{ rad})$. The distance measurements provided by the on-board sensors are reported in Table 8.15 and correspond to the emission diagram of Figure 8.22. In order to simulate one of the phenomena that may lead to outliers, whenever

the incidence angle of the wave emitted by a given sonar is larger than $\gamma + (\beta/2)$, the measured distance is taken at random according to a uniform distribution between 0.5 m and 10 m. The search box $[\mathbf{x}_0]$ is taken equal to $[0\,\text{m}, 12\,\text{m}] \times [0\,\text{m}, 12\,\text{m}] \times [0\,\text{rad}, 2\pi\,\text{rad}]$.

The static localization procedure returns no solution until at least three outliers are tolerated. When $q = 3$, the outer approximation $\overline{\mathbb{P}}^3$ of the solution set, presented in Figure 8.25, consists of two disconnected subsets, one of which is *guaranteed* to contain the actual configuration provided that there are no more than three outliers. This outer approximation satisfies

$$\begin{aligned}\overline{\mathbb{P}}^3 \subset\ & [7.93\,\text{m}, 8.07\,\text{m}] \times [3.43\,\text{m}, 3.57\,\text{m}] \times [5.90\,\text{rad}, 6.10\,\text{rad}] \\ & \cup [1.93\,\text{m}, 2.07\,\text{m}] \times [3.43\,\text{m}, 3.57\,\text{m}] \times [5.90\,\text{rad}, 6.10\,\text{rad}]\,.\end{aligned}$$

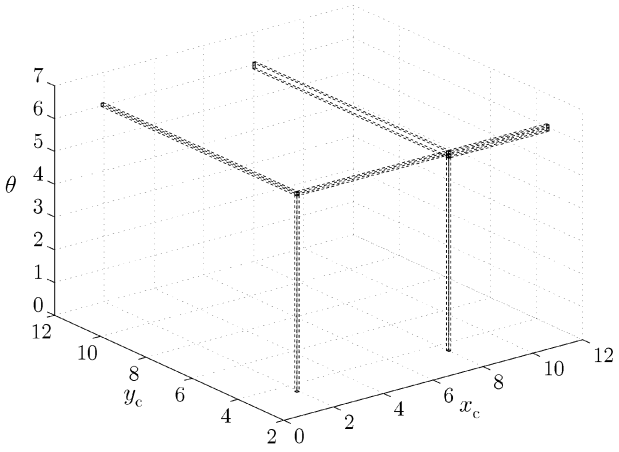


Fig. 8.25. $\overline{\mathbb{P}}^3$ and its projections

Figure 8.26 displays two configurations belonging to $\overline{\mathbb{P}}^3$. The measurements corresponding to outliers have been drawn in bold. The ambiguity in the localization is a consequence of the local symmetries of the map, and should of course not be interpreted as a defect of the estimation method. Guaranteed set inversion only reveals the existence of an identifiability problem, whereas most other localization techniques would limit themselves to providing a single point estimate of the configuration of the robot, without any warning as to the possible existence of radically different solutions.

When q is increased, the volume of the outer approximation of the solution set (computed by adding the volumes of the boxes of $\overline{\mathbb{P}}^q$) may also increase, because some informative measurements may now be ignored. A compromise may thus have to be struck between robustness and accuracy. Here, for $q = 5$,

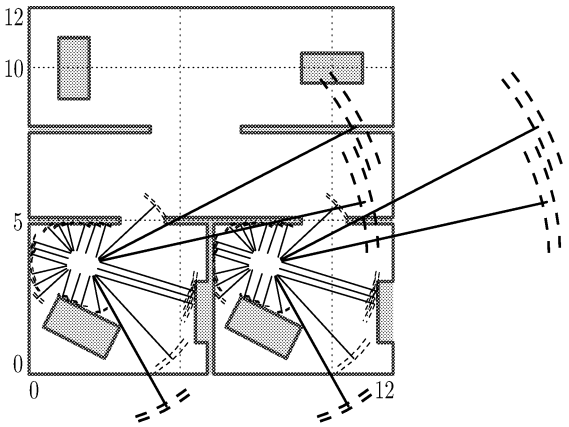


Fig. 8.26. Two of the possible configurations obtained by the static localization procedure; outliers are in bold

the volume of $\overline{\mathbb{P}}^q$ increases notably, with additional disconnected components indicative of the fact that the data considered as outliers are not always the same. Computation also gets more intensive, as eliminating some portions of the initial search box becomes more difficult. Table 8.16 indicates the volumes obtained and computing times from $q = 0$ to $q = 5$, on a PENTIUM-II 450.

Table 8.16. Results of the static localization procedure as a function of the maximum number q of outliers tolerated

q	Volume of $\overline{\mathbb{P}}^q$	Computing time	Cumulated computing time
0	0	25 s	25 s
1	0	48 s	73 s
2	0	89 s	162 s
3	0.0035	129 s	291 s
4	0.0043	167 s	458 s
5	0.0078	216 s	674 s

Remark 8.8 *In Lévêque (1998), the results provided by an earlier version of this localization algorithm were compared with those obtained by extended Kalman filtering. As could be expected, computing time was definitely in favour of the latter, but the outliers had to be weeded out and the association between the measurements and the landmarks of the map had to be performed before the extended Kalman filter could be employed. The extended Kalman filter was of course unable to detect any ambiguity in the localization due to the local symmetries in the map.* ■

8.4.6 Tracking

Assume now that the robot may be moving. Its configuration is thus a function of time, which will be called *state*. To estimate this state in real time, we shall use the recursive causal state estimator of Table 6.13, page 182. Only the results $\overline{\mathbb{P}}_0 = \overline{\mathbb{P}}^q$ of the initial static localization and the distance data obtained during motion may be taken into account.

At each step k, the estimator computes a set $\overline{\mathbb{P}}_k$ guaranteed to contain all the values of the state that are compatible with the information available up to step k, by alternating prediction and correction phases. Recall that the state equation (6.91) of the model underlying the state estimator involves two functions $\mathbf{f}(.)$ and $\mathbf{g}(.)$. The prediction of the evolution of the state vector between steps is performed with the help of $\mathbf{f}(.)$, while $\mathbf{g}(.)$ models the measurements at step k and is used for the correction step. Neglecting the displacement that takes place during the measurements at step k, one can view the correction step as static, so $\mathbf{d}_{\mathrm{m}}(.)$ takes the role of $\mathbf{g}(.)$, and only $\mathbf{f}(.)$ needs to be built.

The robot is assumed to move slowly enough for a kinematic description to be realistic. The components of the state vector then satisfy

$$\begin{aligned} \frac{d\theta}{dt} &= \rho\,\frac{\omega_{\mathrm{r}} - \omega_{\mathrm{l}}}{\delta}, \\ \frac{dx_{\mathrm{c}}}{dt} &= \rho\,\frac{\omega_{\mathrm{r}} + \omega_{\mathrm{l}}}{2}\cos\theta, \\ \frac{dy_{\mathrm{c}}}{dt} &= \rho\,\frac{\omega_{\mathrm{r}} + \omega_{\mathrm{l}}}{2}\sin\theta, \end{aligned} \tag{8.51}$$

where ω_{l} and ω_{r} are respectively the instantaneous rotation speeds of the left and right driving wheels, ρ is the radius of these wheels and δ is the average distance between the points at which they are in contact with the ground (see Figure 8.17). The behaviour of the robot is controlled by acting on the input variables ω_{l} and ω_{r}. An exact discretization of (8.51) is performed, assuming that the rotation speeds of the driving wheels are constant during a sampling period τ. The orientation of the robot at time $k+1$ given its orientation at time k is then given by

$$\theta_{k+1} = \theta_k + \tau\rho\,\frac{\omega_{\mathrm{r}} - \omega_{\mathrm{l}}}{\delta}. \tag{8.52}$$

If ω_{l} and ω_{r} are identical, then

$$\begin{aligned} (x_{\mathrm{c}})_{k+1} &= (x_{\mathrm{c}})_k + \tau\rho\,\frac{\omega_{\mathrm{r}} + \omega_{\mathrm{l}}}{2}\cos\theta_k, \\ (y_{\mathrm{c}})_{k+1} &= (y_{\mathrm{c}})_k + \tau\rho\,\frac{\omega_{\mathrm{r}} + \omega_{\mathrm{l}}}{2}\sin\theta_k, \end{aligned} \tag{8.53}$$

else

$$(x_c)_{k+1} = (x_c)_k + \delta \frac{\omega_r + \omega_l}{\omega_r - \omega_l} \cos(\theta_k + \tau\rho \frac{(\omega_r - \omega_l)}{2\delta}) \sin(\tau\rho \frac{(\omega_r - \omega_l)}{2\delta}),$$
$$(y_c)_{k+1} = (y_c)_k + \delta \frac{\omega_r + \omega_l}{\omega_r - \omega_l} \sin(\theta_k + \tau\rho \frac{(\omega_r - \omega_l)}{2\delta}) \sin(\tau\rho \frac{(\omega_r - \omega_l)}{2\delta}). \tag{8.54}$$

During turns, the distance δ between the points of contact of the driving wheels with the ground is not known precisely. This is why δ will be taken as an interval $[\delta] = [0.57 \text{ m}, 0.63 \text{ m}]$. The prediction of the evolution of the state of the robot depends non-linearly on $[\delta]$, and the deviation between the actual value of δ and the midpoint of $[\delta]$ can be seen as a bounded state perturbation.

The correction step is implemented using the static localization procedure. Only the configuration domain corresponding to the set obtained during the previous prediction step is explored, which speeds up the procedure spectacularly.

Possible outliers only have to be taken into account during the correction step and are thus treated as in the static case, except that the maximum number of outliers q to be tolerated may now depend on the time instant k. One of the possible strategies for tuning q_k is to search first for the set $\overline{\mathbb{P}}_k^0$ of the state vectors consistent with all distance measurements performed at time k ($q_k = 0$), and then to increment q_k by one as long as $\overline{\mathbb{P}}_k^{q_k}$ remains empty.

In practice, it may be advisable to give a larger value to q_k than required to make $\overline{\mathbb{P}}_k^{q_k}$ non-empty. As a matter of fact, the robust state estimation procedure advocated here is only guaranteed if at each step the actual number of outliers is lower than or equal to q_k the maximum number of outliers tolerated. In the example treated in the next section, caution has been exercised by allowing at each step one more outlier than strictly necessary to make $\overline{\mathbb{P}}_k^{q_k}$ non-empty. Such a precautionary measure of course usually comes at the cost of some deterioration of precision since a larger number of measurements must be neglected, so again a compromise must be struck between robustness and precision.

8.4.7 Example

Consider again the situation of Section 8.4.5, but assume now that the robot is in motion. The map of the environment is still described by Figure 8.18, and the initial outer approximation $\overline{\mathbb{P}}_0$ of the state vector corresponds to the result of the static localization performed in Section 8.4.5.

The robot actually moves from the room located on the right of the bottom of Figure 8.18 to that at the top. New measurements are taken every second to correct the predicted set containing the state of the robot. The evolution with k of the projection of the set $\overline{\mathbb{P}}_k^{q_k}$ onto the (x, y) plane is represented on the left of Figures 8.27 and 8.28. On the right of these figures,

the robot is represented in configurations belonging to the estimated solution set. Up to $k = 6$, this set consists of two disconnected parts. After $k = 6$, the left part of the predicted set can be eliminated, because configurations in the right part are consistent with all but five measurements at $k = 6$, whereas to keep configurations in the left part, one should at least tolerate eight outliers at $k = 6$.

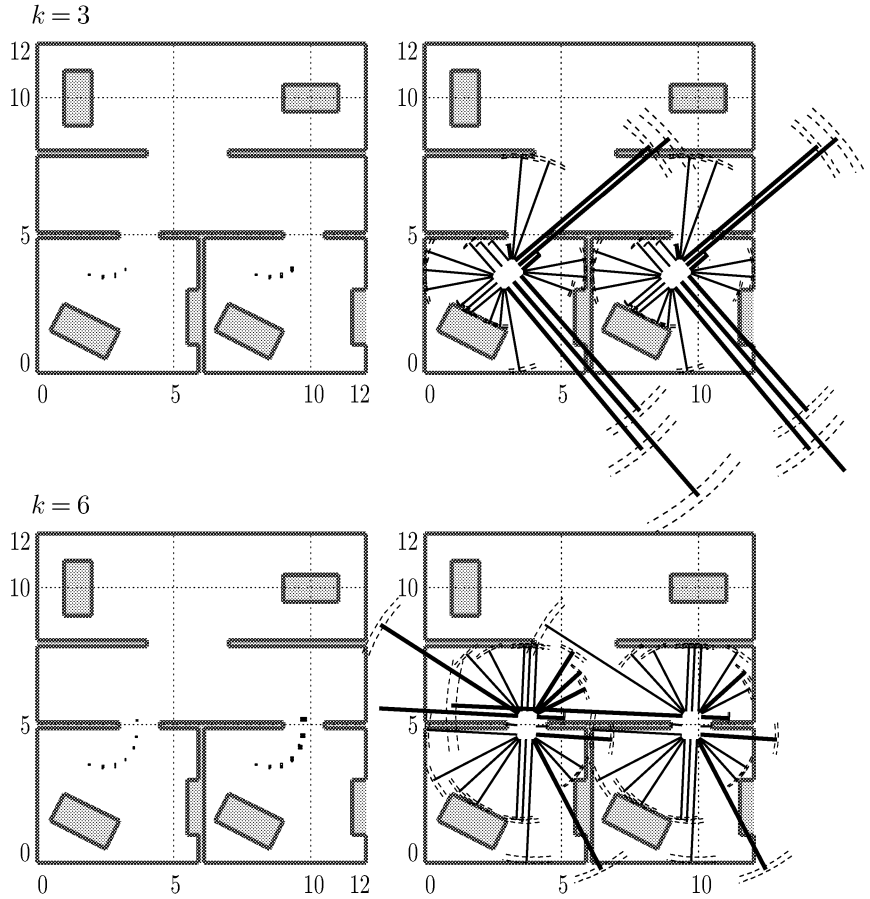


Fig. 8.27. Evolution of the projection onto the (x, y) plane of an outer approximation of the solution set (left), and configurations belonging to this set (right); measurements are collected every second; up to $k = 6$, the local symmetry of the environment allows two types of radically different solutions

Simulating a 20-second scenario takes about 15 s on a PENTIUM-II 450. Each correction step takes much less time than in the static localization phase, because the prior search space results from the previous prediction step and is

much smaller than for the initialization of the procedure. It is thus possible to track the robot in real time, taking into account 24 measurements per second, which seems reasonable for a robot moving relatively slowly.

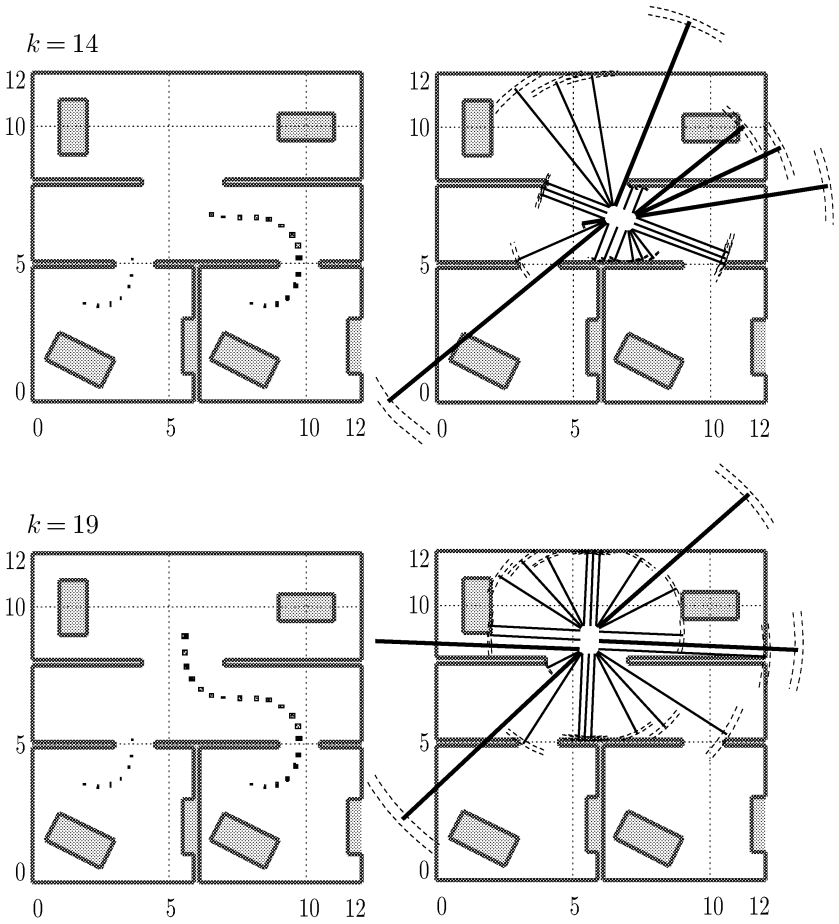


Fig. 8.28. Evolution of the projection on the (x, y) plane of an outer approximation of the solution set (left), and configurations belonging to this set (right); measurements are collected every second; after $k = 6$ the ambiguity due to the local symmetry is eliminated

Table 8.17 indicates the evolution with time of q_k, the maximum number of outliers tolerated. Let $\underline{q}_k$ be the smallest value of q_k such that $\overline{\mathbb{P}}_k^{q_k}$ is not empty; to guard against one non-detected outlier, q_k was taken equal to $\underline{q}_k+1$. Data points corresponding to outliers are indicated in bold on Figures 8.27 and 8.28. Transiently, q_k turns out to be very large (more than one third of

all measurements), which corresponds in this simulated case to configurations for which many incidence angles are beyond the limit β. Computing time is not significantly increased in this type of situation, because of the small size of the search domains.

Table 8.17. Evolution of q_k, the maximum number of outliers tolerated, as a function of time index k

Instant k	1	2	3	4	5	6	7	8	9	10
q_k	3	7	8	9	6	6	7	8	9	8

Instant k	11	12	13	14	15	16	17	18	19	20
q_k	8	10	9	7	10	9	6	7	5	5

Remark 8.9 *In actual applications, data acquisition is often sequential (e.g., with sensors interrogated by groups of four). This constraint can easily be taken into account by using only the last data acquired for the updating of the predicted configuration.* ■

8.5 Conclusions

Three problems of robotics have been considered, for which interval analysis was able to give guaranteed solutions.

Interval solvers made it possible to solve the forward kinematic problem for a Stewart–Gough platform in the most general non-planar case. Trigonometric functions were handled as such, without having to increase the number of unknowns to eliminate them. The real solutions, which are the only ones of interest, were readily isolated. It was not necessary to assume that the geometric coefficients were small integers, and the solutions were provided with an estimate of their precisions. It would also be easy to take into account uncertainty in the geometrical parameters defining the platform or in the lengths of its limbs by giving them interval values.

In the context of path planning, two algorithms have been presented, based on a combination of interval and graph theoretic tools. Interval analysis is used to test boxes of configuration space for feasibility. The main limitation of the approach is that computing time increases exponentially with the number of degrees of freedom of the object to be moved.

The last problem treated was autonomous robot localization, which is particularly amenable to solution via interval analysis, because the number of parameters to be estimated is small. The method advocated in this chapter

has definite advantages over conventional numerical methods. It is not necessary to enumerate all possible associations between sensor data and landmarks, nor is it necessary to consider all possible choices of q outliers among n_s data points. As a result, combinatorial explosion is avoided. The results obtained are global, and no configuration compatible with prior information and measurement can be missed. They are extremely robust, and the estimator used can even handle a majority of outliers. Provided that the number of actual outliers is at most equal to the maximum number tolerated, the results are still guaranteed. The present computing times allow real-time implementation. The method is flexible, and additional information on the physics of the problem could readily be incorporated. One could, for instance, take into account the fact that the operational range of sonars is limited. Other types of sensors, such as rotating laser range finders (Borenstein et al., 1996; Crowley et al., 1998), as well as multi-sensor data fusion (Kam et al., 1997) should form the subject of future studies in the context of interval methods such as those advocated here.

This concludes Part III, devoted to engineering applications. Part IV will be about implementation.

Part IV

Implementation

9. Automatic Differentiation

9.1 Introduction

Interval solvers require the repeated interval evaluation of derivatives of functions. For instance, the evaluation of centred inclusion functions (page 33), Newton contractors (page 86), contractors based on parallel linearization (page 87) and the choice of the direction of bisection in SIVIAX (see (5.4), page 106) all require the computation of derivatives of functions with interval arguments.

This can be decomposed into two steps. The first one is the obtention of a punctual algorithm for evaluating the derivative of the function $\mathbf{f}$ to be differentiated. Except for academic examples, $\mathbf{f}$ has no analytical expression (it may be described by an algorithm), and evaluating its derivative is a hard task that may lead to errors if performed by hand. This is why a systematic methodology is needed, such as the one to be described in this chapter. The second step is to get a guaranteed enclosure of this derivative when intervals are involved. This can readily be achieved using the methods of Chapter 2, so we shall only consider the first step of the procedure. For more information, see Rall (1980; 1981), Corliss (1988), Bischof (1991), Evtushenko (1991), Iri (1991), Bischof et al. (1992), Corliss (1992) and Rall and Corliss (1999).

Automatic differentiation has two modes. The *forward mode* propagates derivatives in the same direction as when evaluating the function $\mathbf{f}$ to be differentiated. The *backward* (or *reverse*) *mode* propagates derivatives in the opposite direction. The principles of forward and backward differentiations are presented in Section 9.2. Section 9.3 provides recipes to build programs computing the derivative of $\mathbf{f}$ with respect to its arguments, when $\mathbf{f}$ is evaluated by a program. Two very simple illustrative examples are presented in detail in Section 9.4.

9.2 Forward and Backward Differentiations

Consider the sequence

$$\mathbf{v}^{k+1} = \boldsymbol{\phi}^{k+1}(\mathbf{v}^k), \quad k \in \{0, \ldots, \overline{k} - 1\}, \tag{9.1}$$

L. Jaulin et al., *Applied Interval Analysis*

where the dimension n_k of the vector $\mathbf{v}^k$ may depend on k, and the initial vector $\mathbf{v}^0$ is assumed to be known. Define the function

$$\mathbf{f} \triangleq \boldsymbol{\phi}^{\overline{k}} \circ \cdots \circ \boldsymbol{\phi}^1. \tag{9.2}$$

This section proposes an approach for computing the numerical value of $\frac{d\mathbf{f}}{d\mathbf{v}_0}$ for a given numerical value of $\mathbf{v}_0$. Define the functions

$$\boldsymbol{\lambda}^k \triangleq \boldsymbol{\phi}^k \circ \boldsymbol{\phi}^{k-1} \circ \cdots \circ \boldsymbol{\phi}^1,\ k \in \{1, \ldots, \overline{k}\},\ \boldsymbol{\lambda}^0 \triangleq \mathbf{I}_{n_0}, \tag{9.3}$$

$$\boldsymbol{\psi}^k \triangleq \boldsymbol{\phi}^{\overline{k}} \circ \cdots \circ \boldsymbol{\phi}^{k+1},\ k \in \{0, \ldots, \overline{k}-1\},\ \boldsymbol{\psi}^{\overline{k}} \triangleq \mathbf{I}_{n_{\overline{k}}}, \tag{9.4}$$

where $\mathbf{I}_n$ denotes the n-dimensional identity function (or matrix). The following properties hold true (see Figure 9.1):

$$\mathbf{v}^k = \boldsymbol{\lambda}^k(\mathbf{v}^0), \tag{9.5}$$

$$\mathbf{v}^{\overline{k}} = \boldsymbol{\psi}^k(\mathbf{v}^k), \tag{9.6}$$

$$\mathbf{f} = \boldsymbol{\psi}^0 = \boldsymbol{\lambda}^0 \circ \boldsymbol{\psi}^0 = \ldots = \boldsymbol{\psi}^k \circ \boldsymbol{\lambda}^k = \ldots = \boldsymbol{\psi}^{\overline{k}} \circ \boldsymbol{\lambda}^{\overline{k}} = \boldsymbol{\lambda}^{\overline{k}}, \tag{9.7}$$

$$\frac{d\mathbf{f}}{d\mathbf{v}_0} = \frac{d\boldsymbol{\psi}^0}{d\mathbf{v}_0} = \frac{d\boldsymbol{\lambda}^{\overline{k}}}{d\mathbf{v}_0}. \tag{9.8}$$

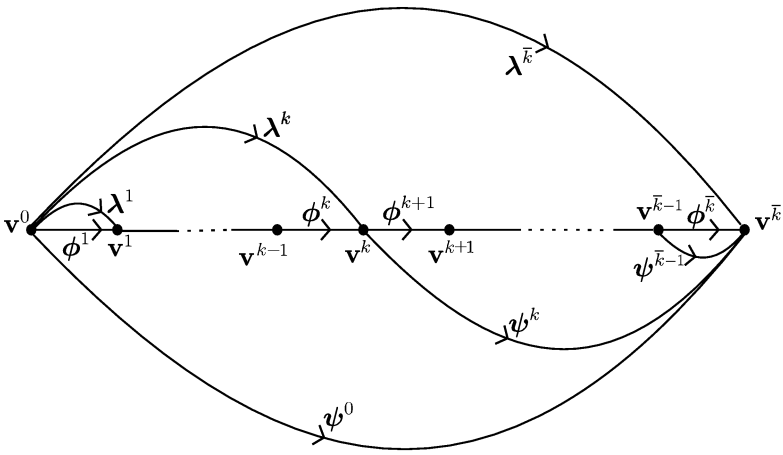


Fig. 9.1. Many ways of evaluating $\mathbf{v}^{\overline{k}} = \mathbf{f}(\mathbf{v}^0)$

9.2.1 Forward differentiation

Equation 9.3 implies that $\boldsymbol{\lambda}^{k+1} = \boldsymbol{\phi}^{k+1} \circ \boldsymbol{\lambda}^k$. Differentiate this relation to get

$$\frac{d\boldsymbol{\lambda}^{k+1}}{d\mathbf{v}^0} = \frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k} \frac{d\boldsymbol{\lambda}^k}{d\mathbf{v}^0}. \tag{9.9}$$

If

$$\mathbf{A}^k \triangleq \frac{d\boldsymbol{\lambda}^k}{d\mathbf{v}^0}, \tag{9.10}$$

then the first derivative of $\mathbf{f}$ with respect to $\mathbf{v}^0$ is given by

$$\frac{d\mathbf{f}}{d\mathbf{v}^0} = \frac{d\boldsymbol{\lambda}^{\overline{k}}}{d\mathbf{v}^0} = \mathbf{A}^{\overline{k}}. \tag{9.11}$$

$\mathbf{A}^{\overline{k}}$ can be computed recursively evaluating the sequences

$$\begin{cases} \mathbf{A}^{k+1} = \dfrac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k}\,\mathbf{A}^k, \\ \mathbf{v}^{k+1} = \boldsymbol{\phi}^{k+1}(\mathbf{v}^k), \end{cases} \quad k = 0, \ldots, \overline{k}-1, \tag{9.12}$$

where $\mathbf{A}^0 = \mathbf{I}_{n_0}$. This corresponds to the forward-differentiation algorithm FD1 of Table 9.1. When FD1 terminates, $\mathbf{A} = \mathbf{A}^{\overline{k}} = \frac{d\mathbf{f}}{d\mathbf{v}^0}$.

Table 9.1. First version of the forward-differentiation algorithm

Algorithm FD1(in: $\mathbf{v}^0$; out: $\mathbf{A}$)	
1	$\mathbf{v} := \mathbf{v}^0$;
2	$\mathbf{A} := \mathbf{I}_{n_0}$;
3	for $k := 0$ to $\overline{k} - 1$
4	$\quad \mathbf{A} := \left(d\phi^{k+1} / d\mathbf{v}^k\right)\ \mathbf{A}$;
5	$\quad \mathbf{v} := \phi^{k+1}(\mathbf{v})$.

Remark 9.1 *The order of the statements* 4 *and* 5 *in the algorithm of Table 9.1 is significant, because* $d\boldsymbol{\phi}^{k+1}/d\mathbf{v}^k$ *is a function of* $\mathbf{v}$. ■

9.2.2 Backward differentiation

Equation 9.4 implies that $\boldsymbol{\psi}^k = \boldsymbol{\psi}^{k+1} \circ \boldsymbol{\phi}^{k+1}$. Differentiate this relation to get

$$\frac{d\boldsymbol{\psi}^k}{d\mathbf{v}^k} = \frac{d\boldsymbol{\psi}^{k+1}}{d\mathbf{v}^{k+1}}\,\frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k}. \tag{9.13}$$

If

$$\mathbf{B}^k \triangleq \frac{d\boldsymbol{\psi}^k}{d\mathbf{v}^k}, \tag{9.14}$$

then the first derivative of $\mathbf{f}$ with respect to $\mathbf{v}^0$ is given by

$$\frac{d\mathbf{f}}{d\mathbf{v}^0} = \frac{d\boldsymbol{\psi}^0}{d\mathbf{v}^0} = \mathbf{B}^0. \tag{9.15}$$

$\mathbf{B}^0$ can be computed recursively evaluating the sequences

$$\begin{cases} \mathbf{v}^{k+1} = \boldsymbol{\phi}^{k+1}(\mathbf{v}^k), & k = 0, \ldots, \overline{k} - 1, \\ \mathbf{B}^k \quad = \mathbf{B}^{k+1} \dfrac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k}, & k = \overline{k} - 1, \ldots, 0, \end{cases} \tag{9.16}$$

where $\mathbf{B}^{\overline{k}} = \mathbf{I}_{n_{\overline{k}}}$. This corresponds to the backward-differentiation algorithm BD1 of Table 9.2. When BD1 terminates, $\mathbf{B} = \mathbf{B}^0 = \frac{d\boldsymbol{\phi}^0}{d\mathbf{v}^0} = \frac{d\mathbf{f}}{d\mathbf{v}^0}$.

Table 9.2. First version of the backward-differentiation algorithm

Algorithm BD1(in: $\mathbf{v}^0$; out: $\mathbf{B}$)	
1	for $k := 0$ to $\overline{k} - 1$
2	$\quad \mathbf{v}^{k+1} := \phi^{k+1}(\mathbf{v}^k)$;
3	$\mathbf{B} := \mathbf{I}_{n_{\overline{k}}}$;
4	for $k := \overline{k} - 1$ down to 0
5	$\quad \mathbf{B} := \mathbf{B}\ \left(d\phi^{k+1} / d\mathbf{v}^k\right)$.

Remark 9.2 *In (9.12) and* FD1*, the iteration counter k of both sequences increases, whereas in (9.16) and* BD1*, the iteration counter k increases when evaluating $\mathbf{v}$ and decreases when evaluating $\mathbf{B}$. As a result, all the components of the $\mathbf{v}^k$s that are needed to evaluate $d\boldsymbol{\phi}^{k+1}/d\mathbf{v}^k$ must be stored. This is not so with* FD1*, and limited storage may lead to using* FD1 *rather that* BD1 *on large-scale problems.* ■

Remark 9.3 *Both (9.12) and (9.16) compute the following product of $\overline{k}$ matrices*

$$\frac{d\boldsymbol{\phi}^{\overline{k}}}{d\mathbf{v}^{\overline{k}-1}} \cdots \frac{d\boldsymbol{\phi}^3}{d\mathbf{v}^2} \frac{d\boldsymbol{\phi}^2}{d\mathbf{v}^1} \frac{d\boldsymbol{\phi}^1}{d\mathbf{v}^0}. \tag{9.17}$$

The only difference is that (9.12) computes this product from the right to the left whereas (9.16) computes it from the left to the right. Depending on the dimension of the matrices, one or the other approach is more efficient. For instance, the product

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 2 & 3 \\ 1 & 4 & 5 \end{pmatrix} \begin{pmatrix} 4 & 7 & 1 \\ 8 & 2 & 0 \\ 1 & 4 & 5 \end{pmatrix} \begin{pmatrix} 9 & 2 & 9 \\ 2 & 5 & 3 \\ 4 & 4 & 5 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} \tag{9.18}$$

requires much less computation if evaluated from the right to the left. Computing $(\mathbf{AB})\,\mathbf{C}$ may thus require much more (or less) computation than computing $\mathbf{A}\,(\mathbf{BC})$. ■

Remark 9.4 *Since* $\boldsymbol{\psi}^k \circ \boldsymbol{\lambda}^k$ *is constant for* $k \in \{0, \ldots, \overline{k}\}$, *after differentiation,* $(d\boldsymbol{\psi}^k/d\mathbf{v}^k)(d\boldsymbol{\lambda}^k/d\mathbf{v}^0)$ *is also independent of* k. *Equations (9.10) and (9.14) then imply that*

$$\mathbf{B}^0 = \mathbf{B}^0\mathbf{A}^0 = \mathbf{B}^1\mathbf{A}^1 = \cdots = \mathbf{B}^k\mathbf{A}^k = \cdots = \mathbf{B}^{\overline{k}}\mathbf{A}^{\overline{k}} = \mathbf{A}^{\overline{k}}. \tag{9.19}$$

This property can be used to check that the implementations of FD1 *and* BD1 *are correct.* ■

9.3 Differentiation of Algorithms

An algorithm can be viewed as a special case of the sequence (9.1), where k is increased by one each time an assignment statement has been executed and where $\mathbf{v}^k$ comprises all the variables to be stored in the computer when the value of the statement counter is k. For a vast class of algorithms, the $\boldsymbol{\phi}^k$s are elementary in the sense that they modify only one variable (*i.e.*, only one component of $\mathbf{v}^k$). As a result, the possibly very large matrices $\mathbf{A}$ and $\mathbf{B}$ involved in FD1 and BD1 are sparse. This can be taken advantage of to reduce computation, as explained in this section.

Three assumptions on the $\boldsymbol{\phi}^k$s involved in (9.1) will now be made. Each of them will entail an adaptation of FD1 and BD1.

9.3.1 First assumption

Assumption 9.1 *The vectors* $\mathbf{v}^1, \ldots, \mathbf{v}^{\overline{k}-1}$ *all have the same dimension* n *and each of the functions* $\boldsymbol{\phi}^2, \ldots, \boldsymbol{\phi}^{\overline{k}-1}$ *modifies only one component of its arguments,* i.e., $\forall k \in \{1, \ldots, \overline{k}-2\}$, $\exists \mu \mid \forall i \neq \mu$, $\phi_i^{k+1}(\mathbf{v}^k) = v_i^k$. ■

Therefore, $\boldsymbol{\phi}^{k+1}(\mathbf{v}^k)$ can be expressed as

$$\boldsymbol{\phi}^{k+1}(v_1^k, \ldots, v_n^k) = \left(v_1^k, \ldots, v_{\mu-1}^k, \phi_\mu^{k+1}(v_1^k, \ldots, v_n^k), v_{\mu+1}^k, \ldots, v_n^k\right)^{\mathrm{T}}, \tag{9.20}$$

and its Jacobian matrix is

$$\frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k} = \begin{pmatrix} 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \ldots & 1 & 0 & 0 & \ldots & 0 \\ \dfrac{\partial\phi_\mu^{k+1}}{\partial v_1^k} & \cdots & \dfrac{\partial\phi_\mu^{k+1}}{\partial v_{\mu-1}^k} & \dfrac{\partial\phi_\mu^{k+1}}{\partial v_\mu^k} & \dfrac{\partial\phi_\mu^{k+1}}{\partial v_{\mu+1}^k} & \cdots & \dfrac{\partial\phi_\mu^{k+1}}{\partial v_n^k} \\ 0 & \ldots & 0 & 0 & 1 & \ldots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}. \tag{9.21}$$

In what follows, the index μ of the component of $\mathbf{v}^k$ modified by $\boldsymbol{\phi}^{k+1}$ will be denoted by $\mu(\boldsymbol{\phi}^{k+1})$. FD1 and BD1 will now be modified to take advantage of Assumption 9.1.

Forward differentiation: Let $\mathbf{a}_i^{\mathrm{T}}$ be the ith row vector of $\mathbf{A}$ in the forward-differentiation algorithm FD1. The assignment statement

$$\mathbf{A} := \frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k}\ \mathbf{A} \tag{9.22}$$

at Step 4 can be rewritten as

$$\begin{pmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_n^{\mathrm{T}} \end{pmatrix} := \frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k}\ \begin{pmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_n^{\mathrm{T}} \end{pmatrix}. \tag{9.23}$$

Because of (9.21), (9.23) reduces to

$$\mathbf{a}_\mu^{\mathrm{T}} := \sum_{j=1}^{n} \frac{\partial \phi_\mu^{k+1}}{\partial v_j^k}\ \mathbf{a}_j^{\mathrm{T}}, \tag{9.24}$$

and all other row vectors $\mathbf{a}_i^{\mathrm{T}}$ of $\mathbf{A}$ should be left unchanged. The forward-differentiation algorithm FD1 can therefore be rewritten as FD2 of Table 9.3.

Table 9.3. Second version of the forward-differentiation algorithm

Algorithm FD2(in: $\mathbf{v}^0$; out: $\mathbf{A}$)		
1	$\mathbf{v} := \mathbf{v}^0$;	
2	$\mathbf{A} := d\phi^1 / d\mathbf{v}^0$;	// Step 4 of FD1, $k = 0$
3	$\mathbf{v} := \phi^1(\mathbf{v})$;	// Step 5 of FD1, $k = 0$
4	for $k := 1$ to $\overline{k} - 2$	
5	$\quad \mu := \mu(\phi^{k+1})$;	
6	$\quad \mathbf{a}_\mu^{\mathrm{T}} := \sum_{j=1}^{n} \left(\partial \phi_\mu^{k+1} / \partial v_j^k\right)\ \mathbf{a}_j^{\mathrm{T}}$;	// see (9.24)
7	$\quad v_\mu := \phi_\mu^{k+1}(\mathbf{v})$;	
8	$\mathbf{A} := (d\phi^{\overline{k}} / d\mathbf{v}^{\overline{k}-1})\ \mathbf{A}$.	// Step 4 of FD1, $k = \overline{k} - 1$

In FD2, Step 4 of FD1 receives special treatment for $k = 0$ and $k = \overline{k} - 1$. At the end of FD2, the assignment statement $\mathbf{v} := \boldsymbol{\phi}^{\overline{k}}(\mathbf{v})$ that would be needed for FD2 to be strictly equivalent to FD1 was not inserted because it does not affect the result generated by FD2. The matrix $\mathbf{A}$ returned by FD2 corresponds to $\mathbf{A}^{\overline{k}} = d\mathbf{f}/d\mathbf{v}^0$.

Backward differentiation: Let $\mathbf{b}_j$ be the jth column vector of $\mathbf{B}$ in the backward-differentiation algorithm BD1. The assignment statement

$$\mathbf{B} := \mathbf{B}\ \frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k} \tag{9.25}$$

at Step 5 can be rewritten as

$$\left(\mathbf{b}_1\ \ldots\ \mathbf{b}_\mu\ \ldots\ \mathbf{b}_n\right) := \left(\mathbf{b}_1\ \ldots\ \mathbf{b}_\mu\ \ldots\ \mathbf{b}_n\right) \frac{d\boldsymbol{\phi}^{k+1}}{d\mathbf{v}^k}. \tag{9.26}$$

Equation 9.21 implies that (9.26) reduces to

$$\begin{aligned} \mathbf{b}_i &:= \mathbf{b}_i + \frac{\partial \phi_\mu^{k+1}}{\partial v_i^k}\ \mathbf{b}_\mu \text{ if } i \neq \mu, \\ \mathbf{b}_\mu &:= \frac{\partial \phi_\mu^{k+1}}{\partial v_\mu^k}\ \mathbf{b}_\mu. \end{aligned} \tag{9.27}$$

Since $\mathbf{b}_\mu$ is involved in the first equation of (9.27), it must be assigned *after* the $\mathbf{b}_i$s for $i \neq \mu$, so the order of the equations in (9.27) is significant. The backward-differentiation algorithm BD1 can now be rewritten as BD2 of Table 9.4. For BD2 to be strictly equivalent to BD1 (see Step 2 of BD1 for $k = \overline{k} - 1$), the statement $\mathbf{v}^{\overline{k}} := \boldsymbol{\phi}^{\overline{k}}(\mathbf{v}^{\overline{k}-1})$ should have been inserted after the loop of Step 2 of BD2, but this is unnecessary as it does not affect the result. Step 5 creates many superfluous variables that BD2 has to store. This can be avoided by using a stack as in BD2BIS of Table 9.5. The stack makes it possible at Step 8 of BD2BIS to update the only component of $\mathbf{v}^k$ that differs from that of $\mathbf{v}^{k+1}$. Recall that $\mathbf{v}^k$ is used at Steps 9 and 10 of BD2BIS for the evaluation of $\partial \phi_\mu^{k+1} / \partial v_i^k$.

Table 9.4. Second version of the backward-differentiation algorithm

Algorithm BD2(in: $\mathbf{v}^0$, out: $\mathbf{B}$)		
1	$\mathbf{v}^1 := \phi^1(\mathbf{v}^0)$;	// Step 2 of BD1, $k = 0$
2	for $k := 1$ to $\overline{k} - 2$	// Step 1 of BD1
3	$\quad \mu := \mu(\phi^{k+1})$;	
4	$\quad v_\mu^{k+1} := \phi_\mu^{k+1}(v_1^k, \ldots, v_n^k)$;	// (9.20)
5	$\quad$ for $i := 1$ to n, $i \neq \mu$, $v_i^{k+1} := v_i^k$;	
6	$\mathbf{B} := (d\phi^{\overline{k}} / d\mathbf{v}^{\overline{k}-1})$;	// Step 5 of BD1, $k = \overline{k} - 1$
7	for $k := \overline{k} - 2$ down to 1	// Step 5 of BD1
8	$\quad \mu := \mu(\phi^{k+1})$;	
9	$\quad$ for $i := 1$ to n, $i \neq \mu$,	// (9.27)
10	$\quad\quad \mathbf{b}_i := \mathbf{b}_i + \left(\partial \phi_\mu^{k+1} / \partial v_i^k\right)\ \mathbf{b}_\mu$;	
11	$\quad \mathbf{b}_\mu := \left(\partial \phi_\mu^{k+1} / \partial v_\mu^k\right)\ \mathbf{b}_\mu$;	
12	$\mathbf{B} := \mathbf{B}\ \left(d\phi^1 / d\mathbf{v}^0\right)$.	// Step 5 of BD1, $k = 0$

Table 9.5. A more efficient version of BD2

Algorithm BD2BIS(in: $\mathbf{v}^0$; out: $\mathbf{B}$)	
1	$\mathbf{v}^1 := \phi^1(\mathbf{v}^0)$;
2	for $k := 1$ to $\overline{k} - 2$
3	$\mu := \mu(\phi^{k+1})$;
4	stack v_μ;
5	$v_\mu := \phi_\mu^{k+1}(v_1, \ldots, v_n)$;
6	$\mathbf{B} := \left(d\phi^{\overline{k}} \middle/ d\mathbf{v}^{\overline{k}-1} \right)$;
7	for $k := \overline{k} - 2$ down to 1
8	$\mu := \mu(\phi^{k+1})$; unstack into v_μ;
9	for $i := 1$ to n, $i \neq \mu$, $\mathbf{b}_i := \mathbf{b}_i + \left(\partial\phi_\mu^{k+1} \middle/ \partial v_i^k \right) \mathbf{b}_\mu$;
10	$\mathbf{b}_\mu := \left(\partial\phi_\mu^{k+1} \middle/ \partial v_\mu^k \right) \mathbf{b}_\mu$;
11	$\mathbf{B} := \mathbf{B} \left(d\phi^1 \middle/ d\mathbf{v}^0 \right)$.

9.3.2 Second assumption

Assumption 9.2 *The first components of* $\mathbf{v}^1$ *correspond to the* n_0 *input variables* v_i^0 *of* $\mathbf{f}$, i.e.,

$$\mathbf{v}^1 = \boldsymbol{\phi}^1\left(\mathbf{v}^0\right) = \begin{pmatrix} \mathbf{I}_{n_0} \\ \mathbf{O}_{(n-n_0)\times n_0} \end{pmatrix} \mathbf{v}^0 = \begin{pmatrix} v_1^0 \cdots v_{n_0}^0 \; 0 \cdots 0 \end{pmatrix}^T, \tag{9.28}$$

where $\mathbf{O}_{m\times n}$ *denotes the* $m \times n$ *zero matrix and* $\mathbf{I}_n$ *denotes the* $n \times n$ *identity matrix.* ■

FD2 and BD2BIS will now be modified to take Assumption 9.2 into account.

Forward differentiation: The statement $\mathbf{A} := \frac{d\boldsymbol{\phi}^1}{d\mathbf{v}^0}$ at Step 2 of FD2 becomes

$$\begin{pmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_n^{\mathrm{T}} \end{pmatrix} := \begin{pmatrix} \mathbf{I}_{n_0} \\ \mathbf{O}_{(n-n_0)\times n_0} \end{pmatrix}, \tag{9.29}$$

so the forward-differentiation algorithm FD2 can be rewritten as FD3 of Table 9.6.

Backward differentiation: Equation 9.28 implies that

$$\begin{aligned} \mathbf{B}\frac{d\boldsymbol{\phi}^1}{d\mathbf{v}^0} &= \begin{pmatrix} \mathbf{b}_1 \ldots \mathbf{b}_{n_0} \; \mathbf{b}_{n_0+1} \ldots \mathbf{b}_n \end{pmatrix} \begin{pmatrix} \mathbf{I}_{n_0} \\ \mathbf{O}_{(n-n_0)\times n_0} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{b}_1 \ldots \mathbf{b}_{n_0} \end{pmatrix}. \end{aligned}$$

Table 9.6. Third version of the forward-differentiation algorithm

Algorithm FD3(in: $\mathbf{v}^0$; out: $\mathbf{A}$)		
1	$\mathbf{v} := (v_1^0, \ldots, v_{n_0}^0, 0, \ldots, 0)^{\mathrm{T}}$;	// Step 3 of FD2
2	for $i := 1$ to n_0, $\mathbf{a}_i^{\mathrm{T}} = (0, \ldots, 0, 1, 0, \ldots, 0)$;	// see (9.29)
3	for $i := n_0 + 1$ to $n, \mathbf{a}_i^{\mathrm{T}} = (0, \ldots, 0)$;	// see (9.29)
4	for $k := 1$ to $\overline{k} - 2$	
5	$\quad \mu := \mu(\phi^{k+1})$;	
6	$\quad \mathbf{a}_\mu^{\mathrm{T}} := \sum_{j=1}^{n} \left(\partial \phi_\mu^{k+1} / \partial v_j^k \right) \mathbf{a}_j^{\mathrm{T}}$;	
7	$\quad v_\mu := \phi_\mu^{k+1}(\mathbf{v})$;	
8	$\mathbf{A} := (d\phi^{\overline{k}} / d\mathbf{v}^{\overline{k}-1})\mathbf{A}$.	

The statement $\mathbf{B} := \mathbf{B} \, \frac{d\boldsymbol{\phi}^1}{d\mathbf{v}^0}$ at Step 11 of BD2BIS thus amounts to removing the last $n - n_0$ columns of $\mathbf{B}$. The backward-differentiation algorithm BD2BIS can then be rewritten as BD3 of Table 9.7.

Table 9.7. Third version of the backward-differentiation algorithm

Algorithm BD3(in: $\mathbf{v}^0$; out: $\mathbf{B}$)	
1	$\mathbf{v} := (v_1^0, \ldots, v_{n_0}^0, 0, \ldots, 0)^{\mathrm{T}}$;
2	for $k := 1$ to $\overline{k} - 2$
3	$\quad \mu := \mu(\phi^{k+1})$;
4	$\quad$ stack v_μ;
5	$\quad v_\mu := \phi_\mu^{k+1}(v_1, \ldots, v_n)$;
6	$\mathbf{B} := d\phi^{\overline{k}} / d\mathbf{v}^{\overline{k}-1}$;
7	for $k := \overline{k} - 2$ down to 1
8	$\quad \mu := \mu(\phi^{k+1})$; unstack into v_μ;
9	$\quad$ for $i := 1$ to n, $i \neq \mu$, $\mathbf{b}_i := \mathbf{b}_i + \left(\partial \phi_\mu^{k+1} / \partial v_i^k \right) \mathbf{b}_\mu$;
10	$\quad \mathbf{b}_\mu := \left(\partial \phi_\mu^{k+1} / \partial v_\mu^k \right) \mathbf{b}_\mu$;
11	remove the last $n - n_0$ columns of $\mathbf{B}$.

9.3.3 Third assumption

Assumption 9.3 *The last $n_{\overline{k}}$ components of $\mathbf{v}^{\overline{k}-1}$ correspond to the output variables $v_i^{\overline{k}}$ of $\mathbf{f}$,* i.e.,

$$\mathbf{v}^{\overline{k}} = \boldsymbol{\phi}^{\overline{k}} \left(\mathbf{v}^{\overline{k}-1} \right) = \left(\mathbf{O}_{n_{\overline{k}} \times (n - n_{\overline{k}})} \mid \mathbf{I}_{n_{\overline{k}}} \right) \mathbf{v}^{\overline{k}-1}. \tag{9.30}$$

■

FD3 and BD3 will now be modified to take Assumption 9.3 into account.

Forward differentiation: Equation 9.30 implies that

$$\frac{d\boldsymbol{\phi}^{\overline{k}}}{d\mathbf{v}^{\overline{k}-1}}\,\mathbf{A} = \left(\mathbf{O}_{n_{\overline{k}}\times(n-n_{\overline{k}})} \;\middle|\; \mathbf{I}_{n_{\overline{k}}}\right)\begin{pmatrix}\mathbf{a}_1^{\mathrm{T}}\\ \vdots\\ \mathbf{a}_n^{\mathrm{T}}\end{pmatrix} = \begin{pmatrix}\mathbf{a}_{n-n_{\overline{k}}+1}^{\mathrm{T}}\\ \vdots\\ \mathbf{a}_n^{\mathrm{T}}\end{pmatrix}. \tag{9.31}$$

The assignment statement $\mathbf{A} := (d\boldsymbol{\phi}^{\overline{k}}/d\mathbf{v}^{\overline{k}-1})\mathbf{A}$ at Step 8 of FD3 thus amounts to removing the first $n - n_{\overline{k}}$ rows of $\mathbf{A}$. The forward-differentiation algorithm FD3 can finally be rewritten as FD4 of Table 9.8. When FD4 terminates, $\mathbf{A}$ is equal to $d\mathbf{f}/d\mathbf{v}^0$.

Table 9.8. Fourth version of the forward-differentiation algorithm

Algorithm FD4(in: $\mathbf{v}^0$; out: $\mathbf{A}$)	
1	$\mathbf{v} := (v_1^0, \ldots, v_{n_0}^0, 0, \ldots, 0)^{\mathrm{T}}$;
2	for $i := 1$ to n_0, $\mathbf{a}_i^{\mathrm{T}} = (0, \ldots, 0, 1, 0, \ldots, 0)$;
3	for $i := n_0 + 1$ to n, $\mathbf{a}_i^{\mathrm{T}} = (0, \ldots, 0)$;
4	for $k := 1$ to $\overline{k} - 2$
5	$\quad \mu := \mu(\phi^{k+1})$;
6	$\quad \mathbf{a}_\mu^{\mathrm{T}} := \sum_{j=1}^{n} \left(\partial \phi_\mu^{k+1} / \partial v_j^k\right)\, \mathbf{a}_j^{\mathrm{T}}$;
7	$\quad v_\mu := \phi_\mu^{k+1}(\mathbf{v})$;
8	remove the first $n - n_{\overline{k}}$ rows of $\mathbf{A}$.

Backward differentiation: The assignment statement

$$\mathbf{B} := \frac{d\boldsymbol{\phi}^{\overline{k}}}{d\mathbf{v}^{\overline{k}-1}} \tag{9.32}$$

at Step 6 of BD3 can now be split into

$$\left(\mathbf{b}_1 \;\ldots\; \mathbf{b}_{n-n_{\overline{k}}}\right) := \mathbf{O}_{n_{\overline{k}}\times(n-n_{\overline{k}})} \tag{9.33}$$

and

$$\left(\mathbf{b}_{n-n_{\overline{k}}+1} \;\ldots\; \mathbf{b}_n\right) = \mathbf{I}_{n_{\overline{k}}} \tag{9.34}$$

The backward-differentiation algorithm BD3 can finally be rewritten as BD4 of Table 9.9. When BD4 terminates, $\mathbf{B}$ is equal to $d\mathbf{f}/d\mathbf{v}^0$.

Table 9.9. Fourth version of the backward-differentiation algorithm

Algorithm BD4(in: $\mathbf{v}^0$; out: $\mathbf{B}$)	
1	$\mathbf{v} := (v_1^0, \ldots, v_{n_0}^0, 0, \ldots, 0)^{\mathrm{T}}$;
2	for $k := 1$ to $\overline{k} - 2$
3	$\quad \mu := \mu(\phi^{k+1})$;
4	$\quad$ stack v_μ;
5	$\quad v_\mu := \phi_\mu^{k+1}(v_1, \ldots, v_n)$;
6	for $i := 1$ to $n - n_{\overline{k}}$, $\mathbf{b}_i = (0, \ldots, 0)^{\mathrm{T}}$;
7	for $i := n - n_{\overline{k}} + 1$ to n, $\mathbf{b}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^{\mathrm{T}}$;
8	for $k := \overline{k} - 2$ down to 1
9	$\quad \mu := \mu(\phi^{k+1})$; unstack into v_μ;
10	$\quad$ for $i := 1$ to n, $i \neq \mu$, $\mathbf{b}_i := \mathbf{b}_i + \left(\partial \phi_\mu^{k+1} / \partial v_i^k\right) \mathbf{b}_\mu$;
11	$\quad \mathbf{b}_\mu := \left(\partial \phi_\mu^{k+1} / \partial v_\mu^k\right) \mathbf{b}_\mu$;
12	remove the last $n - n_0$ columns of $\mathbf{B}$.

9.4 Examples

This section illustrates how FD4 and BD4 can be built when the function to be differentiated is given by an algorithm. Recall that the algorithm to be differentiated can be viewed as a special case of (9.1).

9.4.1 Example 1

Assume that the algorithm to be differentiated is

Algorithm f(in: u; out: y)	
1	$x := 2u$;
2	$x := 3x^2 + u$;
3	$y := x - u$.

Remark 9.5 *This first example has been chosen simple enough to allow differentiation without any knowledge about automatic differentiation. The function to be differentiated is* $f(u) = (3\,(2u)^2 + u) - u$. ■

To comply with the notation of Sections 9.2 and 9.3, rewrite this algorithm using v_1, v_2 and v_3 instead of u, x and y to get

Algorithm f(in: v_1; out: v_3)	
1	$v_2 := 2v_1$;
2	$v_2 := 3v_2^2 + v_1$;
3	$v_3 := v_2 - v_1$.

Assumption 9.1 is satisfied as no more than one variable is changed at any given step. Assumptions 9.2 and 9.3 are also satisfied since the input variable v_1 corresponds to the first component of $\mathbf{v} = (v_1, v_2, v_3)^{\mathrm{T}}$ and the output variable v_3 to its last component. The functions $\boldsymbol{\phi}^k(\mathbf{v})$ of (9.1) are given by

$$\boldsymbol{\phi}^1 = \begin{pmatrix} v_1 \\ 0 \\ 0 \end{pmatrix}, \ \boldsymbol{\phi}^2 = \begin{pmatrix} v_1 \\ 2v_1 \\ 0 \end{pmatrix}, \ \boldsymbol{\phi}^3 = \begin{pmatrix} v_1 \\ 3v_2^2 + v_1 \\ 0 \end{pmatrix}, \\ \boldsymbol{\phi}^4 = \begin{pmatrix} v_1 \\ v_2 \\ v_2 - v_1 \end{pmatrix}, \qquad \boldsymbol{\phi}^5 = v_3. \tag{9.35}$$

Moreover $\mu(\boldsymbol{\phi}^2) = 2,\ \mu(\boldsymbol{\phi}^3) = 2,\ \mu(\boldsymbol{\phi}^4) = 3$ and $\overline{k} = 5$.

Forward differentiation: FD4 translates into the algorithm of Table 9.10. Since f has only one input v_1, $\mathbf{A}$ is a column matrix and its rows $\mathbf{a}_i^{\mathrm{T}}$ are the scalar numbers a_i. Return to the initial notation for the variables u, x and y, and simplify the pseudo-code to get the algorithm of Table 9.11. By convention, the elements of $\mathbf{A}$ associated with the variables u, x and y are denoted by a_u, a_x and a_y. When the algorithm terminates, a_y is equal to df/du at the numerical value taken by u.

Table 9.10. First version of the forward-differentiation algorithm for Example 1

Algorithm $\frac{df}{dv_1}^{\mathrm{FD}}$(in: v_1; out: a_3)	
1	$\mathbf{v} := (v_1, 0, 0)^{\mathrm{T}}$;
2	$a_1 := 1; a_2 := 0;\ a_3 := 0;$ // Steps 2 and 3 of FD4
3	$a_2 := \dfrac{\partial \phi_2^2}{\partial v_1} a_1 + \dfrac{\partial \phi_2^2}{\partial v_2} a_2 + \dfrac{\partial \phi_2^2}{\partial v_3} a_3 = 2a_1;$ // $k = 1,\ \mu = 2$
4	$v_2 := 2v_1;$
5	$a_2 := \dfrac{\partial \phi_2^3}{\partial v_1} a_1 + \dfrac{\partial \phi_2^3}{\partial v_2} a_2 + \dfrac{\partial \phi_2^3}{\partial v_3} a_3 = a_1 + 6v_2 a_2;$ // $k = 2,\ \mu := 2$
6	$v_2 := 3v_2^2 + v_1;$
7	$a_3 := \dfrac{\partial \phi_3^4}{\partial v_1} a_1 + \dfrac{\partial \phi_3^4}{\partial v_2} a_2 + \dfrac{\partial \phi_3^4}{\partial v_3} a_3 = -a_1 + a_2;$ // $k = 3,\ \mu := 3$
8	$v_3 := v_2 - v_1.$ // useless

Backward differentiation: BD4 translates into the algorithm of Table 9.12. Since f has only one output v_3, $\mathbf{B}$ is a row matrix and its columns $\mathbf{b}_i$ are the scalar numbers b_i. Return to the initial notation for the variables u, x and y, and remove unnecessary statements to get the algorithm of Table 9.13.

Table 9.11. Final version of the forward-differentiation algorithm for Example 1

Algorithm $\frac{df}{du}^{\mathrm{FD}}$(in: u; out: a_y)	
1	$a_u := 1;$
2	$a_x := 2a_u;$
3	$x := 2u;$
4	$a_x := a_u + 6xa_x;$
5	$x := 3x^2 + u;$
6	$a_y := -a_u + a_x.$

Table 9.12. First version of the backward-differentiation algorithm for Example 1

Algorithm $\frac{df}{dv_1}^{\mathrm{BD}}$(in: v_1; out: b_1)		
1	$\mathbf{v} = (v_1, 0, 0)^{\mathrm{T}};$	
2	stack v_2; $v_2 := 2v_1;$	// $\mu = 2,\ k = 1$
3	stack v_2; $v_2 := 3v_2^2 + v_1;$	// $\mu = 2,\ k = 2$
4	stack v_3; $v_3 := v_2 - v_1;$	// $\mu = 3,\ k = 3$
5	$b_1 := 0;\ b_2 := 0;\ b_3 := 1;$	// Steps 6 and 7 of BD4
6	unstack into v_3	// $\mu = 3, k = 3$
7	$b_1 := b_1 + \frac{\partial \phi_3^4}{\partial v_1} b_3 = b_1 - b_3;$	
8	$b_2 := b_2 + \frac{\partial \phi_3^4}{\partial v_2} b_3 = b_2 + b_3;$	
9	$b_3 := \frac{\partial \phi_3^4}{\partial v_3} b_3 = 0;$	
10	unstack into v_2	// $\mu = 2,\ k = 2$
11	$b_1 := b_1 + \frac{\partial \phi_2^3}{\partial v_1} b_2 = b_1 + b_2;$	
12	$b_3 := b_3 + \frac{\partial \phi_2^3}{\partial v_3} b_2 = b_3;$	
13	$b_2 := \frac{\partial \phi_2^3}{\partial v_2} b_2 = 6v_2 b_2;$	
14	unstack into v_2	// $\mu = 2,\ k = 1$
15	$b_1 := b_1 + \frac{\partial \phi_2^2}{\partial v_1} b_2 = b_1 + 2b_2;$	
16	$b_3 := b_3 + \frac{\partial \phi_2^2}{\partial v_3} b_2 = b_3;$	
17	$b_2 := \frac{\partial \phi_2^2}{\partial v_2} b_2.$	

By convention, the elements of $\mathbf{B}$ associated with the variables u, x and y are denoted by b_u, b_x and b_y. When the algorithm of Table 9.13 terminates, b_u is equal to df/du at the numerical value taken by u.

Table 9.13. Final version of the backward-differentiation algorithm for Example 1

Algorithm $\frac{df}{du}^{\mathrm{BD}}$(in: u, out: b_u)	
1	$x := 2u;$
2	stack x; $x := 3x^2 + u;$
3	$y := x - u;$
4	$b_u := 0;\ b_x := 0;\ b_y := 1;$
5	$b_u := b_u - b_y;\ b_x := b_x + b_y;\ b_y := 0;$
6	unstack into x;
7	$b_u := b_u + b_x;\ b_x := 6xb_x;\ b_u := b_u + 2b_x.$

With a bit of practice, it becomes easy to code the forward and the backward differentiation algorithms directly from the code evaluating the function to be differentiated.

9.4.2 Example 2

Consider the discrete-time dynamical system (Walter and Pronzato, 1997) described by

$$y(k+1) = p_1 y(k), \text{ where } y(0) = p_2. \tag{9.36}$$

Assume that prior values $\check{y}(1), \ldots, \check{y}(n)$ are available for $y(1), \ldots, y(n)$. The parameter vector $\mathbf{p} = (p_1, p_2)^{\mathrm{T}}$ is to be estimated by minimizing the cost

$$\begin{aligned} c(\mathbf{p}) &= \textstyle\sum_{k=1}^{n} \left(y(p_1, p_2, k) - \check{y}(k)\right)^2 \\ &= \left(p_1^n p_2 - \check{y}(n)\right)^2 + \cdots + \left(p_1^2 p_2 - \check{y}(2)\right)^2 + \left(p_1 p_2 - \check{y}(1)\right)^2. \end{aligned} \tag{9.37}$$

The evaluation of the derivative of c with respect to $\mathbf{p}$ is required by most optimization algorithms. Again, an analytical expression of the gradient vector can easily be obtained by hand. To illustrate the use of automatic differentiation, $c(\mathbf{p})$ is put in the form of the following algorithm.

Algorithm c(in: p_1, p_2; out: s)
1 $s := 0;$
2 $y := p_2;$
3 for $k := 1$ to n
4 $\quad y := p_1 y;$
5 $\quad s := s + (y - \check{y}(k))^2.$

Forward differentiation: FD4 translates into the algorithm of Table 9.14. Since c has two inputs p_1 and p_2, the row vectors $\mathbf{a}_{p_1}^{\mathrm{T}}, \mathbf{a}_{p_2}^{\mathrm{T}}, \mathbf{a}_s^{\mathrm{T}}$ and $\mathbf{a}_y^{\mathrm{T}}$ all have two entries. When the algorithm of Table 9.14 terminates, $\mathbf{a}_s^{\mathrm{T}}$ is equal to $(\frac{dc}{dp_1}(\mathbf{p}), \frac{dc}{dp_2}(\mathbf{p}))$.

Table 9.14. Forward-differentiation algorithm for Example 2

Algorithm $\frac{dc}{d\mathbf{p}}^{\mathrm{FD}}$(in: p_1, p_2; out: $\mathbf{a}_s^{\mathrm{T}}$)
1 $\mathbf{a}_{p_1}^{\mathrm{T}} := (1\ 0)\,;\ \mathbf{a}_{p_2}^{\mathrm{T}} := (0\ 1)\,;\ \mathbf{a}_s^{\mathrm{T}} := (0\ 0)\,;\ \mathbf{a}_y^{\mathrm{T}} := (0\ 0)\,;$
2 $s := 0;$
3 $\mathbf{a}_y^{\mathrm{T}} := \mathbf{a}_{p_2}^{\mathrm{T}};\ y := p_2;$
4 for $k := 1$ to n
5 $\quad \mathbf{a}_y^{\mathrm{T}} := p_1 \mathbf{a}_y^{\mathrm{T}} + y\mathbf{a}_{p_1}^{\mathrm{T}};\ y := p_1 y;$
6 $\quad \mathbf{a}_s^{\mathrm{T}} := \mathbf{a}_s^{\mathrm{T}} + 2(y - \check{y}(k))\mathbf{a}_y^{\mathrm{T}}.$

Backward differentiation: BD4 translates into the algorithm of Table 9.15. Since c has only one output, the column vectors $\mathbf{b}_y, \mathbf{b}_{p_1}, \mathbf{b}_{p_2}$ and $\mathbf{b}_s$ are scalar numbers denoted by b_y, b_{p_1}, b_{p_2} and b_s. When the algorithm of Table 9.15 terminates, b_{p_1} and b_{p_2} are respectively equal to $\frac{dc}{dp_1}(\mathbf{p})$ and $\frac{dc}{dp_2}(\mathbf{p})$.

9.5 Conclusions

This chapter has presented some basic notions on the automatic differentiation of a program evaluating a function $\mathbf{f}$. When the code of this program is

Table 9.15. Backward-differentiation algorithm for Example 2

Algorithm $\frac{dc}{d\mathbf{p}}^{\mathrm{BD}}$(in: p_1, p_2; out: b_{p_1}, b_{p_2})	
1	$s := 0;$
2	$y := p_2;$
3	for $k := 1$ to n
4	stack y; $y := p_1 y;$
5	stack s; $s := s + (y - \check{y}(k))^2;$
6	$b_y := 0;\ b_{p_1} := 0;\ b_s := 1;$
7	for $k := n$ down to 1
8	unstack into s;
9	$b_y := b_y + 2(y - \check{y}(k))b_s;$
11	unstack into y;
12	$b_{p_1} := b_{p_1} + y b_y;\ b_y := p_1 b_y;$
14	$b_{p_2} := b_y.$

not too long, it is easy to build the differentiating program by hand using the methodology illustrated in Section 9.4 and simplifying the resulting code as much as possible. Of course, a computer implementation of the methodology is preferable when complex programs must be handled.

Different criteria should be taken into account for the choice between forward and backward differentiation. For an *easy implementation*, forward differentiation may be preferred because it can be implemented using operator overloading (Hammer et al., 1995). To *save space*, forward differentiation may be preferred as it does not require storing values taken by the variables in the program to be differentiated. To *save run time*, the choice between forward and backward differentiation should depend on the number of inputs n_{u} and the number of outputs n_{y} of the code computing $\mathbf{f}$. Recall that each variable v_i of the algorithm for $\mathbf{f}$ corresponds to n_{u} variables (through the row vector $\mathbf{a}_i^{\mathrm{T}}$) for forward differentiation and to n_{y} variables (through the column vector $\mathbf{b}_i$) for backward differentiation. In Section 9.4.2, for instance, the algorithm to be differentiated has two inputs and one output, so $\mathbf{a}_i$ is two-dimensional whereas $\mathbf{b}_i$ is scalar. Now, the vectors $\mathbf{a}_i$ and $\mathbf{b}_i$ have to be stored and updated at each iteration. If the number of inputs is larger than the number of outputs, backward differentiation will generally be quicker and in the opposite case forward differentiation should be preferred.

10. Guaranteed Computation with Floating-point Numbers

10.1 Introduction

One of the main features of interval analysis is its ability to provide boxes guaranteed to contain the image of a given box by a function. This *containment* property has to be preserved by computer implementation. The intervals computed using a finite-precision representation of real numbers should therefore always contain those that would be obtained with an infinite precision. A trade-off should moreover be found between execution time and accuracy of interval evaluation.

The first part of this brief chapter is dedicated to the consequences of the floating-point representation used for real numbers on the implementation of interval software. Section 10.2 gives some indications on the IEEE 754 standard for binary floating-point arithmetic. This standard is complied with by the processors equipping most of today's personal computers and workstations, and has contributed much to the normalization of a previously rather chaotic situation (Severance, 1998). We shall see that it includes features facilitating the implementation of interval computation, such as directed rounding, but leaves some problems open. Section 10.3 suggests an implementation of interval computation, taking into account empty intervals and intervals with infinite bounds. Finally, Section 10.4 gives pointers to presently available software.

10.2 Floating-point Numbers and IEEE 754

Among the representations proposed to approximate real numbers on computers (see Swartzlander and Alexopoulos, 1975, for the *sign/logarithm* representation or Matula and Kornerup, 1985, for the *slash number system*), the floating-point representation is the most widely used. A floating-point number can be written as

$$\pm d_0.d_1d_2\ldots d_{p-1}\times\beta^e, \tag{10.1}$$

where $d_0.d_1d_2\ldots d_{p-1}$ is the p-digit *significand*, β is the *base* and e is the *unbiased* signed *exponent*, constrained to belong to some interval $[e_{\min}, e_{\max}]$.

L. Jaulin et al., *Applied Interval Analysis*

The reasons for calling the exponent unbiased will become clear shortly. The notation (10.1) corresponds to the real number

$$\pm \left(d_0\beta^0 + d_1\beta^{-1} + d_2\beta^{-2} + \cdots + d_{p-1}\beta^{-(p-1)} \right) \times \beta^e, \tag{10.2}$$

with $(0 \leqslant d_i < \beta)$.

Example 10.1 *The real number* 0.5 *is implicitly expressed in base* 10. *For* $\beta = 10$ *and* $p = 4$, *its floating-point representation is* 5.000×10^{-1}. *For* $\beta = 2$ *and* $p = 6$, *it becomes* 1.00000×2^{-1}. ■

This representation, however, is not uniquely defined. For instance,

$$2.500 \times 10^{-1} = 0.025 \times 10^{1}. \tag{10.3}$$

This complicates the design of some algorithms, such as those requiring the comparison of numbers. To ensure uniqueness, the leading digit d_0 in (10.1) is forced to be non-zero. The resulting representation is said to be *normalized*.

Remark 10.1 *When* $\beta = 2$, *as usual on computers,* d_0 *is either* 0 *or* 1, *thus in a normalized binary representation* d_0 *is always equal to* 1. ■

10.2.1 Representation

The IEEE 754 standard includes a norm for the binary representation of floating-point numbers ($\beta = 2$). More details may be found in IEEE Computer Society (1985), Goldberg (1991) and Kahan (1996). Four floating-point formats have been defined, namely *single*, *double*, *single extended* and *double extended*. The first two are the most widely used, and correspond to `float` and `double` in C or C++. Each format is characterized by the width p of its significand and its interval of allowed exponents; see Table 10.1.

Table 10.1. IEEE 754 floating-point formats

Format	p	$e_{\min}$	$e_{\max}$	Exponent width	Format width
single	24	-126	127	8	32
single ext.	32	$\leqslant -1022$	$\geqslant 1023$	$\geqslant 11$	$\geqslant 43$
double	53	-1022	1023	11	64
double ext.	64	$\leqslant -16382$	$\geqslant 16383$	$\geqslant 15$	$\geqslant 79$

For both extended formats, only a lower bound of the number of digits of the exponent is specified. This number of digits may thus depend on the implementation. Note that $e_{\max}$ is greater than $|e_{\min}|$. This is to avoid overflow when computing $1/x$ if x is a non-zero floating-point number with the

smallest absolute value in the format considered (e.g., $\pm 1.000\ldots0 \times 2^{-126}$ in the single format). Underflow, however, is not prevented, but this was regarded by the authors of the standard as less of a problem than overflow. The exponents $e_{\min} - 1$ and $e_{\max} + 1$ have been reserved for the coding of special numbers (see Section 10.2.3).

Representing a floating-point number also requires coding the signs of its significand and exponent. A specific bit is used for coding the sign of the significand, with 0 for + and 1 for −. The sign of the exponent is coded using a shift of the exponent by a prespecified offset (e.g., +127 for the single format). The result is called a *biased exponent*. Provided that floating-point numbers are stored with the exponent followed by the significand, this approach has the advantage of preserving the ordering of the floating-point numbers in their representations.

Since a bit is required to code the sign of the significand, 33 bits would seem be needed for the single format. To stay within 32 bits, it is possible to use the fact that the leading digit d_0 in (10.1) is always equal to one for a normalized binary representation and therefore need not be stored. The leading digit is then hidden in the significand. Table 10.2 presents some bit patterns. The hidden digit is put between parentheses.

Table 10.2. Decimal representation and bit pattern of floating-point numbers

Decimal	Binary representation			Interpretation
represent.	Sign	Exponent	Significand	of binary representation
1	0	01111111	(1) 000 . . . 000	$(-1)^0 \times 1.000 \times 2^{127-127}$
−2	1	10000000	(1) 000 . . . 000	$(-1)^1 \times 1.000 \times 2^{128-127}$
16.5	0	10000011	(1)000010...0	$(-1)^0 \times 1.00001 \times 2^{131-127}$ $= 1 \times 2^4 + 1 \times 2^{-1}$

10.2.2 Rounding

One of the consequences of describing real numbers with a finite number of bits is that they usually cannot be represented exactly. It is thus necessary to provide rounding mechanisms to get representable numbers. Even the result of a computation based on floating-point numbers must often be rounded to get a floating-point number.

Rounding error is measured in *ulp* (*units at the last place*). The difference between the actual number and its floating-point approximation is thus expressed using as the unit the least significant bit in the floating-point format considered. For example, if $\beta = 2$ and $p = 4$, then representing 1.001 by 1.000

results in an error of 1 ulp; representing 1.10001 by 1.100 corresponds to an error of $0.01 = 1.0 \times 2^{-2} = 0.25$ ulp. More generally, if a real number z is approximated by a floating-point number $\pm d_0.d_1 d_2 \ldots d_{p-1} \times \beta^e$, the error in ulp is given by

$$\left| \pm d_0.d_1 d_2 \ldots d_{p-1} - \frac{z}{\beta^e} \right| \beta^{p-1}. \tag{10.4}$$

When the error is less than 0.5 ulp, the floating-point number is said to be *correctly rounded* (Kahan, 1996).

Four rounding modes are specified in the standard. The default rounding mode is to the nearest floating-point number. The results of the four basic arithmetical operations, the square root, the *remainder of* operator % (if $a = n \times b + r$, with $(a, b) \in \mathbb{R}^2$ and $n \in \mathbb{Z}$, then the remainder $r = a\%b$ satisfies $|r| < |b|$) and the conversion from integer to floating-point numbers are required to be correctly rounded in the default rounding mode. The other three rounding modes are towards 0, towards $+\infty$ and towards $-\infty$. The last two rounding modes are particularly useful for the implementation of guaranteed computation, as will be seen in Section 10.3. Figure 10.1 illustrates these four rounding modes. The bold vertical segments correspond to numbers that are exactly representable in the floating-point system considered. The real numbers corresponding to two consecutive bold segments differ by one ulp. The real number x is not representable exactly and the four rounding modes lead to two possible results.

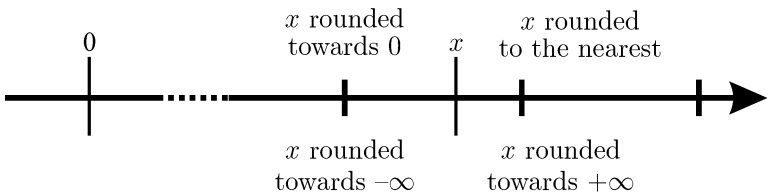


Fig. 10.1. The four rounding modes specified by IEEE 754

The results of transcendental functions are presently not required to be correctly rounded, because of what is known as the *table maker's dilemma* (Goldberg, 1991; Daumas et al., 1995; Muller, 1997; Lefèvre et al., 1998). Consider a floating-point system with base 2 and significand of width p. Assume, for example, that the logarithm of a floating-point number a has been evaluated with a precision of $p + 2$ digits:

$$\log a \simeq \underbrace{\mathbf{1.x \ldots x01}}_{p \text{ digits}} 11. \tag{10.5}$$

The **x**s in (10.5) represent digits that play no role in the dilemma. When rounding is towards $-\infty$, $\log a$ should be rounded to $\mathbf{1.x\ldots x01}$ if a computation with more digits would reveal that $\log a = \mathbf{1.x\ldots x01}110\ldots$ and to $\mathbf{1.x\ldots x10}$ if it appears that $\log a = \mathbf{1.x\ldots x10}010\ldots$ Assume now that the evaluation is made with increasing precision, and that its result is $\mathbf{1.x\ldots x01}111$ and then $\mathbf{1.x\ldots x01}1111$, and then... As logarithm is a transcendental function, it is impossible to know in advance when the precision will be sufficient, hence the dilemma for the choice of the result of the rounding. So, for some very special numbers, computation would need to be made with a very high precision (and thus at a very high computational cost) for the dilemma to be solved, and this is not required by IEEE 754.

This dilemma has an important consequence on the difficulty of evaluating tight intervals guaranteed to contain the values taken by transcendental functions, as will be seen in Section 10.3.3.

10.2.3 Special quantities

Special quantities have been defined in IEEE 754 to allow correct handling of exceptional situations such as division by zero or an attempt to evaluate the square root of a negative number. The codes of these special quantities are reported in Table 10.3.

Table 10.3. Special quantities and their coding according to IEEE 754

Number	Sign	Exponent	Significand
$\pm\infty$	$\pm$	$e_{\max}+1$	$1.00\ldots0$
±0	$\pm$	$e_{\min}-1$	$1.00\ldots0$
denormalized	$\pm$	$e_{\min}-1$	$\neq 1.00\ldots0$
NaN	any	$e_{\max}+1$	$\neq 1.00\ldots0$

Infinite quantities. The quantities $\pm\infty$ provide a way to continue computation after an overflow. Propagating $+\infty$ is much safer than representing it by a very large finite number, as illustrated by the following example.

Example 10.2 *When* $\log(\exp(1000))$ *is evaluated using single-precision floating-point numbers,* $\exp(1000)$ *overflows. An* IEEE 754*-compliant processor would then code it as* $+\infty$*, and the final result would also be* $+\infty$*, indicating an overflow in the computation. If a very large number was used instead of* $+\infty$*, e.g.,* 10^{38}*, then the final result of the computation would be* $\log\left(10^{38}\right) \approx 87$*, which looks reasonable but is of course meaningless as* $\log(\exp(1000)) = 1000$. ■

Signed zeros. Since the leading digit d_0 in the significand of a normalized binary floating-point number is equal to 1, the representation of 0 is problematic. By convention, the *signed zeros* $+0 = 1.0 \times \beta^{e_{\min}-1}$ and $-0 = -1.0 \times \beta^{e_{\min}-1}$ are used. The standard imposes that $+0 = -0$ is considered as true; otherwise, as simple a test as `(x==0)` would have an unpredictable behaviour. It should however be kept in mind that these two numbers are distinct. This is of particular importance for interval divisions. Thus, $[4,5] / [+0,2] = [2,+\infty]$ whereas $[4,5] / [-0,2] = [-\infty,+\infty]$. Although IEEE 754 recommends the implementation of a `CopySign()` function that reveals the sign of 0, few compilers offer this function (it is provided in Borland's C++ Builder, but was not available in its predecessors).

Denormalized numbers. Such numbers are introduced to allow the representation of smaller numbers than possible with the normalized representation (at the cost of a reduction in the number of significant digits). For denormalized numbers, the leading digit is assumed to be 0.

***NaN*s.** As indicated by Table 10.3 there are many *NaN*s. Each of them stands for *Not a Number*, but should rather be interpreted as *Not any Number* (Kahan, 1996) when it is used to indicate that a result is invalid. Any invalid operation has thus been requested to yield a *NaN*. Examples of invalid operations are $\sqrt{x}$ with $x < 0$, $0 \times \infty$, $0.0/0.0$, ∞/∞, $(\pm\infty) - (\pm\infty)$ when the signs in the parentheses are the same, $y\%0.0$ and $\infty\%y$, with $y \in \mathbb{R}$. Any arithmetic operation involving *NaN*s produces a *NaN*, unless the result obtained when replacing each of these *NaN*s by an arbitrary number (which may be finite or infinite) does not depend on the value of the replacement.

The concept of *NaN* was introduced for numerical computation, and is not widely used yet in mathematical analysis. It provides an elegant way to cope with exceptional situations that may be encountered, for example, during root search or global optimization. Usually an initial search domain is required. If the function to be evaluated is not defined over some parts of this domain, then a badly designed search algorithm might stall during the exploration of these parts. With *NaN*s, it becomes possible to detect *a posteriori* that the function was not defined at some points of the search domain without terminating search prematurely. Another possible technique is to throw an exception whenever the function turns out not to be defined, see Section 11.13.

Even if this is not specified in IEEE 754, other real functions should sometimes also return a *NaN*, such as $\log(x)$, with $x \leqslant 0$, $\sin(\infty)$, $\arcsin(y)$ with $|y| > 1$, etc. This is so, for example, for Intel's Pentium.

10.3 Intervals and IEEE 754

IEEE 754 does not specify how interval computation should be implemented, but makes it easier to satisfy requirements such as containment or portability.

This section will summarize proposals presented in Chiriaev and Walster (1998), Walster (1998) and Lerch and Wolff von Gudenberg (2000), for the implementation of interval arithmetic on IEEE 754-compliant computers. These proposals have been followed by SUN for the development of FORTRAN and C++ compilers providing a native interval type, which is treated on the same level as the real type. Machine representation of intervals will first be considered, before describing a closed interval arithmetic satisfying the containment requirement. Finally, we shall see that it is possible to obtain guaranteed inclusion functions even for transcendental functions.

10.3.1 Machine intervals

Let $\mathcal{R}$ be the set of all *finite machine numbers*, *i.e.*, of all real numbers that are representable in a given floating-point format. Denote by $\aleph$ the largest element of $\mathcal{R}$. We have

$$-\infty < -\aleph < \cdots < 0 < \cdots < \aleph < \infty, \tag{10.6}$$

where 0 represents +0 and −0 indifferently, as in the rest of this section.

Let $\overline{\mathcal{R}}$ be the set of all machine numbers, $\overline{\mathcal{R}} = \mathcal{R} \cup \{-\infty, +\infty\}$, where $-\infty$ and $+\infty$ are the standard IEEE 754 infinite quantities. Any element of $\mathbb{R}$ can be mapped without ambiguity into one element of $\overline{\mathcal{R}}$, using either rounding towards $+\infty$ denoted by $\uparrow(.)$ or rounding towards $-\infty$ denoted by $\downarrow(.)$. These rounding modes both preserve the ordering $\leqslant$ of $\mathbb{R}$ and are idempotent, *i.e.*, $\uparrow(\uparrow(x)) = \uparrow(x)$ and $\downarrow(\downarrow(x)) = \downarrow(x)$. They are provided by IEEE 754, as described in Section 10.2.2. For real numbers with an absolute value lower than $\aleph$, the rounding error is guaranteed to be less than one ulp in both cases.

Let $\mathcal{IR}$ be the set of all *finite machine intervals*

$$\mathcal{IR} = \{[a, b] \mid a \in \mathcal{R},\ b \in \mathcal{R},\ a \leqslant b\}, \tag{10.7}$$

and $\overline{\mathcal{IR}}$ be the set of all *machine intervals*

$$\begin{aligned}\overline{\mathcal{IR}} = \mathcal{IR} &\cup \{[a, +\infty] \mid a \in \mathcal{R}\} \cup \{[-\infty, b] \mid b \in \mathcal{R}\} \\ &\cup [-\infty, +\infty] \cup \left[NaN_{\varnothing}, NaN_{\varnothing}\right],\end{aligned} \tag{10.8}$$

where the interval $\left[NaN_{\varnothing}, NaN_{\varnothing}\right]$ represents the empty set $\varnothing$, obtained, for example, by intersecting two disjoint intervals (see Section 2.3). Note that $[-\infty, -\infty]$ and $[+\infty, +\infty]$ do not belong to $\overline{\mathcal{IR}}$. The intervals $[-\infty, -\aleph]$ and $[\aleph, +\infty]$ have a clearer set-theoretic interpretation and facilitate implementation, because operations such as $-\infty + \infty$ or $+\infty - \infty$ are avoided during interval addition or subtraction.

Any arithmetical operation involving $\varnothing$ or any elementary function applied to $\varnothing$ returns $\varnothing$. $NaN_{\varnothing}$ is a *NaN* only used for the coding of the empty set, which cannot be obtained otherwise. This coding takes advantage of the fact that the IEEE 754 *NaN*s are absorbing, *i.e.*, that any computation

involving *NaN*s returns *NaN*s. Using $\left[NaN_{\varnothing},NaN_{\varnothing}\right]$ greatly facilitates the implementation of interval algorithms; often, $\left[NaN_{\varnothing},NaN_{\varnothing}\right]$ can be treated like any other standard interval, and no check for emptiness is required.

$\uparrow(.)$ and $\downarrow(.)$ can be used to map any interval of $\mathbb{IR}$ into one element of $\overline{\mathcal{IR}}$ without ambiguity. This is done by the outward-rounding function $\updownarrow(.)$, defined as follows:

$$\forall [x] = [\underline{x}, \overline{x}] \in \mathbb{IR}, \quad \updownarrow([x]) = [\downarrow(\underline{x}), \uparrow(\overline{x})] \in \overline{\mathcal{IR}}. \tag{10.9}$$

Intervals whose lower bound is greater than $\aleph$ are thus represented by $[\aleph, +\infty]$; similarly, intervals whose upper bound is smaller than $-\aleph$ are represented by $[-\infty, -\aleph]$.

Remark 10.2 *It is possible to implement outward rounding using only one type of directed rounding, as*

$$\updownarrow([x]) = [\downarrow(\underline{x}), \uparrow(\overline{x})] = [\downarrow(\underline{x}), -\downarrow(-\overline{x})] = [-\uparrow(-\underline{x}), \uparrow(\overline{x})]. \tag{10.10}$$

This property may be used to reduce the number of time-consuming switchings of the rounding mode of the processor (Knofel, 1993). ■

10.3.2 Closed interval arithmetic

For any operator $\diamond$ in $\{+, -, *, /\}$, $[x]$ in $\overline{\mathcal{IR}}$ and $[y]$ in $\overline{\mathcal{IR}}$,

$$\updownarrow([x] \diamond [y]) \in \overline{\mathcal{IR}}, \tag{10.11}$$

and

$$[x] \diamond [y] \subset \updownarrow([x] \diamond [y]). \tag{10.12}$$

The operator $\updownarrow(.)$ on $\overline{\mathcal{IR}}$ thus makes it possible to define an arithmetic on machine intervals that satisfies the containment requirement. This arithmetic is also closed, as all arithmetic operations that are not defined when interval operands contain 0 or infinite quantities are extended applying the rules of computation with infinite quantities (Kahan, 1968; Chiriaev and Walster, 1998).

Example 10.3 *With this arithmetic,*

$$\begin{aligned}
&\updownarrow([2,3] * [5,7]) = [10,21],\\
&\updownarrow(2 * [500, \aleph]) = [1000, +\infty],\\
&\updownarrow([0,3] * [10, +\infty]) = [-\infty, +\infty],\\
&\updownarrow([-\infty, -\aleph] + [\aleph, +\infty]) = [-\infty, +\infty],\\
&\updownarrow([1,3] / [0,3]) = [-\infty, +\infty],\\
&\updownarrow([-5,2] / [0,0]) = [-\infty, +\infty],\\
&\updownarrow([0,0] / [0,0]) = [-\infty, +\infty].
\end{aligned} \tag{10.13}$$

■

When only arithmetical interval operations are involved, IEEE 754 guarantees an error of less than one ulp in the computation of the bounds. Thus, maximum precision is obtained at maximum speed, and no trade-off between speed and precision has to be found.

Example 10.4 *The result of the division of* $x = 5$ *by* $y = 50$ *is* 0.1*, which is not exactly representable by a floating-point number in base* 2*. Directed rounding makes it possible to compute an interval* $[z]$ *guaranteed to contain* x/y*, with*

$$\underline{z} = 1.10011001100110011001100\mathbf{0} \times 2^{-4}$$

and

$$\overline{z} = 1.10011001100110011001100\mathbf{1} \times 2^{-4}$$

if single precision is used. The bounds of $[z]$ *differ only by one ulp.* ■

A pseudo-code for interval addition is given in Table 10.4.

Table 10.4. Interval addition

Algorithm operator+ (in: $[a]$, $[b]$; out: $[r]$)	
1	store_rounding_mode();
2	$[r] := \left[\downarrow(\underline{a} + \underline{b}), \uparrow(\overline{a} + \overline{b})\right]$;
3	restore_rounding_mode().

Three points have to be noticed. First, there is no need to check whether $[a]$ or $[b]$ is the empty set $\varnothing$, as the representation of $\varnothing$ by $[NaN_{\varnothing}, NaN_{\varnothing}]$ guarantees that the result will be $[NaN_{\varnothing}, NaN_{\varnothing}]$ as soon as any of the operands is empty. Second, the current rounding mode is stored before interval addition and restored after, in order to make evaluations on real numbers with maximum accuracy. Finally, the rounding mode is switched twice during the evaluation of $[r]$ but could be switched once only, see Remark 10.2.

10.3.3 Handling elementary functions

The main problem with elementary functions is that some of them, such as log or tan, are not defined on the entire real line. The actions to be taken when evaluating, for example, $\log([-3, 1])$ is one of the major points of disagreement between intervalists. In this book, we have chosen to return the range of a function over a given interval, even if this function is not defined on

part of this interval, see (2.30). If a function is not defined on any part of an interval argument, then the range returned is the empty set, as in PROLOG 4.

An implementation of the square root (sqrt) satisfying the containment constraint is easily achieved with a precision of one ulp on each bound, as sqrt is requested by IEEE 754 to be correctly rounded, see Table 10.5.

Table 10.5. Interval square root

Algorithm sqrt(in: $[a]$; out: $[r]$)	
1	$[b] := [a] \cap [0, \infty]$;
2	if $([b] = \varnothing)$ //optional
3	$[r] := \varnothing$; return;
4	store_rounding_mode();
5	$[r] := \left[\downarrow\left(\text{sqrt}(\underline{b})\right), \uparrow\left(\text{sqrt}(\overline{b})\right)\right]$;
6	restore_rounding_mode().

Testing $[b]$ for emptiness is optional on any computer on which any elementary function applied on a *NaN* returns a *NaN*, as required by IEEE 754. Here, the two switchings of the rounding mode are necessary.

Example 10.5 *With this implementation,*

$$\updownarrow\left(\sqrt{[5,8]}\right) = \left[\downarrow\left(\sqrt{5}\right), \uparrow\left(\sqrt{8}\right)\right], \tag{10.14}$$

$$\updownarrow\left(\sqrt{[-3,7]}\right) = \left[0, \uparrow\left(\sqrt{7}\right)\right], \tag{10.15}$$

$$\updownarrow\left(\sqrt{[15,+\infty]}\right) = \left[\downarrow\left(\sqrt{15}\right), +\infty\right], \tag{10.16}$$

$$\updownarrow\left(\sqrt{[-5,-2]}\right) = \varnothing. \tag{10.17}$$

■

For transcendental functions, due to the table maker's dilemma, it is not possible to guarantee the enclosure of the correct mathematical result by simply switching to the appropriate rounding mode (see Section 10.2.2). A solution is to expand the interval computed so that it is guaranteed to contain the actual mathematical result. The number of ulps by which the result must be expanded depends on the quality of the implementation of the transcendental functions. Figure 10.2 illustrates a situation where computations compliant with IEEE 754 produce an interval $[y]$ that does not contain $\log([x])$. The problem is then solved by expanding $[y]$ by one ulp for each bound. When dealing with transcendental functions, a trade-off has to be found between speed of computation and accuracy.

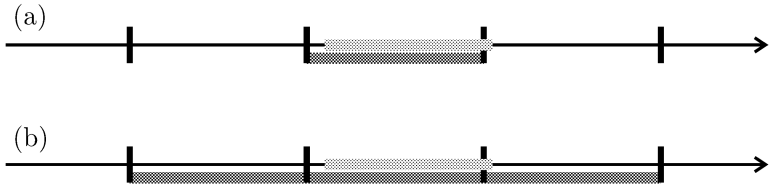


Fig. 10.2. Evaluation of the image of $[x]$ by the log function; the actual image is in light grey; (a) image as approximated according to IEEE 754 (in dark grey); because of the table maker's dilemma, the approximation does not contain the actual image; (b) guaranteed approximation of the image, obtained by expanding the previous approximation by one ulp for each bound (in dark grey)

10.3.4 Improvements

The implementation that has just been described corresponds to the *simple* extended interval system defined in Walster (1998). For arithmetic operations, only what is provided by IEEE 754 is needed, as exemplified by the implementation of interval addition described in Section 10.3.2.

More sophisticated interval systems have been presented in Walster (1998), such as the *full* extended interval system, which provides better control when underflow occurs in operations involving intervals. This reduces the number of situations where the entire real line is returned as a result. The main difference with the simple system is the distinction between the signed zeros; in the simple extended system $[+0, 5]$ is considered equal to $[-0, 5]$, which is no longer the case with the full system. The next example illustrates the consequences of such a distinction.

Example 10.6 *Take* $[a] = [4, 5]$ *and* $[b] = [0.5, \aleph]$. *When evaluated with a library implementing the simple extended system,*

$$\updownarrow([a] \,/\, \updownarrow(1/\,[b])) = \updownarrow([4, 5] \,/\, [0, 2]) = [-\infty, +\infty]\,. \tag{10.18}$$

The same quantity evaluated with a library implementing the full extended system would be

$$\updownarrow([a] \,/\, \updownarrow(1/\,[b])) = \updownarrow([4, 5] \,/\, [+0, 2]) = [2, +\infty]\,. \tag{10.19}$$

Such a result requires specifying that $\downarrow(1/\aleph) = +0$ *and detecting the sign of* 0 *for the second division.* ■

Implementing the full extended system is a complicated task, and only simple extended systems have been released so far.

10.4 Interval Resources

The purpose of this section is to give pointers to available software for interval computation. A regularly updated list of such pointers may be found at

`http://www.cs.utep.edu/interval-comp/main.html`

The reader is also invited to consult Kearfott (1996b) for a historical presentation of the development of interval resources.

A series of interval software has been developed since the 1960s at Karlsruhe University, and the present XSC family of libraries (for *eXtension for Scientific Computing*) has benefitted from the experience accumulated. Thus, PASCAL-XSC (Klatte et al., 1992), FORTRAN-XSC and C-XSC (Klatte et al., 1993; Hammer et al., 1995) extend PASCAL, FORTRAN and C++ to intervals. A common characteristic of these tools is their control of one of the main sources of numerical errors by the implementation of an accurate dot product (Kulisch and Miranker, 1981; Wolff von Gudenberg, 1994). Standard algorithms are provided for such tasks as the global optimization of possibly multi-modal functions or the solution of sets of possibly non-linear equations. These libraries are available for most platforms and provide a software support of the IEEE 754 standard when this standard is not fully implemented in hardware. Details and portions of the source code may be found at

`http://www.xsc.de/`

A new version of C-XSC, which complies with the new ISO-C++ standard, is under development under the name of C-XSC++. A beta version is available at

`http://www.uni-karlsruhe.de/~uad3/cxsc/`

The *Basic Interval Arithmetic Subroutines Library* (Corliss, 1991) could serve as a template for any interval library, independently of the language in which this library would be coded. Several tools have been developed based on this work; the most popular is probably the PROFIL/BIAS library, written in C++ (Knüppel, 1994). This library is freely available for most platforms, but less complete than C-XSC. It may be downloaded from

`ftp://ti3sun.ti3.tu-harburg.de/pub/profil/`

More details may be found in Chapter 11.

INTLIB (for *INTerval LIBrary*) was developed by Kearfott in FORTRAN, especially for the solution of sets of non-linear equations (Kearfott et al., 1992; Kearfott, 1996b). This library and other interesting tools may be downloaded from

`http://interval.louisiana.edu/kearfott.html`

The very recent release by SUN of the FORTE FORTRAN/HPC and C++ compilers is particularly noteworthy. These compilers are the first to offer a native interval type, allowing the coding of algorithms based on interval computations without the need for any additional library. A closed interval arithmetic system is also implemented (see Section 10.3). For the time being, these compilers only run on computers equipped with the SOLARIS operating system. Limited-time trial versions of both of them may be downloaded from

`http://www.sun.com/forte/index.html`

see also the CD-ROM provided with this book.

Another C++ interval package providing a closed interval arithmetic system is `fi_lib++` (Lerch and Wolff von Gudenberg, 2000). This library should become freely available at

`http://www.math.uni-wuppertal.de/org/WRST/xsc/download.html`

All these products have benefitted considerably from object-oriented programming and operator overloading, which allow intervals and interval vectors to be manipulated about as simply as other standard data types.

To solve relatively simple problems for which compilation can be avoided, interpreted languages can be used. Interval MAPLE add-ons have been developed by Connell and Corless (1993). For MATLAB, Zemke developed the tutorial tool B4M (BIAS for MATLAB 5), and Rump is at the origin of the much more ambitious INTLAB library (Rump, 1999, 2001), where special attention has been paid to the computation of transcendental functions. INTLAB runs under WINDOWS, UNIX and LINUX. INTLAB and B4M may be downloaded from

`http://www.ti3.tu-harburg.de/english/index.html`

10.5 Conclusions

The IEEE 754 standard for binary floating-point representation provides a framework for a guaranteed implementation of the algorithms presented in this book. Of special importance is rounding control, which allows the limitation of precision resulting from a floating-point representation of real numbers to be taken into account in such a way that the results are still guaranteed. Although the first compilers and thus the first libraries for interval computation complied with only part of the specifications of this standard, the trend is towards stricter compliance.

The release by SUN of the FORTE series of compilers now makes it possible to cope with intervals directly. One of the main present challenges is the design of hardware for the same purpose. Efforts have been put into the design of dedicated coprocessors (Wolff von Gudenberg, 1996; Schulte and Swartzlander, Jr., 2000) and into the suggestion of modifications of the architecture of processors (Stine and Schulte, 1998a, 1998b; Kolla et al., 1999). Unfortunately, most of the results presently available are confined to the specification level and to the realization of prototypes.

11. Do It Yourself

11.1 Introduction

The first purpose of this chapter will be to show how YOUR LIBRARY, a basic library for interval analysis, can be implemented in C++. As will soon become apparent, such an implementation is a lot of work. One may thus wonder if one would not be better off using a readily available library, and the second purpose of this chapter will be to explain how this can be done. We have chosen the PROFIL/BIAS library, because it is licensed free of charge, and runs on a large choice of platforms. The time spent building YOUR LIBRARY will facilitate the understanding of the source code of PROFIL/BIAS, as they share their basic syntax. The last purpose of this chapter will be to give some details on how the algorithms described in the rest of the book may be implemented, using either YOUR LIBRARY or PROFIL/BIAS.

The material is organized as follows. First the minimum knowledge about C++ required to start is summarized. The basic objects of YOUR LIBRARY, namely intervals, interval vectors and matrices, are then introduced. For each of them, the presentation is similar. We start by recalling specific notions of C++ if needed, before building up the object structure step by step. Typical problems of implementation are then illustrated in detail, which should allow completion of YOUR LIBRARY as suggested by the exercises. We then explain how these objects are implemented and used in PROFIL/BIAS. Exercises illustrating the implementation of some of the algorithms described earlier complete the presentation of these objects. Subpavings, which are not part of PROFIL/BIAS, are presented in much more detail. The chapter ends with a section on error handling.

Readers not interested in library implementation may skip Sections 11.3, 11.6 and 11.9. The solutions of the exercises may be found at

```
http://www.lss.supelec.fr/books/intervals
```

11.2 Notions of C++

This section can obviously not replace a C++ course. More details about the language can be found in many excellent text books, including Stroustrup

L. Jaulin et al., *Applied Interval Analysis*

(1991), Capper (1994) and Anderson and Anderson (1998), or on the WEB, for example at

```
http://www.cetus-links.org/oo_c_plus_plus.html
```

or

```
http://math.nist.gov/~RPozo/c++class/
```

11.2.1 Program structure

It would be quite unreasonable for a large-scale project to consist of a single module. Rather, the project is organized as a series of more or less independent modules stored in *source files* (identified by an extension that depends on the compiler and may be `.cpp` or `.cc`). One of the modules may take care of graphics, a second one of interval computations, a third one of the handling of interval vectors, and so on. A given module may call functions of other modules, which requires the knowledge of the definitions and functions of these modules. This knowledge is provided by the *header file* associated with each module, identified by the extension `.hpp` or `.h`.

Consider, for instance, the module `myapp.cpp`, which exemplifies the `main` function required by any project. Assume that this module uses a library for interval computation, the source code of which is in `ival.cpp`. The header file `ival.h` of this library must be included in `myapp.cpp` as follows.

```
//-------------------------------------------------------
// File:    myapp.cpp
// Purpose: illustrates the inclusion of files

#include "ival.h"          // to use the interval library

main ()
{
  // rest of the module
  //...
}
//-------------------------------------------------------
```

Just before the compilation of `myapp.cpp`, a *precompiler* replaces the line `#include "ival.h"` by the content of `ival.h`. Header files should therefore contain as little code as possible, to avoid a needless increase in the size of the code to be compiled and of the resulting executable code. Source files thus extended are processed by the *compiler* to generate *object files*. Object files are then linked together by the *linker* to generate the *executable file*. The linker may also collect suitable parts of already compiled libraries (files with the extension `.lib`). The structure of a C++ project is summarized in Figure 11.1.

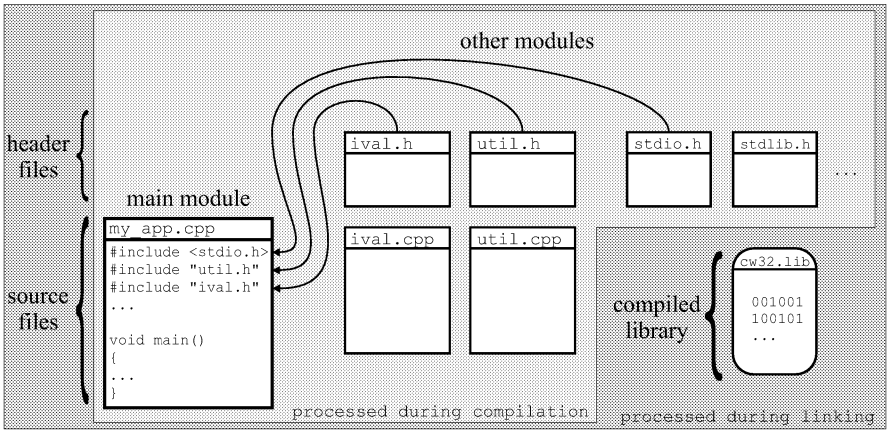


Fig. 11.1. Structure of a C++ project

Remark 11.1 *Any chain of characters following* `//` *in a given line is treated as a comment and ignored by the compiler. It is equally possible to indicate that a chain extending on several lines should be treated as a comment by inserting it between* `/*` *and* `*/`. ■

Other headers may be needed to allow the use of standard C++ libraries, for instance,

```
#include <iostream.h>
```

for standard input–output operations,

```
#include <math.h>
```

for mathematical functions such as sine or cosine.

The name of the header file to be included is put between brackets (`< >`) when it corresponds to a library provided with the compiler. Usually, this name is put between quotes (`" "`) for user-defined libraries. The type of delimiter indicates the directory in which the file is to be found.

11.2.2 Standard types

The most commonly used standard types in C++ are `char` for characters, `int` for signed integers, `float` and `double` for single and double precision floating-point numbers. Any variable must be declared before being used. This declaration may take place anywhere before the first use of the variable, and is performed as follows:

```
int i;    // integer variable
double x; // double-precision floating-point variable
```

C++ is case dependent and all statements end with a semicolon. A variable that cannot be modified during execution will be preceded by the *qualifier* `const`, as in

```
const int i = 3; // integer i cannot be modified
```

11.2.3 Pointers

The *address-of* operator `&` is employed to locate the memory where a variable is stored. Thus, `&x` contains the address where `x` is stored; `&x` is called a *pointer* to `x`. It is also possible to declare variables that are pointers:

```
double* px; // px is a pointer to a double
px = &x;    // px now contains the address of x
```

To access the variable stored at the address indicated by a pointer, the *dereferencing operator* `*` is used:

```
double x;
double* px; px = &x;
*px = 3; // *px is x, so x = 3
```

Remark 11.2 *Strictly speaking,* `double* px` *(declaration of the pointer* `px` *to a* `double`*) differs from* `double *px` *(declaration of the* `double` *pointed at by* `px`*). In practice, both are treated by the compiler in the same way, so the positions of spaces are not significant. However, the second syntax is recommended to avoid potential confusions appearing, e.g., when one wants to declare two pointers to* `double`*s in a single instruction. The correct syntax is* `double *px, *py`*, since* `double* px, py` *or* `double *px, py` *would correspond to the declaration of a pointer* `px` *and a* `double py`*.* ■

11.2.4 Passing parameters to a function

To illustrate how functions exchange information, consider first a function evaluating the mean of two double-precision floating-point numbers

```
double MeanValue(double a, double b)
{
  double m;      // m is local to the function
  m = (a + b) / 2;
  return m;
}
```

In this example, a double containing the value of `m` is returned to the calling function. The braces { } define the scope of a function or are used to group statements. The function `MeanValue` may be called as follows:

```
double alpha, beta, gamma;
alpha = 3; beta = 5;
gamma = MeanValue(alpha, beta);
```

The parameters of `MeanValue` are passed *by value*. At each call, a copy of each of them is made and the function works on these copies. Thus, `alpha` and `a` are stored at different locations. Any modification of the values of variables local to a function thus leaves the original parameters in the calling function unchanged.

Consider now a function exchanging the values of two `double`s. Since the parameters in the calling function must be changed, they cannot be passed by value. A possible technique is to pass them *by reference*:

```
void Exchange(double& a, double& b)
{
  double temp;
  temp = a;
  a = b;
  b = temp;
}
```

The `&` symbol in the list of arguments, not to be confused with the *address-of* operator represented by the same symbol, indicates that the following parameter is passed by reference. Again, spaces are not significant, so `double&a`, `double& a` and `double &a` are equivalent, but for the same reason as in Remark 11.2, it is recommended to write `double &a` in declaration statements. The `void` return type indicates that nothing is returned to the calling function. `Exchange` may be called as follows:

```
double alpha, beta;
alpha = 3; beta = 4;
Exchange(alpha, beta);  // now alpha = 4 and beta = 3
```

Here, `alpha` and `a` represent the same variable and share the same memory location.

Remark 11.3 *It is good practice to put the qualifier* `const` *in front of any argument of a function that should not be modified by this function, and this has recently become mandatory for* `const` *variables passed by reference.* ■

Finally, parameters may be passed *by address*. In this case, pointers to the parameters to be transmitted are passed. Among other things, this approach makes it possible to pass a function as an argument of another function, as will be seen in Section 11.8, page 327, and Section 11.12.3, page 342.

11.3 INTERVAL Class

Some more C++ notions will now be introduced to allow the creation of `INTERVAL` objects. Objects are entities characterized by their *properties* (or

data members) and by *member functions* (or *methods*) that apply to them. Member functions provide interfaces for accessing and modifying properties of objects. In C++, the notion of object is implemented using *classes*. An `INTERVAL` class will thus be designed, which will allow the application programmer to ignore the details of the implementation of intervals, just as one usually need not know the details of how floating-point numbers are represented. The `INTERVAL` class should contain (or *encapsulate*) the properties of intervals, the definition of arithmetical operations on intervals, input–output functions, etc.

The first step in the creation of the `INTERVAL` class of YOUR LIBRARY is the definition of a header file `ival.h` collecting all its properties and the headers of its member functions:

```
//-------------------------------------------------------
// File:     ival.h
// Purpose:  INTERVAL class specifications
#ifndef __INTERVAL__
#define __INTERVAL__

#include <iostream.h>        // for basic unformatted i/o

class INTERVAL{
  private:
    double inf, sup;
  public:
// constructors
    INTERVAL()                                    // default
      {inf = 0; sup = 0;}
    INTERVAL(const double a, const double b)//initialized
      {inf = a; sup = b;}
    INTERVAL(const INTERVAL& a)       // copy constructor
      {inf = a.inf; sup = a.sup;}
// destructor
    ~INTERVAL() {};
// other member functions
    INTERVAL& operator=(const INTERVAL&);   // assignment
// friend functions
  // read-only access functions
    friend double Inf(const INTERVAL& a)
      { return a.inf };
    friend double Sup(const INTERVAL& a)
      { return a.sup };
    friend double Diam(const INTERVAL&);
    friend INTERVAL Hull(const INTERVAL&,
                                        const INTERVAL&);
```

```
  // overloaded operators
    friend INTERVAL operator+(const INTERVAL&,
                                        const INTERVAL&);
    friend INTERVAL operator/(const INTERVAL&,
                                          const double);
    friend ostream& operator<<(ostream&,
                                        const INTERVAL&);
//...
};          // class definition blocks end by a semi-colon
#endif
//-------------------------------------------------------
```

The header file can stand as a documentation for the class. In what follows, the various parts of this header will be explained, and the reader will be asked to come back to it. As here, header files often begin with `#ifndef...`, `#define...` and end with `#endif`. These directives are used by the precompiler to prevent multiple inclusions by checking that the corresponding header file has not already been included. The name `__INTERVAL__` has been chosen arbitrarily, in such a way that the corresponding chain of characters cannot be found anywhere else in the source code.

The definition of a class starts by the keyword `class` followed by its the name and by a block `{...};` containing its properties and member functions. Note the semi-colon ending the block. An interval may be characterized by its lower and upper bounds, thus our `INTERVAL` class encapsulates two `private` properties, `inf` and `sup`, which correspond to these two bounds. Private properties can only be read or modified (using the *access operator* `.`) by member functions of the `INTERVAL` class. Thus, if `a` is an instance of the `INTERVAL` class, `a.inf` contains its lower bound. This protection of the private properties will make it possible, for instance, to prevent the instanciation of invalid intervals with their lower bound greater than their upper bound. The properties and member functions that can be accessed outside the class are placed after the keyword `public`. It is customary (but not mandatory) to give names starting with lower-case letters to the properties of a class and names starting with upper-case letters to the member functions.

Remark 11.4 *Other characterizations of intervals could have been considered (for instance, an interval could have been specified by its centre and width). This would obviously have had important consequences on the implementation of the rest of the class.* ■

11.3.1 Constructors and destructor

The instanciation of an `INTERVAL` is very similar to that of a `float`. Thus, one may write a program `firstapp.cpp` that uses the `ival` module as defined by `ival.h`

```
//-------------------------------------------------------
// File:    firstapp.cpp
// Purpose: INTERVAL instanciation

#include "ival.h"           // to use the INTERVAL class

int main()
{
  INTERVAL x;                 // default constructor used
  INTERVAL y(2,3);       // initialized constructor used
  INTERVAL z(y);                 // copy constructor used
  //...
}
//-------------------------------------------------------
```

These instanciations implicitly call *constructors*. Constructors are member functions with the same name as the class and no return type. They allow the initialization of the objects being created. Here, they specify the bounds of the intervals. Three constructors are presented, the implementation of which is performed directly inside the header file `ival.h` as they are very short. The *default constructor*, called for the instanciation of `x`, sets both of its bounds to 0. The *initialized constructor*, called for the instanciation of `y`, sets its lower bound to 2 and its upper bound to 3. This constructor may check the validity of the intervals that it creates, and may also emit an error message if appropriate (see Section 11.13). Finally, the *copy constructor*, called for the instanciation of `z`, copies the interval passed as an argument, so here $\mathtt{z} = [2, 3]$.

The *destructor* `~INTERVAL()` is called automatically at the end of a program or upon exit of a function, for each instance of the `INTERVAL` class that has been created within this program or function. Thus, when leaving the function `main()` of `firstapp.cpp`, the destructor will be called three times. Destructors make it possible, among other things, to free dynamically allocated memory (see Section 11.6).

Exercise 11.1 *Create a file* `ival.h` *as indicated, and supplement it with a constructor with a single* `double` *argument so as to instanciate punctual intervals.* ■

11.3.2 Other member functions

It is now possible to create and initialize intervals. The next member functions mentioned in `ival.h` (assignment, arithmetical operations) are more complicated, so their description will be put in a file `ival.cpp` in order to keep `ival.h` readable. The first member function mentioned after the destructor, as

```
INTERVAL& operator=(const INTERVAL&);
```

performs an *overloading* of the assignment operator `=` for intervals. This member function assigns the value of the `INTERVAL` argument to the calling `INTERVAL` instance. This argument is preceded by the qualifier `const`, indicating that it should not be modified in the body of the function. This protection is particularly useful in cascaded calls to functions with the same parameter, as the compiler will notify any attempt to modify a constant argument.

The code overloading the assignment operator may look as follows:

```
//-----------------------------------------------------
// File:    ival.cpp
// Purpose: INTERVAL class implementation

#include "ival.h"           // to use the INTERVAL class

INTERVAL& INTERVAL::operator=(const INTERVAL& a)
{
  if (this==&a)        // prevents self-assignment a = a
    return (*this);
  inf = a.inf; sup = a.sup;
  return (*this);
}
//...
//-----------------------------------------------------
```

`INTERVAL::` indicates that the member function belongs to the `INTERVAL` class. The keyword `this` corresponds to a pointer to the address of the current object (`*this` is thus a reference to the object itself). This address is compared to the address `&a` of the argument `a` to avoid self-assignment. The rest of the member function is similar to the copy constructor. The overloaded operator `=` may be employed as follows:

```
INTERVAL x, y(3,4), z(1,3);
x = y;
y = x = z;  // cascading is possible
            // x and y are now equal to z
```

The last functions listed in `ival.h` are *friend functions* of the `INTERVAL` class. Friend and member functions share access to the private members of the class, but the syntaxes of their calls differ. Friend-functions calls obey the usual syntax for mathematical functions. The friend functions `Inf` and `Sup` of the `INTERVAL` class provide read-only access to the interval bounds:

```
INTERVAL x(3,4);
double lowerbound;
lowerbound = Inf(x);  // call of a friend function
```

A member function would require the use of the access operator . and thus be less intuitive. This is why friend functions will be used to evaluate properties of an interval, such as its width or its centre, or to implement usual mathematical functions (`Cos`, `Sin`, `Exp`...).

A possible implementation of a friend function computing the width of an interval (and called `Diam` for compatibility with PROFIL/BIAS) is

```
//-----------------------------------------------------
// File: ival.cpp (continued)
//...
double Diam(const INTERVAL& a)    // evaluation of width
{ return (a.sup - a.inf); }
//...
//-----------------------------------------------------
```

Since friend functions are not considered as members, they are not preceded by `INTERVAL::`. Binary operators will also be implemented as friend functions. The code for the addition operator + may thus be:

```
//-----------------------------------------------------
// File:  ival.cpp (continued)
//...
#include <float.h>          // for rounding-mode control
//...
INTERVAL operator+(const INTERVAL& a, const INTERVAL& b)
{
  INTERVAL res;
  unsigned int cw = _control87(NULL,NULL); // stores the
                                  // current rounding mode
  _control87(RC_DOWN,MCW_RC); // rounding is towards -oo
  res.inf = a.inf + b.inf;
  _control87(RC_UP,MCW_RC);   // rounding is towards +oo
  res.sup = a.sup + b.sup;
  _control87(cw,MCW_RC);        // initial rounding mode
                                          // is restored
  return res;
}
//...
//-----------------------------------------------------
```

Including the module `float` allows the statement `_control87(.,.)` to be used to retrieve or change the floating-point control word of an INTEL-compatible mathematical coprocessor, according to the IEEE 754 standard (see Chapter 10). It is particularly useful to specify the rounding modes used for the outward rounding of the results of arithmetical operations. As mentioned in Section 10.3.2, page 294, it is advisable to store the rounding mode before modifying it and, to set it back to its initial value upon exit.

Exercise 11.2 *Supplement* ival.cpp *with the friend function* Hull *computing the interval hull of two* INTERVALs. *The header of* Hull *is already in* ival.h. ■

Exercise 11.3 *Based on* Diam, *supplement* ival.h *and* ival.cpp *with other friend functions such as*

```
friend double Mid(const INTERVAL& a);
```

computing the centre of an interval a *and*

```
friend int Intersection(INTERVAL& r, const INTERVAL& a,
                                     const INTERVAL& b);
```

where r *is the intersection of* a *and* b. *This function should return* 1 *if the intersection is non-empty and* 0 *otherwise.* ■

Exercise 11.4 *Overload* operator- *and* operator* *for two* INTERVALs. ■

Exercise 11.5 *Overload* operator<= *to test whether an* INTERVAL a *is included in an* INTERVAL b. *The header of this function should be*

```
friend int operator<=(const INTERVAL& a, const INTERVAL& b);
```

This function should return 1 *if* a *is included in* b *and* 0 *otherwise.* ■

The overloading of the operator / for the division of an interval by an interval poses no particular problem unless the divisor contains zero. In the latter case, various policies may be followed, see Section 11.13, page 349. Consider first the simpler situation where the divisor is not an interval but a double. If the divisor is zero, then the following code returns an approximation of the entire real line under the form of the interval [-Infinity, Infinity] (see Section 10.2.3, page 291):

```
//-------------------------------------------------------
// File: ival.cpp (continued)
//...
INTERVAL operator/(const INTERVAL& a, const double b)
{
  INTERVAL res;
  unsigned int cw = _control87(NULL,NULL); // stores the
                                  // current rounding mode
  if (b > 0)
  { _control87(RC_DOWN,MCW_RC);  // rounding towards -oo
    res.inf = a.inf / b;
    _control87(RC_UP,MCW_RC);    // rounding towards +oo
    res.sup = a.sup / b; }
  else if (b < 0)
  { _control87(RC_DOWN,MCW_RC);  // rounding towards -oo
```

```
    res.inf = a.sup / b;
    _control87(RC_UP,MCW_RC);    // rounding towards +oo
    res.sup = a.inf / b; }
  else
  { res.inf = -Infinity; res.sup = Infinity; }
  _control87(cw,MCW_RC);         // restores the initial
                                        // rounding mode
  return res;
}
//...
//-------------------------------------------------------
```

The quantity `Infinity`, corresponding to $+\infty$ has not been defined yet and is not C++ standard. So it has to be defined using its bit pattern (see Section 10.2.3). As `Infinity` will be used by many functions, it should be placed at the beginning of `ival.h`.

```
//-------------------------------------------------------
// File:     ival.h
// Purpose:  INTERVAL class specifications
#ifndef __INTERVAL__
#define __INTERVAL__

#include <iostream.h>        // for basic unformatted i/o

// definition of Infinity using its bit pattern
union UREAL { unsigned short ushort[4]; double real; };
static union UREAL PosInfty = {{ 0x0000, 0x0000,
                                       0x0000, 0x7FF0 }};
static double Infinity = PosInfty.real;

class INTERVAL{
...
//-------------------------------------------------------
```

Exercise 11.6 *Overload* `operator/` *for two* `INTERVAL`*s. When* 0 *belongs to the interval divisor,* `[-Infinity, Infinity]` *should be returned (see Section 10.3.2, page 294).* ■

The last friend function listed in `ival.h` overloads the binary *insertion operator* `<<` to allow the insertion of an interval in an output stream. This makes it possible to write the two bounds of an interval on the screen or to a file. It may be implemented as follows:

```
//-------------------------------------------------------
// File: ival.cpp (continued)
//...
```

```
#include <iostream.h>     // to allow streams to be used
//...
ostream& operator<<(ostream& os, const INTERVAL& a)
{
  os << "[" << a.inf << "," << a.sup << "]";
  return (os);
}
//...
//-------------------------------------------------------
```

The lines

```
INTERVAL a(2,3);
cout << "a = " << a << endl;  // console output
```

produce the following output to the screen (console):

```
a = [2,3]
```

This version of the INTERVAL class, when completed with the results of the exercises, allows most basic arithmetical operations on intervals.

11.3.3 Mathematical functions

The inclusion functions for the standard mathematical functions remain to be implemented. As they are not always needed, they will be put in a separate module, called func for compatibility with PROFIL/BIAS. This module constitutes an interval counterpart to the standard mathematical library math.

The header func.h lists the functions to be implemented:

```
//-------------------------------------------------------
// File:    func.h
// Purpose: standard math functions for INTERVALs
#ifndef __FUNCTIONS__
#define __FUNCTIONS__

#include "ival.h"           // to use the INTERVAL class
#include <math.h>   // for standard mathematical library

INTERVAL Exp    (const INTERVAL& x);
INTERVAL Log    (const INTERVAL& x);
//...
INTERVAL Sin    (const INTERVAL& x);
INTERVAL Cos    (const INTERVAL& x);
INTERVAL Tan    (const INTERVAL& x);
//...
INTERVAL Sqr    (const INTERVAL& x);
```

```
INTERVAL Sqrt   (const INTERVAL& x);
//...
INTERVAL ArcSin (const INTERVAL& x);
INTERVAL ArcTan (const INTERVAL& x);
//...
#endif
//-----------------------------------------------------
```

Only some of these functions will be presented in detail. Based on these examples, the implementation of the others should pose no problem.

Monotonic functions are particularly simple to implement. For instance, an inclusion function for the exponential function could be implemented in `func.cpp` as follows

```
//-----------------------------------------------------
// File:    func.cpp
// Purpose: standard math functions for INTERVALs
#include "func.h"
//...
INTERVAL Exp(const INTERVAL& x)
{ return INTERVAL(exp(Inf(x)), exp(Sup(x))); }
//...
//-----------------------------------------------------
```

Outward rounding was not implemented. As a matter of fact, rounding control is not requested by IEEE 754 for the standard functions, except for `sqrt`, and guaranteed rounding for transcendental functions is still an active domain of research (see Section 10.2.2, page 289).

The implementation of an inclusion function for the logarithm function is similar, except that the domain of definition of this function is $\mathbb{R}^+$. The policy described in Section 10.3.3, page 295, may be followed. Another possibility is to implement a special error treatment when an interval argument $[x]$ is not entirely included in $\mathbb{R}^+$ (see Section 11.13, page 349).

Exercise 11.7 *Based on the code for* `Exp`*, supplement* `func.cpp` *with inclusion functions for other monotonic functions, such as* `Sqrt` *for the square root.* ■

For non-monotonic functions, the use of an algorithm is usually necessary. Thus, to obtain an inclusion function for the square function, one may write:

```
//-----------------------------------------------------
// File:    func.cpp (continued)
//...
INTERVAL Sqr(const INTERVAL& x)
{
  double infsqr = Inf(x)*Inf(x);
```

```
  double supsqr = Sup(x)*Sup(x);
  if (Inf(x) >= 0)
    return INTERVAL(infsqr, supsqr);
  else if (Sup(x) <= 0)
    return INTERVAL(supsqr, infsqr);
  else
    return INTERVAL(0, max(infsqr, supsqr));
}
//...
//-------------------------------------------------------
```

Exercise 11.8 *Build an inclusion function* `Sin` *for the sine function. From the result, build inclusion functions* `Cos` *and* `Tan` *for the cosine and tangent functions.* ■

The library built so far may be used to treat the exercises of Section 11.5. One may prefer to use a ready-made library, as suggested in the next section.

11.4 Intervals with PROFIL/BIAS

PROFIL/BIAS is a library for interval computation. It runs under many operating systems, from UNIX to DOS *via* OS/2. It may also be run under WINDOWS, after slight modifications of its source code. Rounding-mode control is provided for a variety of processors, including RS/6000, SPARC and PENTIUM. It is also highly configurable (one may, for example, give up outward rounding in order to speed up computation). The library can be downloaded from

`ftp://ti3sun.ti3.tu-harburg.de/pub/profil/`

A slightly modified version of this library (see `readme.txt`) may be downloaded from

`http://www.lss.supelec.fr/books/intervals`

Documentation for PROFIL/BIAS may be found at

`http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html`

The library consists of two layers, namely BIAS and PROFIL.

11.4.1 BIAS

BIAS (for Basic Interval Arithmetic Subroutines), has been written in C in the spirit of the FORTRAN BLAS library (for Basic Linear Algebra Subroutines). It allows computation on punctual and interval scalars, vectors and matrices. The basic arithmetic operations, rounding controls and elementary

mathematical functions are implemented. Guaranteed results are obtained using rounding mode control for all the operations for which this is made possible by the IEEE 754 standard. BIAS consists of four main modules:

- `bias0` handles floating-point and interval scalars;
- `bias1` handles floating-point and interval vectors;
- `bias2` handles floating-point and interval matrices;
- `biasf` implements inclusion functions for the usual mathematical functions.

As the BIAS layer is transparent to the user, it will not be described in detail here; see Knüppel (1993, 1994) or the documentation available on the WEB for more information.

11.4.2 PROFIL

PROFIL (for Programmer's Run time Optimized Fast Interval Library) is a C++ interface for BIAS, which takes advantage of the availability of classes and operator overloading to make statements involving interval scalars, interval vectors and interval matrices syntactically identical to those involving their floating-point counterparts. Portability has been ensured by using standard C++, but aliases have been defined for the names of some types of variable, using the keyword `typedef`. Thus

```
typedef double REAL;
```

makes `REAL` a synonym of `double`. The correspondence for the most frequent types of variable is indicated in Table 11.1. For pointers, PROFIL/BIAS defines `PREAL` as a synonym of `REAL*`. More generally, any PROFIL/BIAS specific type preceded by `P` corresponds to a pointer to a variable of the type.

Table 11.1. Some standard C++ types and their PROFIL/BIAS equivalent

Standard C++	PROFIL/BIAS	Variable type
`void`	`VOID`	none
`char`		character
`int`	`INT`	signed integer
`float`		single-precision floating-point number
`double`	`REAL`	double-precision floating-point number

11.4.3 Getting started

The next program performs the addition of intervals:

```
//------------------------------------------------------
// File:    add.cpp
// Purpose: addition of INTERVALs
#include "ival.h"   // interval module of PROFIL library

void main()
{
  INTERVAL a(3,4);                          // a = [3,4]
  INTERVAL b(5,7);                          // b = [5,7]
  INTERVAL c;
  c = a + b;
// console output
  cout << a << " + " << b << " = " << c << endl;
}
//------------------------------------------------------
```

To allow compilation of this program, the modules `bias0` and `ival` must be included in the project. Addition (or any other standard operation) is simply written like its real counterpart. An output to the screen is performed using the standard insertion operator `<<`.

All the standard mathematical functions (`Cos`, `Sin`, `Tan`, `Exp`, `Log`, `Log10`, `Sqrt`, `Sqr`...) are provided by the `func` module of PROFIL (see the `func.h` header file). The use of the `func` module requires the `biasf` module (or a library containing it) to be included in the project. Note that the PROFIL/BIAS names of the standard mathematical functions start with capital letters. These functions can be evaluated with floating-point and interval arguments. Names starting with lower-case letters can only be used with floating-point arguments.

The following example illustrates the use of elementary mathematical functions:

```
//------------------------------------------------------
// File:    standmat.cpp
// Purpose: standard math functions for INTERVALs
#include "ival.h"   // interval module of PROFIL library
#include "func.h"    // standard functions for INTERVALs

void main()
{
  INTERVAL a(3,4);
  INTERVAL b(-2,3);
  INTERVAL c,d;
```

```
    c = Exp(a); d = Sqr(b);

    // console output
    cout << "Exp(" << a << ") = " << c << endl;
    cout << "Sqr(" << b << ") = " << d << endl;
}
//-------------------------------------------------------
```

11.5 Exercises on Intervals

These exercises may indifferently be treated with YOUR LIBRARY or PROFIL/BIAS.

Exercise 11.9 *Write a program to compare the values of the two following inclusion functions for* $x^2 - x$ *at* $[x] = [-1, 3]$*:*

$$[f]_1([x]) = [x]^2 - [x],$$

$$[f]_2([x]) = ([x] - 1/2)^2 - 1/4.$$

This program requires the `ival` *module to be included in the project.* ■

Exercise 11.10 *Consider the function* f *defined by* $f(x) = x^2 + \sin(x)$*. Compute the images obtained for* $[x]_1 = [\frac{2\pi}{3}, \frac{4\pi}{3}]$ *and* $[x]_2 = [\frac{99\pi}{100}, \frac{101\pi}{100}]$ *when using the natural, mean, Taylor and minimal inclusion functions associated with* f*, as given in page 36. Compare the performances of these inclusion functions.* ■

Exercise 11.11 *Consider the equation*

$$f(\mathbf{x}) = 0, \tag{11.1}$$

where

$$f(\mathbf{x}) = x_1 \exp(x_2) + \sin(x_3). \tag{11.2}$$

The prior domains for the variables x_1, x_2 *and* x_3 *are chosen as*

$$[-1000, 0] \times [-10, 10] \times [-3.14, 3.14]. \tag{11.3}$$

The purpose of this exercise is to implement a forward–backward propagation algorithm (see Section 4.2.4, page 77) to compute domains for x_1, x_2 *and* x_3 *compatible with* (11.1)*. First, write an algorithm computing the domain* $[y] = [f]([x_1], [x_2], [x_3])$ *by forward propagation, and check that* $0 \in [y]$*. Then, using Table 4.6, page 79, implement backward propagation. Does it lead to an efficient reduction? Remember to check whether intersections are empty, which would indicate that there is no solution to (11.1) in (11.3). Try other prior domains.* ■

11.6 Interval Vectors

In order to build a class of interval vectors, a few more notions of C++ are needed. Several approaches are available to create vectors. The simplest one is to use an array. For instance

```
float a[5];
```

creates an array `a` of five `float`s, *i.e.*, allocates the space for five `float`s in the memory of the computer. Access to the entries of this array is then performed as follows:

```
a[0] = 4;                    // writes on first entry of a
a[3] = 3.5;
a[4] = a[0] * a[3];           // writes on last entry of a
```

Note that, contrary to mathematical usage, the first element of this array is indexed by 0 and not by 1. The size of the array is *fixed at compile time*, which lacks flexibility. To be able to handle *dynamic* arrays (*i.e.*, arrays with variable sizes) one must resort to the use of pointers (Section 11.2.3, page 304). A pointer to the memory location where the first entry of the vector will be stored must first be created, for example, by

```
float *pa;  // pa points to a float or an array of floats
```

Memory must then be allocated *dynamically* (*i.e.*, at run time) to store `n` `float`s. This is performed by the statement

```
pa = new float[n];     // an array of n floats is created
```

Of course, the positive integer `n` must have been given a suitable numerical value. Access to the entries of the array may then be performed as follows:

```
pa[n-1] = pa[0] + pa[4];    // writes on last entry of pa
```

To protect oneself against resource leak, care must be taken to free memory space when it is no longer needed. This is performed with the keyword `delete` for scalars and `delete[]` for arrays.

```
delete[] pa;           // frees the memory allocated to pa
```

Consider now an array `a` of five `float`s. The statement

```
a[5] = 6.4;                    // writes outside the array
```

assigns a value to the sixth entry of `a`, which is outside the space allocated to this array. As a result, memory space in principle available for other tasks is overwritten, with possibly fatal consequences, although no error is generated during compilation. This can be avoided if access control is implemented. In C++, vector classes can be defined to take care of these problems of memory management, as will be seen in the next section.

11.6.1 INTERVAL_VECTOR class

An interval vector can be characterized by its number of entries, which may be stored in an integer, and the address of the memory location where its first entry is placed. This address may be stored in a pointer to an INTERVAL. The integer and pointer form the private properties of an INTERVAL_VECTOR, whose header, called ivalvec.h again for compatibility with PROFIL/BIAS, may be as follows:

```
//-------------------------------------------------------
// File:    ivalvec.h
// Purpose: specification of the INTERVAL_VECTOR class
#ifndef __INTERVAL_VECTOR__
#define __INTERVAL_VECTOR__

#include "ival.h"                    // to use INTERVALs
#include <iostream.h>    // to use basic unformatted i/o

class INTERVAL_VECTOR{
  private:
    int nElements;
    INTERVAL* theElements;
  public:
// constructors
    INTERVAL_VECTOR()                          // default
      {nElements = 0; theElements = NULL;}
    INTERVAL_VECTOR(int n)               // initialized
      {// some code checking that n > 0
       // could be inserted here
       nElements = n; theElements = new INTERVAL[n];}
    INTERVAL_VECTOR(const INTERVAL_VECTOR&);    // copy
// destructor
    ~INTERVAL_VECTOR()
      {delete[] theElements;}
// other member functions
  // assignment
    INTERVAL_VECTOR& operator=(const INTERVAL_VECTOR&);
  // fonction call operator
    INTERVAL& operator()(int);
// friend functions
  // read-only access functions
    friend int Dimension(const INTERVAL_VECTOR& a)
      { return a.nElements; }
  // overloaded operators
    friend INTERVAL_VECTOR operator+
      (const INTERVAL_VECTOR&, const INTERVAL_VECTOR&);
```

```
    friend INTERVAL_VECTOR operator-
      (const INTERVAL_VECTOR&, const INTERVAL_VECTOR&);
    friend INTERVAL_VECTOR operator*
      (const INTERVAL_VECTOR&, const INTERVAL&);
    friend ostream& operator<<(ostream&,
                               const INTERVAL_VECTOR&);
//...
};
#endif
//-------------------------------------------------------
```

In what follows, the various parts of this header will be explained, and the reader will be asked to refer to it.

Remark 11.5 *The various* PROFIL/BIAS *vector classes contain one more private property, the integer* `IsTemporary` *which serves as a flag indicating if a vector is the temporary result of an intermediary operation that can be destructed upon completion. Using this flag improves code efficiency, at the cost of some restriction in syntax. This point will not be considered in* YOUR LIBRARY. ■

11.6.2 Constructors, assignment and function call operators

Dynamic allocation of memory is necessary whenever a non-empty vector is created. This is performed by all constructors of the `INTERVAL_VECTOR` class, with the exception of the default constructor that creates a vector with no entry. As an illustration, consider the following copy constructor, to be placed in `ivalvec.cpp` (the default and initialized constructors, much shorter, are in `ivalvec.h`):

```
//-------------------------------------------------------
// File:     ivalvec.cpp
// Purpose:  implementation of the INTERVAL_VECTOR class
#include "ivalvec.h"          // to use INTERVAL_VECTORs

INTERVAL_VECTOR::INTERVAL_VECTOR
                              (const INTERVAL_VECTOR& v)
{
   nElements = v.nElements;
   if (v.theElements == NULL) theElements = NULL;
   else theElements = new INTERVAL[nElements];
   for (int i = 0; i < nElements; i++)
     theElements[i] = v.theElements[i];
}
//...
//-------------------------------------------------------
```

Remark 11.6 *For the sake of simplicity, the content of each component of* `v` *is copied here using a loop. This could be done much faster by copying the content of the array pointed by* `v.theElements` *into the array pointed by* `theElements`*, using the* `memcpy` *function of the standard* `memory` *module.* ■

The number of entries of the new instance of the `INTERVAL_VECTOR` class is thus made identical to that of `v`, and the same amount of memory as for `v` is dynamically allocated. Each entry of `v` is then copied into its counterpart in the array allocated by `new`. When the allocation fails for lack of memory, `new` returns a `NULL` pointer, which can be used to detect and handle this error (see Section 11.13, page 349).

The destructor is systematically called when an object is no longer used. It takes care of freeing the corresponding memory, as shown in `ivalvec.h` above.

The assignment operator resembles the copy constructor:

```
//-------------------------------------------------------
// File:     ivalvec.cpp (continued)
//...
INTERVAL_VECTOR& INTERVAL_VECTOR::operator=
                                (const INTERVAL_VECTOR& v)
{
   if (this == &v) return (*this);
   if (nElements != v.nElements)
     { nElements = v.nElements; delete[] theElements; }
   if (v.theElements == NULL) theElements = NULL;
   else theElements = new INTERVAL[nElements];
   for (int i = 0; i < nElements; i++)
     theElements[i] = v.theElements[i];
   return (*this);
}
//...
//-------------------------------------------------------
```

First, the address of `v` is compared to that of the invoking object (`this`) to prevent self assignment. If the two vectors do not have the same number of elements, the invoking object must be re-sized. The memory allocated to `theElements` is then freed before being reallocated. The content of `v` is then copied to `this`. A reference to the calling object is finally returned to allow the cascading of assignments, as in `a = b = c;` where `a`, `b` and `c` are `INTERVAL_VECTOR`s.

Exercise 11.12 *Based on the model of* `ivalvec.h`*, write the header file* `vector.h` *of a module implementing vectors of* `double`*s.* ■

The *function call* operator, denoted by `operator()(...)`, can be used for accessing entries of vectors under boundary control. A possible implementation is as follows:

```
//-------------------------------------------------------
// File:     ivalvec.cpp (continued)
//...
#include <stdlib.h>              // for the exit function
//...
INTERVAL& INTERVAL_VECTOR::operator()(int i)
{
  if ((i < 1)||(i > nElements))  // basic error handling
  { cout << "attempt to violate the boundary of "
         << "INTERVAL_VECTOR " << (*this) << endl;
    exit(EXIT_FAILURE); }
  return (*(theElements + i - 1));  // returns ith entry
}
//...
//-------------------------------------------------------
```

The index `i` is first tested to determine whether it belongs to the interval of valid indices (from `1` to `nElements`). Either an error is generated and the instruction `exit(EXIT_FAILURE)` causes the program to abort, or the reference to the `INTERVAL` corresponding to the index is returned. This reference provides read-write access to the entries of the vector.

```
INTERVAL_VECTOR a(4); // a 4-component vector is created
a(3) = 4;
a(4) = a(3);
a(6) = 0;                              // generates an error
```

The possibility of inserting instructions to detect and handle errors in the code of member functions has been illustrated on the function call operator. The lines that would allow a similar detection and handling will be omitted from the code of the next functions, for the sake of brevity. In practice, they should not be dispensed with if a robust library is to be built (see also Section 11.13, page 349).

Remark 11.7 `operator()(...)` *has been implemented in such a way that the smallest valid index is* `1` *(as usual in mathematics). When* `i` *is equal to* `1`*, what is returned is* $*$(`theElements`), i.e., *the first entry of the vector.* ■

Remark 11.8 *The test* `if((i<1)||(i>nElements))` *involves the logical* OR *operator, written as* `||`*. The logical* AND *is written as* `&&` *and the complementation operator* $\neg$ *is denoted by an exclamation mark. For instance,* `!(5>0)` *holds false.* ■

11.6.3 Friend functions

The first friend function in `ivalvec.h` provides read-only access to the size of the vector. The next friend functions extend addition, subtraction and

multiplication by a scalar to INTERVAL_VECTORs. This is done by component-wise application of the corresponding scalar operators. The overloading of operator+ for two INTERVAL_VECTORs is, for instance, performed by

```
//-------------------------------------------------------
// File:     ivalvec.cpp (continued)
//...
INTERVAL_VECTOR operator+(const INTERVAL_VECTOR& a,
                                const INTERVAL_VECTOR& b)
{
  // creates a vector to store the result
  INTERVAL_VECTOR res(a.nElements);
  for (int i = 1; i <= a.nElements; i++)
    res(i) = a(i) + b(i);
  return (res);
}
//...
//-------------------------------------------------------
```

Here, component-wise addition is performed. This implementation is far from being optimal, because for each addition of scalar intervals, the overloaded function call operator is called three times, which involves as many unnecessary checkings of index ranges. This addition would be implemented more efficiently as

```
res.theElements[i-1] = a.theElements[i-1]
                                + b.theElements[i-1];
```

which does not use the function call operator.

Exercise 11.13 *Supplement* ivalvec.h *and* ivalvec.cpp *with the overloading of*

1. operator- *for the subtraction of two* INTERVAL_VECTOR*s,*
2. operator* *for the product of an* INTERVAL_VECTOR *by an* INTERVAL*,*
3. operator* *for the product of an* INTERVAL *by an* INTERVAL_VECTOR*,*
4. operator<< *to obtain a basic output of an* INTERVAL_VECTOR *to a stream. This function may be implemented as a friend function with the header*

```
friend ostream& operator<<(ostream&,
                                const INTERVAL_VECTOR&);
```

■

Exercise 11.14 *Supplement* ivalvec.h *and* ivalvec.cpp *with a friend functions evaluating the* INTERVAL_VECTOR *corresponding to the interval hull of two* INTERVAL_VECTOR*s. The header of this function may be*

```
friend INTERVAL_VECTOR Hull(const INTERVAL_VECTOR& a,
                            const INTERVAL_VECTOR& b);
```

■

Exercise 11.15 *Supplement* `ivalvec.h` *and* `ivalvec.cpp` *with a friend function evaluating the intersection of two* `INTERVAL_VECTOR`*s. The header of this function may be*

```
friend int Intersection(INTERVAL_VECTOR& r,
     const INTERVAL_VECTOR& a, const INTERVAL_VECTOR& b);
```

where `r` *is the intersection of* `a` *and* `b`*. This function should return* `1` *if the interval vectors intersect and* `0` *otherwise.* ■

Exercise 11.16 *Supplement* `ivalvec.h` *and* `ivalvec.cpp` *with friend functions providing a test to check the inclusion of an interval vector in another one. This could be done by overloading* `operator<=`*, and a possible header is*

```
friend int operator<=(const INTERVAL_VECTOR& a,
                      const INTERVAL_VECTOR& b);
```

The test should return `1` *if* `a` *is included in* `b` *and* `0` *otherwise.* ■

11.6.4 Utilities

The implementation of algorithms such as SIVIA requires specific tools, to be stored in a module `util`. This module implements, for instance, the bisection of an interval vector `x` across its `i`th dimension *via* the functions `Lower` and `Upper`, which compute the two interval vectors resulting from the bisection.

```
//-------------------------------------------------------
// File:     util.h
// Purpose:  INTERVAL utilities
#ifndef __UTILITIES__
#define __UTILITIES__
#include "ivalvec.h"          // to use INTERVAL_VECTORs

INTERVAL_VECTOR Lower (const INTERVAL_VECTOR& x, int i);
INTERVAL_VECTOR Upper (const INTERVAL_VECTOR& x, int i);
//...
#endif
//-------------------------------------------------------
```

Remark 11.9 *In the rest of the book, we chose to call* left box *and* right box *the two boxes resulting from a bisection, by analogy with the left and right subtrees (see Section 3.3.2, page 51). Here,* lower *and* upper *are used instead of* left *and* right *to ensure compatibility with* PROFIL/BIAS. ■

`Upper` may be implemented as follows:

```
//-----------------------------------------------------
// File:     util.cpp
// Purpose:  INTERVAL utilities
#include "util.h"

INTERVAL_VECTOR Upper (const INTERVAL_VECTOR& x, int i)
{
  INTERVAL_VECTOR t(x);
  t(i) = INTERVAL(Mid(x(i)), Sup(x(i)));
  return t;
}
//...
//-----------------------------------------------------
```

Exercise 11.17 *Implement* `Lower`. ■

11.7 Vectors with PROFIL/BIAS

Three vector classes are available in PROFIL/BIAS.

Module name	Class name	Description
`vector`	`VECTOR`	vectors of doubles
`intvec`	`INTEGER_VECTOR`	vectors of integers
`ivalvec`	`INTERVAL_VECTOR`	vectors of intervals

The use of the `INTERVAL_VECTOR` class requires the `bias0` and `bias1` modules (or a library containing them) to be included in the project. The following simple program illustrates some basic features of vectors implemented using PROFIL/BIAS.

```
//-----------------------------------------------------
// File:    addvect.cpp
// Purpose: addition of two INTERVAL_VECTORs
#include "ivalvec.h"        // INTERVAL_VECTORs library

void main()
{
  INTERVAL_VECTOR a(3);            // the dimension of a
  INTERVAL_VECTOR b(3);                // and b is three
  INTERVAL_VECTOR c;             // c is not initialized
```

```
  a(1) = INTERVAL(5,7);
  a(2) = INTERVAL(-2,3);
  a(3) = -2;  // all components of a are now initialized
  b = a;
  b(3) = INTERVAL(4,5);
  c = a + b;

  // console output
  cout << a << " + " << b << " = " << c << endl;
}
//-----------------------------------------------------
```

After the declaration of three `INTERVAL_VECTOR`s, two of them are initialized and added. The result is stored in the third, and the result displayed on the console. The size of `c` is adjusted to that of the result of `a + b`; memory management is thus transparent to the user.

Other functions such as multiplication or division of a vector by a scalar, scalar product, etc. are also available, see the PROFIL/BIAS documentation.

11.8 Exercises on Interval Vectors

The aim of this section is to build and use a first version of SIVIA. Only the tools developed in YOUR LIBRARY will be required, but all the exercises of this section may also be treated with PROFIL/BIAS.

The functions to be developed will be placed in a module called `sivia`.

```
//-----------------------------------------------------
// File:     sivia.h
// Purpose:  first version of Sivia
#include "ivalvec.h"          // to use INTERVAL_VECTORs

// defines a new type "interval Booleans"
typedef enum{IB_TRUE, IB_FALSE, IB_INDET} INTERVAL_BOOL;
// PIBT stands for "Pointer to an Interval Boolean Test"
typedef INTERVAL_BOOL (*PIBT)(const INTERVAL_VECTOR&);
double MaxDiam (const INTERVAL_VECTOR&, int&);
void Sivia (PIBT, const INTERVAL_VECTOR&, double);
//-----------------------------------------------------
```

`sivia.h` starts with the definition of two new types of variable (using the keyword `typedef`). The first new type `INTERVAL_BOOL` corresponds to interval Booleans (see Section 2.5.1, page 38), *i.e.*, to variables that can take their values in the set defined after the keyword `enum`, namely `IB_TRUE`, `IB_FALSE`, and `IB_INDET` for indeterminate. The prefix `IB_` was introduced to avoid conflicts

with `TRUE` and `FALSE` of PROFIL/BIAS. The second new type `PIBT` corresponds to pointers to interval Boolean tests. It allows a test to be passed as a parameter of a function. Thus, an interval Boolean test can be passed to `Sivia` as a parameter. This makes `Sivia` flexible by allowing it to work on different tests.

The syntax of the definition of a type of pointer to a function is

```
typedef return_type (*type_name)(type of the parameters);
```

A pointer of the type `PIBT` thus points to a function having a constant reference to an `INTERVAL_VECTOR` as a parameter and returning an `INTERVAL_BOOL`. The header of the function implementing the test should look like

```
INTERVAL_BOOL Name_of_the_test (const INTERVAL_VECTOR&);
```

Once this test has been defined, `Sivia` can be called by

```
Sivia(Name_of_the_test, X, eps);
```

where `X` is the predefined search box, and `eps` the predefined precision factor (see Section 3.4.1, page 55). To characterize a set defined by another test, it suffices to create a new function similar to `Name_of_the_test`.

Exercise 11.18 *Build an interval Boolean test* `IvalBoolTest` *to be used by* `Sivia` *to characterize the set*

$$S = \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; x^4 - 4x^2 + 4y^2 \in [-0.1, 0.1] \right\}.$$

This test should return `IB_TRUE` *if* $[x]^4 - 4[x]^2 + 4[y]^2 \subset [-0.1, 0.1]$, `IB_FALSE` *if* $[x]^4 - 4[x]^2 + 4[y]^2 \cap [-0.1, 0.1] = \varnothing$ *and* `IB_INDET` *otherwise.* ■

The result of the next exercise will be used by `Sivia` to select the dimension across which indeterminate boxes will be bisected.

Exercise 11.19 *Code* `MaxDiam` *using the* `Diam` *function provided in the* `ival` *module.* `MaxDiam` *should return a* `double` *containing the width of the box passed as a parameter. The index of the first component with maximal width of this box should be passed back to the calling function by reference to an* `int`. ■

Exercise 11.20 *Code* `Sivia` *using a recursive structure (*i.e., *the routine will call itself).* `IvalBoolTest` *should be called first. If its result is true or false, then a message indicating the result should be displayed, followed by a return statement. Else the box is indeterminate, and its width should be computed using* `MaxDiam`. *If it turns out to be lower than the precision parameter* `eps`, *then a message should be displayed, followed by a return statement. Else, the current box should be bisected using* `Lower` *and* `Upper`, *and* `Sivia` *should be called for each of the resulting subboxes.* ■

Solutions to Exercises 11.18 to 11.20 may be found below:

```
//-------------------------------------------------------
// File:     sivia.cpp
// Purpose:  first version of Sivia
#include "sivia.h"
#include "util.h"              // to use Lower and Upper

int MaxDiam(const INTERVAL_VECTOR& x)
{
  int mdcomp = 1;                        // initialization
  double diam = Diam(x(mdcomp));
  // the widths of all components must be compared
  for (int i = 2; i <= Dimension(x); i++)
    if (Diam(x(i)) > diam)  // a component with a larger
      { diam = Diam(x(i)); mdcomp = i; }  // width found
  return mdcomp;
}

void Sivia(PIBT IBTest, const INTERVAL_VECTOR& x,
                                              double eps)
{
   int test = IBTest(x);
   if (test == IB_TRUE)
   { cout << x << " is included in S" << endl;
     return; }
   if (test == IB_FALSE)
   { cout << x << " does not intersect S" << endl;
     return; }

   int maxdiamcomp;
   if (test == IB_INDET)
   { if (MaxDiam(x, maxdiamcomp) < eps)
        { cout << x << " is indeterminate" << endl;
          return; }     // x is too small to be bisected
    // x is bisected into xlow and xup ...
      INTERVAL_VECTOR xlow = Lower(x, maxdiamcomp);
      INTERVAL_VECTOR xup = Upper(x, maxdiamcomp);
    // and Sivia is called recursively
      Sivia(IBTest, xlow, eps);
      Sivia(IBTest, xup, eps);
   }
}
//-------------------------------------------------------
```

To use SIVIA, it now suffices to define the test to be inverted and to write the function `main()`, which will call `Sivia()`:

```
//-------------------------------------------------------
// File:     siviademo.cpp
// Purpose:  first test of Sivia
#include "sivia.h"
#include "func.h"

// Definition of the test
INTERVAL_BOOL IvalBoolTest (const INTERVAL_VECTOR& x) {
  INTERVAL z = Sqr(x(1)) * (Sqr(x(1)) - INTERVAL(4))
                           + Sqr(x(2)) * INTERVAL(4);
  INTERVAL r;  // temporary interval, which will contain
               // the intersection of z and [-0.1,0.1]
  if (z <= INTERVAL(-0.1,0.1)) return IB_TRUE;
  if (Intersection(r, z, INTERVAL(-0.1,0.1)) == 0)
    return IB_FALSE;
  return IB_INDET;
}

void main()
{
  INTERVAL_VECTOR x(2);                    // search box
  x(1) = INTERVAL(-5,5);
  x(2) = INTERVAL(-5,5);
  Sivia(IvalBoolTest, x, 0.05);    // eps is set to 0.05
}
//-------------------------------------------------------
```

This version of SIVIA produces only a console output. A graphic output may be added, but this is beyond the scope of this book. At

`http://www.lss.supelec.fr/books/intervals`

the reader will find a version of SIVIA that writes the solution set in a file that can be imported, e.g., by MATLAB, to be plotted. A figure similar to that on page 62 is then obtained.

Exercise 11.21 *Code the interval Boolean test* `IBTAnnular` *to test a box of* $\mathbb{IR}^2$ *for inclusion in the area* $\mathcal{S}_c$ *between circles centred on the origin and with radii* 1 *and* 2. *Characterize* $\mathcal{S}_c$ *using* `Sivia`. ■

Exercise 11.22 *Code the interval Boolean test* `IBTUpRight` *to test a box of* $\mathbb{IR}^2$ *for inclusion in the set* $\mathcal{S}_q$ *corresponding to the upper right quadrant of* $\mathbb{R}^2$. *Characterize* $\mathcal{S}_q$ *using* `Sivia`. ■

Exercise 11.23 *Code the interval Boolean test* `IBTInt` *to test a box of* $\mathbb{IR}^2$ *for inclusion in the set* $\mathcal{S}_i = \mathcal{S}_c \cap \mathcal{S}_q$, *where* $\mathcal{S}_c$ *and* $\mathcal{S}_q$ *are defined as in Exercises 11.21 and 11.22. Characterize* $\mathcal{S}_i$ *using* `Sivia`. ■

Exercise 11.24 *Evaluate the real and imaginary parts of* $P(j\omega, \mathbf{p})$*, as defined in Example 7.9, page 207. Using* `Sivia`*, prove that the CSP*

$$\mathcal{H} : (P(j\omega, \mathbf{p}) = 0, \mathbf{p} \in [\mathbf{p}],\ \omega \in [0\ rad/s, 24\ rad/s])$$

has no solution and that the polynomial $P(s, \mathbf{p})$ *is robustly stable.* ■

11.9 Interval Matrices

Building a complete `INTERVAL_MATRIX` class is a considerable task, because of the number of member functions needed to implement all the possible interactions with other classes. This is why we shall limit ourselves to giving some hints about how this has been done in PROFIL/BIAS, before presenting some features of this library that will be useful to implement the algorithms described in this book. Examples of such implementations will form the subject of exercises.

The size of a matrix is characterized by its numbers of rows and columns, represented here by the two positive integers `nRows` and `nCols`. For its storage, the matrix is transformed into a one-dimensional array by stacking its rows. The matrix entry located at the `r`th row and `c`th column is stored at the `(r-1)*nCols+(c-1)`th memory location after that designated by the pointer `theElements`. The memory location pointed to by `theElements` is thus used to store the upmost and leftmost entry. The header of the interval matrix library `ivalmat` may be

```
//-------------------------------------------------------
// File:     ivalmat.h
// Purpose:  specification of the INTERVAL_MATRIX class
#include "ivalvec.h"          // to use INTERVAL_VECTORs

class INTERVAL_MATRIX{
  private:
    int nRows, nCols;
    INTERVAL *theElements;
  public:
// constructors
    INTERVAL_MATRIX()                          // default
      {nRows = 0; nCols = 0; theElements = NULL;}
    INTERVAL_MATRIX(int r, int c)          // initialized
      {nRows = r; nCols = c;
       theElements = new INTERVAL[r * c];}
    //...
// destructor
    ~INTERVAL_MATRIX() {delete[] theElements;}
// other member functions
```

```
    // function call operator
      INTERVAL& operator()(int r, int c)
      { return (theElements[(r-1) * nCols + (c-1)]); }
    //...
  // friend functions
    // read-only access functions
      friend int RowDimension(const INTERVAL_MATRIX& x)
        { return x.nRows; }
      friend int ColDimension(const INTERVAL_MATRIX& x)
        { return x.nCols; }
  //...
  };
  //-----------------------------------------------------
```

The default constructor builds an empty matrix. The initialized constructor creates an array of `r*c` intervals beginning at the memory location `theElements`. The destructor frees the allocated memory.

As for interval vectors, the function call operator is used to access a given entry of the matrix. A reference to this entry is returned to allow a possible modification of its value, taking into account how the matrix has been stored. Assignment, addition and subtraction of matrices, multiplication and division by scalars are performed as in the `INTERVAL_VECTOR` class.

Classes for matrices of reals or integers are built according to the same model. From now on, the reader is urged to switch to the matrix classes provided by PROFIL/BIAS.

11.10 Matrices with PROFIL/BIAS

Three matrix classes are provided by PROFIL/BIAS.

Module name	Class name	Description
`matrix`	`MATRIX`	matrix of doubles
`intmat`	`INTEGER_MATRIX`	matrix of integers
`ivalmat`	`INTERVAL_MATRIX`	matrix of intervals

The use of the `INTERVAL_MATRIX` class requires the `bias0`, `bias1` and `bias2` modules (or a library containing them) to be included in the project. The following example illustrates some features of the `INTERVAL_MATRIX` class.

```
//-----------------------------------------------------
// File:    matxdemo.cpp
// Purpose: using matrices
```

```
#include "ivalvec.h"          // INTERVAL_VECTOR library
#include "ivalmat.h"          // INTERVAL_MATRIX library

void main()
{
  INTERVAL_MATRIX a(4,3);  // a has 4 rows and 3 columns
  INTERVAL_VECTOR b(3);
  INTERVAL_VECTOR c;             // c is not initialized

  Initialize(a,INTERVAL(-1,1));// all entries are [-1,1]
  a(2,3) = INTERVAL(-2,3);        // entry is now [-2,3]
  a(3,3) = -2;                   // entry is now [-2,-2]
  c = Row(a,2);                     // access to 2nd row
  c = Col(a,3);                  // access to 3rd column
  b(1) = 1; b(2) = INTERVAL(1,2); b(3) = INTERVAL(4,5);
  c = a * b;          // product of a matrix by a vector

  // console output
  cout << a << " * " << b << " = " << c << endl;
}
//-------------------------------------------------------
```

Many other functions are available. `Inf`, `Sup`, `Mid`, `Diam` respectively compute the lower and upper bounds, the centres and the widths of all the entries of an `INTERVAL_MATRIX`. The results are stored in a `MATRIX`. Useful functions are also provided in the `util` module, such as

- `MATRIX Inverse(MATRIX&);` which returns the inverse of a matrix of `doubles`,
- `MATRIX Transpose(MATRIX&);` which returns the transpose of a matrix of `doubles`,
- `MATRIX Id(INT n);` which returns an `n` $\times$ `n` identity matrix.

These functions are not available for the `INTERVAL_MATRIX` class. The reader is invited to have a look at the self-explanatory header `ivalmat.h` or at the documentation of PROFIL/BIAS.

11.11 Exercises on Interval Matrices

In this section, PROFIL/BIAS will be used to implement some of the tools presented in Section 4.2 to solve CSPs. These tools will be implemented in a new `contractors` module.

Exercise 11.25 (Intervalization of Gauss elimination) *The problem is to find a box enclosure of the solution set of the CSP*

$$\mathcal{H}: \begin{pmatrix} \mathbf{A} \in [\mathbf{A}], \mathbf{b} \in [\mathbf{b}], \mathbf{p} \in [\mathbf{p}] \\ \mathbf{Ap} - \mathbf{b} = \mathbf{0} \end{pmatrix}. \tag{11.4}$$

The Gauss-elimination contractor presented in Section 4.2.2, page 70, attempts to reduce the domain of $\mathbf{p}$. *This involves only basic operations on interval vectors and matrices. The header of the* `contractors` *module may include*

```
//-------------------------------------------------------
// File:     contractors.h
// Purpose:  implementation of some contractors
#include "ivalmat.h"          // to use interval matrices

// contractor based on Gauss elimination
INTERVAL_VECTOR GaussElimination(INTERVAL_MATRIX A,
        INTERVAL_VECTOR b, INTERVAL_VECTOR p, INT& err);
//...
//-------------------------------------------------------
```

1. *Implement* `GaussElimination`, *based on Table 4.3, page 72. Note that this contractor modifies the parameters* `A` *and* `b`. *To prevent this, they may be passed by value. An* `err` *flag may be used to return the error status of the procedure. If a problem has occurred, this flag can be assigned any value but* 0, 0 *indicating that no error has been detected.*
2. *Apply* `GaussElimination` *on the domains*

$$[\mathbf{A}] = \begin{pmatrix} [4,5] & [-1,1] & [1.5,2.5] \\ [-0.5,0.5] & [-7,-5] & [1,2] \\ [-1.5,-0.5] & [-0.7,-0.5] & [2,3] \end{pmatrix}, \tag{11.5}$$

$$[\mathbf{b}] = \begin{pmatrix} [3,4] \\ [0,2] \\ [3,4] \end{pmatrix} \text{ and } [\mathbf{p}] = \begin{pmatrix} [-10,10] \\ [-10,10] \\ [-10,10] \end{pmatrix}. \tag{11.6}$$

3. *Check that this contractor is idempotent.* ■

The fixed-point methods presented in Section 4.2.3, page 72, attempt to solve the CSP

$$\mathcal{H}: (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}]) \tag{11.7}$$

using an algorithm $\boldsymbol{\psi}$ such that

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \Leftrightarrow \mathbf{x} = \boldsymbol{\psi}(\mathbf{x}). \tag{11.8}$$

The next two exercises are devoted to building contractors based on this idea.

Exercise 11.26 (Gauss–Seidel contractor) *The Gauss–Seidel contractor presented in Section 4.2.3, page 73, uses a transformation of the CSP* (11.4) *and evaluates*

$$\operatorname{diag}([\mathbf{A}])^{-1}([\mathbf{b}] - \operatorname{extdiag}([\mathbf{A}])\,[\mathbf{p}]).$$

The algorithm to be coded should build $\operatorname{diag}([\mathbf{A}])^{-1}$*, using the inverse of each diagonal entry of* $[\mathbf{A}]$*;* $\operatorname{extdiag}([\mathbf{A}])$ *is obtained by keeping the non-diagonal entries of* $[\mathbf{A}]$ *and forcing the diagonal entries to zero. The remaining computations involve only simple vector and matrix operations. The header of the* `contractors` *module may be supplemented with*

```
// contractor based on Gauss-Seidel elimination
INTERVAL_VECTOR GaussSeidelIteration(INTERVAL_MATRIX A,
        INTERVAL_VECTOR b, INTERVAL_VECTOR p, INT& err);
```

Again, the err *flag may be used to signal problems, e.g., during the construction of* $\operatorname{diag}([\mathbf{A}])^{-1}$*, due to the presence of an interval containing zero in the diagonal of* $[\mathbf{A}]$*.*

1. *Implement the Gauss–Seidel contractor.*
2. *Apply it on the domains defined by (11.5) and (11.6).*
3. *Iterate application; does the contractor converge?* ■

Exercise 11.27 (Krawczyk contractor) *The Krawczyk contractor has been presented in Section 4.2.3, page 75. Consider the CSP*

$$\mathcal{H} : (\mathbf{f}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in [\mathbf{x}]), \tag{11.9}$$

with $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$*, and*

$$\begin{cases} f_1(x_1, x_2) = x_1^2 - 4x_2, \\ f_2(x_1, x_2) = x_2^2 - 2x_1 + 4x_2, \end{cases} \tag{11.10}$$

and

$$[\mathbf{x}] = [-0.1, 0.1] \times [-0.1, 0.3]. \tag{11.11}$$

The Jacobian matrix for $\mathbf{f}$ *is*

$$\mathbf{J_f} = \begin{pmatrix} 2x_1 & -4 \\ -2 & 2x_2 + 4 \end{pmatrix}. \tag{11.12}$$

Implement the algorithm of Table 4.5, page 76, for this example. The `Inverse` *function of the* `util` *module of* PROFIL/BIAS *may be helpful. Note that building a generic Krawczyk contractor would require the use of pointers to functions taking an interval vector as a parameter and returning an interval vector (for* $[\mathbf{f}]$*) and an interval matrix (for* $[\mathbf{J_f}]$*). The implementation and the use of such pointers are similar to those of pointers to interval Boolean tests described in Section 11.8.* ■

The results obtained by the Gauss-elimination and Gauss–Seidel contractors may be improved by an appropriate preconditioning of the system of equations; see Section 4.3.2, page 75. This is illustrated by the following exercise.

Exercise 11.28 (Preconditioning) *In* (11.4), $[\mathbf{A}]$ *and* $[\mathbf{b}]$ *are multiplied by the preconditioning matrix* $(\mathrm{mid}\,([\mathbf{A}]))^{-1}$ *before applying a contractor.*

1. *Add the two following contractors to the* `contractors` *module:*

```
// contractors using preconditioning
INTERVAL_VECTOR PrecGaussElimination(INTERVAL_MATRIX A,
        INTERVAL_VECTOR b, INTERVAL_VECTOR p, INT& err);
INTERVAL_VECTOR PrecGaussSeidelIteration(INTERVAL_MATRIX A,
        INTERVAL_VECTOR b, INTERVAL_VECTOR p, INT& err);
```

The `Inverse` *function of the* `util` *module of* PROFIL/BIAS *may be used to evaluate* $(\mathrm{mid}\,([\mathbf{A}]))^{-1}$.

2. *Compare the performances obtained with the preconditioned contractors to those obtained without preconditioning.* ■

11.12 Regular Subpavings with PROFIL/BIAS

Regular subpavings as described in Chapter 3 are not supported by PROFIL/BIAS. The aim of this section is thus to implement regular subpavings as basic objects for set inversion and image evaluation.

Subpavings are stored using binary trees. Although such trees are classically used in sorting algorithms and symbolic computation, their implementation as may be found in standard template libraries is not adequate for our purpose and many of the functions needed are not implemented. This is why a specific implementation will be developed.

11.12.1 `NODE` class

The implementation of subpavings is based on the `NODE` class. Each node of the binary tree representing a subpaving is associated with a box of this subpaving. This node also contains pointers to the nodes corresponding to the left and right children of this box. The `NODE` class thus encapsulates a pointer to an `INTERVAL_VECTOR` and two pointers to `NODE`s. A `SUBPAVING` will then be represented by a pointer to a `NODE`. The header of a module implementing the `NODE` class (and therefore `SUBPAVING`s) may be

```
//-------------------------------------------------------
// File:     subpaving.h
// Purpose:  definition of NODE and SUBPAVING
#include "ivalvec.h"          // to use INTERVAL_VECTORs
#include <iostream.h>                // for standard i/o

class NODE;
typedef NODE* SUBPAVING;  // makes SUBPAVING an alias of
                                    // pointer to a NODE
class NODE
{
  private:
    INTERVAL_VECTOR *theBox;
    SUBPAVING leftChild;
    SUBPAVING rightChild;
  public:
// constructors
    NODE()                                        // default
    { theBox = NULL;
      leftChild = NULL; rightChild = NULL; }
    NODE(const INTERVAL_VECTOR& v)        // initialized
    { theBox = new INTERVAL_VECTOR(v);
      leftChild = NULL; rightChild = NULL; }
    NODE(const NODE& n);                            // copy
// destructor
    ~NODE()
    { delete theBox;
      delete leftChild; delete rightChild; }
// friend functions
    friend INTERVAL_VECTOR Box(SUBPAVING a)
    { return (*a->theBox); }
    friend SUBPAVING LeftChild(SUBPAVING a)
    { return a->leftChild; }
    friend SUBPAVING RightChild(SUBPAVING a)
    { return a->rightChild; }
    friend bool IsEmpty(SUBPAVING a)
    { return ((a == NULL) || (a->theBox == NULL)); }
    friend bool IsLeaf(SUBPAVING a)
    { return (!IsEmpty(a) && IsEmpty(a->leftChild)
                          && IsEmpty(a->rightChild)); }
    friend ostream& operator<<(ostream&, SUBPAVING);
};
//-------------------------------------------------------
```

The default constructor of a NODE sets all its properties to NULL. A SUBPAVING pointing to such a node, as created, for example, by

```
SUBPAVING A = new NODE();
```

is considered to be empty. Another useful representation of an empty subpaving is the NULL-valued SUBPAVING, created, for example, by

```
SUBPAVING B = NULL;
```

Although A and B have been created differently, they will behave in the same way in the algorithms to be presented. The initialized constructor defines a NODE that reduces to a leaf and thus contains only an INTERVAL_VECTOR, and two NULL-valued pointers. Finally, the copy constructor presented below has a recursive structure similar to that already used in SIVIA. Such a recursive structure is typical of computation on binary trees.

```
//-------------------------------------------------------
// File:     subpaving.cpp
// Purpose:  implementation of NODE and SUBPAVING
#include "subpaving.h"              // to use SUBPAVINGs

NODE::NODE(const NODE& n)            // copy constructor
{ theBox = new INTERVAL_VECTOR(*n.theBox);
  // recursion on the children
  if (n.leftChild)
    leftChild = new NODE(*n.leftChild);
  else leftChild = NULL;
  if (n.rightChild)
    rightChild = new NODE(*n.rightChild);
  else rightChild = NULL; }
//-------------------------------------------------------
```

The INTERVAL_VECTOR copy constructor is first called to copy theBox. The NODE copy constructor is then recursively called on the nodes associated with the left and right subpavings, if they exist. The destructor, implemented in subpaving.h, shares the same recursive structure: delete leftChild and delete rightChild implicitly call the destructor on the left and right subpavings, after releasing the memory allocated to theBox.

The friend functions listed in subpaving.h allow access to some properties of SUBPAVINGs. Box, LeftChild and RightChild provide access to the properties of the subpaving passed as their argument. IsEmpty and IsLeaf respectively test whether a subpaving is empty and whether it corresponds to a leaf.

Exercise 11.29 *Implement* operator<< *to provide a basic output to streams for* SUBPAVINGs. *Only boxes corresponding to leaves may be inserted.* ■

11.12.2 Set inversion with subpavings

SIVIA (Section 3.4.1, page 55) may be considered as a basic function manipulating subpavings; this is why it will be implemented as a friend of the `NODE` class. The header `subpaving.h` may thus be supplemented as follows:

```
//-------------------------------------------------------
// File:     subpaving.h (continued)
// Purpose:  definition of NODE and SUBPAVING
//           and of functions manipulating subpavings
//...                          previously included modules
#include "util.h"              // to use Lower and Upper

// define type "interval Booleans"
typedef enum{IB_TRUE, IB_FALSE, IB_INDET} INTERVAL_BOOL;
// define type "pointer to an interval Boolean test"
typedef INTERVAL_BOOL (*PIBT)(const INTERVAL_VECTOR&);
//...
class NODE
{
//...
    friend SUBPAVING Sivia (PIBT, SUBPAVING, double);
};
// utility function
  double MaxDiam (INTERVAL_VECTOR&, int&);

//-------------------------------------------------------
```

This version of SIVIA returns a subpaving, contrary to the version of page 327. The two versions can coexist, as C++ allows function overloading. Depending on the types of the arguments, one or the other function will be called. An example of a call to SIVIA for subpavings is

```
SUBPAVING X = Sivia(Name_of_the_test, S, eps);
```

where `S` is a predefined search subpaving, `Name_of_the_test` is an interval Boolean test defining the set to be characterized (see Section 11.8, page 327) and `eps` is the precision factor. The solution subpaving `X` is returned.

`MaxDiam` can be implemented in `subpaving.cpp` in the same manner as in Section 11.8 (only the set-inversion algorithm differs):

```
//-------------------------------------------------------
// File:     subpaving.cpp (continued)
//...
SUBPAVING Sivia (PIBT IBTest, SUBPAVING S, double eps)
{
  if (IsEmpty(S)) return NULL;
```

```
  int test = IBTest(Box(S));
  int maxdiamcomp;

  if (test == IB_FALSE)
    return NULL;
  if ((test == IB_TRUE)
            || (MaxDiam(Box(S),maxdiamcomp) < eps))
    return new NODE(*S);
  if (IsLeaf(S))
    Expand(S, maxdiamcomp);
  return Reunite(Sivia(IBTest, LeftChild(S), eps),
            Sivia(IBTest, RightChild(S), eps), Box(S));
// Expand and Reunite to be presented
// in the next two subsections
}
//-------------------------------------------------------
```

SUBPAVINGs allow an implementation of SIVIA that closely follows the description of Section 3.4.1, page 55. Notice, however, that only an outer subpaving is computed here, and that some statements have been added to improve efficiency.

If the search subpaving `S` is empty, `Sivia` returns an empty subpaving. Else, a user-defined interval Boolean test is evaluated on `Box(S)`, the box corresponding to the root of `S`. If the test holds `IB_FALSE`, an empty subpaving is returned. If it holds `IB_TRUE` or if the root box is deemed small enough, a subpaving corresponding to the copy of `S` is returned. If `S` is a leaf, `Expand` transforms it into a non-minimal subpaving containing two boxes resulting from the bisection of `Box(S)` across its first component of maximum width. `Sivia` is then recursively called on the left and right children of `S`. The two resulting subpavings are finally merged (by `Reunite`) in order to get a minimal subpaving.

As implemented here, `Sivia` modifies `S`. This could easily be avoided, if needed. Let us now turn to the functions `Expand` and `Reunite`, which are used in `Sivia`.

Expanding a subpaving. `Expand` grafts two sibling leaves to a node, provided that this node is degenerate and thus corresponds to a leaf. As this modifies private properties of `NODE`s, `Expand` is implemented as a friend of the `NODE` class, which requires inserting

```
friend void Expand(SUBPAVING S, int comp);
```

in `subpaving.h`. The code for `expand` is as follows:

```
//-------------------------------------------------------
// File:     subpaving.cpp (continued)
//...
```

```
void Expand(SUBPAVING S, int comp)
{
  if (!IsLeaf(S)) return;
  S->leftChild = new NODE(Lower(Box(S), comp));
  S->rightChild = new NODE(Upper(Box(S), comp));
}
//-------------------------------------------------------
```

If the subpaving `S` to be expanded is not a leaf, it is left as is. If it is a leaf, two children are added. Their boxes result from the bisection of `Box(S)` across the dimension `comp`.

Reuniting subpavings. `Reunite` computes a minimal subpaving from two sibling subpavings. It may be implemented as follows:

```
//-------------------------------------------------------
// File:     subpaving.cpp (continued)
//...
SUBPAVING Reunite(SUBPAVING lChild, SUBPAVING rChild,
                                    INTERVAL_VECTOR& x)
{
  if (IsEmpty(lChild) && IsEmpty(rChild)) return NULL;

  SUBPAVING result = new NODE(x); // resulting subpaving

  if (IsLeaf(lChild) && IsLeaf(rChild))
  { delete lChild; delete rChild; return result; }

  result->leftChild = lChild;
  result->rightChild = rChild;
  return result;
}
//-------------------------------------------------------
```

Implementation closely follows the presentation of Section 3.3.3, page 52. An empty subpaving is returned if both subpavings to be reunited are empty. If both are leaves, they are destroyed and the parent node (which has thus become a leaf) is returned. In all other cases, both subpavings are attached to their parent node.

Exercise 11.30 *Code a function* `NbLeaves`*, evaluating the number of leaves of a* `SUBPAVING`*.* ■

Exercise 11.31 *Code a function* `Volume`*, summing the volumes of all the leaves of a* `SUBPAVING`*.* ■

The aim of the next exercise is to allow the evaluation by SIVIA of an outer approximation of $\mathbf{f}^{-1}\left(\overline{\mathbb{Y}}\right)$ for any regular subpaving $\overline{\mathbb{Y}}$. This involves testing whether $[\mathbf{f}]\left([\mathbf{x}]\right) \subset \overline{\mathbb{Y}}$, *i.e.*, whether a box belongs to a subpaving.

Exercise 11.32 *Using the algorithm* INSIDE *of Section 3.3.3, page 52, implement*

```
friend int operator<=(const INTERVAL_VECTOR&, SUBPAVING);
```

to test whether an `INTERVAL_VECTOR` *is included in a* `SUBPAVING`. ■

Exercise 11.33 *This exercise is intended to illustrate the possibility of using* SIVIA *to evaluate the* direct *image of a set by a function, provided that this function is invertible in the usual sense.*

1. *Compute a subpaving* $\overline{\mathbb{S}}_c$ *containing the set* $\mathcal{S}_c$ *of Exercise 11.21. Evaluate the volume of this subpaving.*
2. *Use* `Sivia` *to compute a subpaving* $\overline{\mathbb{S}}_{c1}$ *containing* $\mathbf{f}(\overline{\mathbb{S}}_c)$*, where*

   $$\mathbf{f} : (x_1, x_2)^T \mapsto (2x_1 - x_2, -x_1 + 2x_2)^T .$$

   Note that $\mathbf{f}$ *is invertible in the usual sense, with inverse*

   $$\mathbf{f}^{-1} : (x_1, x_2)^T \mapsto \left(\tfrac{2}{3}x_1 + \tfrac{1}{3}x_2, \tfrac{1}{3}x_1 + \tfrac{2}{3}x_2\right)^T ,$$

   so

   $$\left\{ \mathbf{f}(x_1, x_2) \;\middle|\; (x_1, x_2) \in \overline{\mathbb{S}}_c \right\} = \left\{ (x_1, x_2) \;\middle|\; \mathbf{f}^{-1}(x_1, x_2) \in \overline{\mathbb{S}}_c \right\} .$$

   Thus, to compute $\overline{\mathbb{S}}_{c1}$*, it suffices to perform the set-inversion of* $\overline{\mathbb{S}}_c$ *by* $\mathbf{f}^{-1}$*.*
3. *Compute a subpaving* $\overline{\mathbb{S}}_{c2}$ *containing* $\mathbf{f}^{-1}\left(\overline{\mathbb{S}}_{c1}\right)$*. This subpaving would be equal to* $\overline{\mathbb{S}}_c$ *if no pessimism were introduced. Evaluate the volume of* $\overline{\mathbb{S}}_{c2}$ *and compare it to the volume of* $\overline{\mathbb{S}}_c$*. Evaluate the influence of the precision parameter* `eps` *on pessimism.* ■

11.12.3 Image evaluation with subpavings

IMAGESP (Section 3.4.2, page 59) performs basic subpaving manipulations. It will thus also be implemented as a friend of the `NODE` class. The header of the `subpaving` module should therefore be supplemented as follows:

```
//-------------------------------------------------------
// File:     subpaving.h (continued)
// Purpose:  definition of NODE and SUBPAVING
//           and of functions manipulating subpavings
//...                          previously included modules
#include "imlist.h"                  // to manage lists of
                                       // INTERVAL_VECTORS
//...
// defines type "pointer to an INTERVAL_VECTOR function"
typedef INTERVAL_VECTOR (*PIVF)(const INTERVAL_VECTOR&);
//...
class NODE
```

```
{
//...
    // friend functions for the basic steps of ImageSp
    friend void Mince(SUBPAVING, double);
    friend void Evaluate(PIVF, SUBPAVING, IMAGELIST&,
                                        INTERVAL_VECTOR&);
    friend SUBPAVING Regularize(INTERVAL_VECTOR&,
                                      IMAGELIST&, double);
    // ImageSp
    friend SUBPAVING ImageSp(PIVF, SUBPAVING, double);
};
//...
//-------------------------------------------------------
```

This modified header implements `PIVF`, a new type of pointer to an interval vector function that will be employed to point to the vector inclusion function to be used by IMAGESP. The header of this function should look like

```
INTERVAL_VECTOR Name_of_function(const INTERVAL_VECTOR&);
```

Once this function has been properly defined, an outer approximation `Y` of the image of a regular subpaving `X` can be obtained as follows:

```
SUBPAVING Y = ImageSp(Name_of_function, X, eps);
```

where `eps` is the predefined precision factor.

Exercise 11.34 *Implement an interval vector function* `f` *corresponding to* $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$ *defined by*

$$f_1(x_1, x_2) = x_1^2 + x_2 - x_2^2,$$
$$f_2(x_1, x_2) = x_1^2 + x_2^2.$$

The implementation may involve the natural inclusion function or any centred form. ■

The modified header also contains friend functions corresponding to the three basic steps of IMAGESP, namely mincing, evaluation and regularization. Before considering the implementation of these three steps, we shall explain how the lists of boxes involved in IMAGESP can be managed.

List management. We developed the `imlist` module, based on the `veclist` module of PROFIL/BIAS, to provide a new class `IMAGELIST` to store and update the lists of boxes involved in IMAGESP. Any instance of `IMAGELIST` contains `IMAGELIST_ELEMENT`s. The properties of each of these elements are `Box`, an `INTERVAL_VECTOR`, and `Volume`, a `double` corresponding to the volume of `Box`. `IMAGELIST_ELEMENT`s are sorted by decreasing volume of their `Box`.

To use the IMAGELIST class for ImageSp, only one property and a few member functions are needed. This is why we shall limit ourselves here to listing them and refer the reader to the information about veclist provided in the documentation of profil/bias for more details.

The initialized constructor of list elements is

```
IMAGELIST_ELEMENT(INTERVAL_VECTOR&);
```

The property

```
IMAGELIST_ELEMENT* current;
```

points to the element of the list that is currently active.
A box can be inserted in the list using

```
IMAGELIST& operator+=(INTERVAL_VECTOR&);
```

The first element of the list is accessed using

```
IMAGELIST_ELEMENT& First(IMAGELIST&);
```

This function updates the current property, making this first element active.

```
IMAGELIST_ELEMENT& Next(IMAGELIST&);
```

gives access to the element stored after the currently active element of the list and makes it active by updating the current property.

```
INT IsEmpty(IMAGELIST&);
```

tests whether a list is empty.

```
INT Finished(IMAGELIST&);
```

tests whether the last element of a list has been reached.

Remark 11.10 *The* list *module provided by the* standard template library *cannot be used, due to an incompatibility with the* INTERVAL_VECTOR *class.* ■

Mincing, evaluation and regularization will now be considered in turn.

Mincing. The first step of ImageSp will be performed by the Mince function, which transforms a minimal subpaving into a non-minimal one consisting of boxes with widths lower than the precision parameter eps. Mince may be implemented as follows:

```
//-------------------------------------------------------
// File:     subpaving.cpp (continued)
//...
void Mince(SUBPAVING A, double eps)
{
  if (IsEmpty(A)) return;
  if (IsLeaf(A))
```

```
  { int comp;
    if (MaxDiam(Box(A), comp) < eps) return;
    else Expand(A, comp);
  }

  Mince(A->leftChild, eps);
  Mince(A->rightChild, eps);
}
//...
//-------------------------------------------------------
```

If the subpaving to be minced is empty, nothing is done. If it is a leaf and the corresponding box is larger than `eps`, then the leaf is expanded. If the subpaving is not a leaf, then `Mince` is recursively applied to its left and right children.

Evaluation. The images of all the boxes generated by `Mince` must now be evaluated. `Evaluate` computes the list of these images and the interval hull (`hull`) of the union of all the boxes listed in the resulting image list. Implementation may be as follows:

```
//-------------------------------------------------------
// File:     subpaving.cpp (continued)
//...
void Evaluate(PIVF f, SUBPAVING A, IMAGELIST& list,
                                   INTERVAL_VECTOR& hull)
{
  if (IsEmpty(A)) return;
  if (IsLeaf(A))
  {
    INTERVAL_VECTOR image(f(Box(A)));
  // computation of interval hull
    if (IsEmpty(list)) hull = image;
    else hull = Hull(hull, image);
  // images are stored in list
    list += IMAGELIST_ELEMENT(image);
    return;
  }

  Evaluate(f, A->leftChild, list, hull);
  Evaluate(f, A->rightChild, list, hull);
}
//...
//-------------------------------------------------------
```

Regularization. All these image boxes must now be merged into a single subpaving. This may be performed as follows:

```
//-------------------------------------------------------
// File:     subpaving.cpp (continued)
//...
SUBPAVING Regularize (INTERVAL_VECTOR& hull,
                              IMAGELIST& list, double eps)
{
  if (IsEmpty(list)) return NULL;
  if (hull == First(list).Box) return new NODE(hull);
  int maxdcomp;
  if (MaxDiam(hull,maxdcomp) < eps)
    return new NODE(hull);

  INTERVAL_VECTOR lefthull = Lower(hull, maxdcomp);
  INTERVAL_VECTOR righthull = Upper(hull, maxdcomp);
  IMAGELIST leftlist,rightlist;
  INTERVAL_VECTOR inter;
  while (!Finished(list))
  {
    if (Intersection(inter,Current(list).Box,lefthull))
       leftlist += IMAGELIST_ELEMENT(inter);
    if (Intersection(inter,Current(list).Box,righthull))
       rightlist += IMAGELIST_ELEMENT(inter);
    Next(list);
  }

  return Reunite(Regularize(lefthull, leftlist, eps),
           Regularize(righthull, rightlist, eps), hull);
}
//-------------------------------------------------------
```

The box `hull` is the interval hull of the union of all the boxes listed in `list`. These boxes are to be merged into a subpaving, the root of which will correspond to the box `hull`. If `list` is empty, an empty subpaving is returned. If the largest box of the list, *i.e.*, the box stored in its first element, is equal to `hull`, then the subpaving reduces to this box. If the width of `hull` is lower than the precision parameter `eps`, then `list` contains only boxes with width lower than `eps`, thus the regularized subpaving reduces to `hull`. This allows the subpaving to be created after a finite number of recursions. It only contains boxes with width lower than `eps`.

If none of these tests is satisfied, `hull` is bisected into `lefthull` and `righthull` and two corresponding lists (`leftlist` and `rightlist`) are generated, which respectively contain the intersections of all elements of `list` with `lefthull` and `righthull`. `Regularize` is then recursively applied on these two boxes and the corresponding lists.

ImageSp can now be written very concisely:

```
//-----------------------------------------------------
// File:     subpaving.cpp (continued)
//...
SUBPAVING ImageSp(PIVF f, SUBPAVING A, double eps)
{
  IMAGELIST images;
  INTERVAL_VECTOR hull;
  Mince(A,eps);
  Evaluate(A, f, images, hull);
  return (Regularize(hull, images, eps));
}
//-----------------------------------------------------
```

Exercise 11.35 *Evaluate the image of the box* $[-2,2]^2$ *by the function defined in Exercise 11.34 with the help of* `ImageSp`. ■

11.12.4 System simulation and state estimation with subpavings

Four exercises will bring the reader from simulation to joint parameter and state estimation.

Exercise 11.36 *Consider the discrete-time system*

$$\begin{pmatrix} x_1(k+1) \\ x_2(k+1) \end{pmatrix} = \begin{pmatrix} \rho\cos(\pi/4) & -\rho\sin(\pi/4) \\ \rho\sin(\pi/4) & \rho\cos(\pi/4) \end{pmatrix} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix}, \tag{11.13}$$

where $\rho = 0.85$. *At step* $k = 0$, $(x_1(0), x_2(0))^T$ *is only known to belong to* $\mathbb{X}_0 = [4,5] \times [-1,1]$. *The problem is to evaluate the set* $\mathbb{X}_k$ *containing all the possible values of* $(x_1(k), x_2(k))^T$ *at any given time* $k > 0$. *With the help of* IMAGESP, *code an algorithm evaluating a subpaving* $\overline{\mathbb{X}}_k$ *guaranteed to contain* $\mathbb{X}_k$ *for* $k = 1, \ldots, 10$. *The precision of the description will be controlled by the value given to* `eps` *by the user. Evaluate the influence of* `eps` *on computing time and on the quality of description, assessed by comparing the volumes of the subpavings computed. All subpavings should be written to a file (e.g., to* `output.spv`*). Two-dimensional projections of these subpavings may then be plotted using*

```
DrawSpaving.exe
```

available at

```
http://www.lss.supelec.fr/books/intervals
```

■

Exercise 11.37 *Modify the code written for Exercise 11.36 to answer the same questions assuming now that the parameter ρ in the drift matrix of (11.13) is uncertain and only known to belong to the interval $[\rho] = [0.8, 0.9]$, so that (11.13) becomes*

$$\begin{pmatrix} x_1(k+1) \\ x_2(k+1) \end{pmatrix} = \begin{pmatrix} [\rho]\cos(\pi/4) & -[\rho]\sin(\pi/4) \\ [\rho]\sin(\pi/4) & [\rho]\cos(\pi/4) \end{pmatrix} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix}. \quad (11.14)$$

■

Exercise 11.38 *Assume now that measurements $y(k)$, $k = 0, \ldots, 10$, provide information about the state of the uncertain system described by (11.14), according to the observation equation*

$$y(k) = x_1(k) + w_k, \quad (11.15)$$

where w_k is a bounded measurement noise belonging to $[-0.1, 0.1]$. The purpose of this exercise is to implement a state estimator for this system, alternating prediction and correction steps as explained in Section 6.4, page 168.

1. *Implement the prediction step using* `ImageSp`.
2. *Implement the correction step using* `Sivia`.
3. *Generate data $y(k)$, $k = 0, \ldots, 10$, by simulating (11.14) with $[\rho]$ replaced at each time instant k by ρ_k picked at random in $[0.8, 0.9]$, with w_k picked at random in $[-0.1, 0.1]$ and with $x_1(0) = 4.5$ and $x_2(0) = 0$.*
4. *Evaluate a subpaving $\overline{\mathbb{X}}_k$ containing $\mathbb{X}_k$ for $k = 0, \ldots, 10$, assuming that the initial state is only known to belong to $\mathbb{X}_0 = [4, 5] \times [-1, 1]$ and that the data are those obtained as the result of the previous question.* ■

Exercise 11.39 *Consider the same problem as in Exercise 11.38, but assume now that the unknown parameter ρ is constant and can be appended to the state vector to form an extended state vector. The aim of this exercise is to build a set estimator for this extended state vector to perform joint state and parameter estimation.*

1. *Write the discrete-time state equation satisfied by the extended state vector.*
2. *Generate data $y(k)$, $k = 0, \ldots, 10$, by simulating (11.14) with $[\rho]$ replaced by $\rho^* = 0.85$, with w_k picked at random in $[-0.1, 0.1]$ and with $x_1(0) = 4.5$ and $x_2(0) = 0$.*
3. *Use the same method as in Exercise 11.38 to estimate the extended state vector from the data thus generated.*
4. *Modify the code for the simulation of the data and for the estimation of the extended state vector to allow the unknown parameter ρ to vary according to*

$$\rho_k = \rho_{k-1} + \delta_k,$$

where δ_k belongs to $[-0.01, 0.01]$ and ρ_0 belongs to $[0.8, 0.9]$. ■

11.13 Error Handling

Many strategies may be considered in the presence of errors, from burying one's head in the sand to aborting the program within which the error took place. The best solution usually lies between these two extremes, as illustrated by the next two examples.

Example 11.1 *Consider an interval Newton routine to find all the zeros of* $f(x) = \tan(x) - x$ *over* $[-10\pi, 10\pi]$. *The function* tan *is not defined on the entire search interval. This is no reason for not running the algorithm. In such a case, one would rather take* $[f]([-10\pi, 10\pi]) = [-\infty, +\infty]$, *to indicate that* $[-10\pi, 10\pi]$ *may contain zeros of* $f(x)$. ■

Example 11.2 *Assume now that the function* f *of the previous example is only an intermediary result in long and costly computations, and that the propagation of the interval* $[-\infty, +\infty]$ *in these computations leads to useless results. One may then prefer to abort the procedure as soon as possible to save time and money, and facilitate the detection of the problem.* ■

The measures to be taken when a given error is detected thus depend on the context. Only those errors that call for a systematic action can be treated inside a library. The others should only be detected, the responsibility for the measures to be taken resting with the user. We shall limit ourselves to presenting simple options.

11.13.1 Using `exit`

This is the simplest way to handle errors, but also the least flexible. The function `exit()` belongs to the `stdlib` module. It has already been used in Section 11.6.2, page 321. In the following code, `exit()` is used whenever an addition of two vectors with differing sizes is attempted:

```
//-------------------------------------------------------
// File:     ivalvec.cpp (continued)
//...
INTERVAL_VECTOR operator+(const INTERVAL_VECTOR& a,
                          const INTERVAL_VECTOR& b)
{
  if (a.nElements != b.nElements)
  // the program is aborted
  { cout << "Attempt to add vectors of differing sizes"
         << endl;
    exit(EXIT_FAILURE); }
  // no error is encountered, the addition is performed
  INTERVAL_VECTOR res(a.nElements);
  for (int i = 1; i <= a.nElements; i++)
```

```
      res(i) = a(i) + b(i);
    return (res);
}
//...
//-------------------------------------------------------
```

A message is sent to the standard output stream (the code could be modified to insert the message, for example, in a `.log` file), and the program is aborted. The parameter `EXIT_FAILURE` signals to the operating system that the program has terminated with an error. Many C++ libraries handle errors in this way. Sometimes, flags can be set at compile time in order to adjust the sensitivity of the code to errors. For example, if the flag `__BIASRAISEDIVIDEBYZERO__` is set, PROFIL/BIAS aborts the program when a division by an interval containing 0 is encountered. If the flag is not set, the entire real line is returned.

11.13.2 Exception handling

A C++ *exception* causes the function where the error occurred to be exited, but may allow the program to be continued, contrary to `exit`. Consider again protection against the addition of vectors with differing sizes. The previous code could be replaced by the following one, where an exception is *thrown* whenever this error occurs.

```
//-------------------------------------------------------
// File:     ivalvec.cpp (continued)
//...
INTERVAL_VECTOR operator+(INTERVAL_VECTOR& a,
                                      INTERVAL_VECTOR& b)
{
  if (a.nElements != b.nElements)
  // an exception is thrown
    throw "Attempt to add vectors of differing sizes";
  // no exception thrown, so addition can be performed
  INTERVAL_VECTOR res(a.nElements);
  for (int i = 1; i <= a.nElements; i++)
    res(i) = a(i) + b(i);
  return (res);
}
//...
//-------------------------------------------------------
```

The argument of `throw` may be a string, an integer, or even a class that may contain much more information than just a string. As soon as an exception is thrown, the execution of the function ends. The calling function should *catch* the exception, by using `try` and `catch` blocks, as in

```
//-------------------------------------------------------
// File:     exceptdemo.cpp
// Purpose:  demonstrates how an exception is caught
#include "ivalvec.h"          // to use INTERVAL_VECTORs
#include <iostream.h>                    // standard i/o

void main()
{
  INTERVAL_VECTOR x(3),y(2),z;

  x(1) = INTERVAL(2,3); x(2) = INTERVAL(-3,4);
  x(3) = INTERVAL(-6,-5);
  y(1) = INTERVAL(2,3); y(2) = INTERVAL(-3,4);

  try {       // if an exception is thrown in this block
    z = x + y;
  }                              // then it will be caught
  catch (char *msg) {  // msg contains the string thrown
    cout << msg << endl;
    return;
  }
  // other functions
//...
//-------------------------------------------------------
```

Functions that may throw exceptions are placed into a `try` block. If an exception is thrown, the `catch` block is executed. In this simple illustrative example, computation is terminated exactly as it would have been using `exit()`, but nothing forbids inclusion of a more sophisticated treatment in the `catch()` block.

11.13.3 Mathematical errors

This is probably the most frequent type of error in our context. How should a routine react, for instance, when asked to evaluate the logarithm of an interval with a negative lower bound? Most libraries for interval computations use one of the two options presented, namely exiting or throwing an exception, which is far from being always satisfactory.

The representation of intervals proposed in Section 10.3 makes it possible to reduce the number of cases where a numerical error requires a specific treatment, by introducing infinite and void intervals and by using a set-theoretic stand point for interval computation.

References

Ackermann, J. (1992). Does it suffice to check a subset of multilinear parameters in robustness analysis?, *IEEE Transactions on Automatic Control* **37**(4): 487–488.

Ackermann, J., Hu, H. and Kaesbauer, D. (1990). Robustness analysis: a case study, *IEEE Transactions on Automatic Control* **35**(3): 352–356.

Adrot, O. (2000). *Diagnostic à base de modèles incertains utilisant l'analyse par intervalles : l'approche bornante*, PhD dissertation, Institut National Polytechnique de Lorraine, Nancy, France.

Amato, F., Garofalo, F., Glielmo, L. and Pironti, A. (1995). Robust and quadratic stability via polytopic set covering, *International Journal of Robust and Nonlinear Control* **5**(8): 745–756.

Anderson, B. D. O., Jury, E. I. and Mansour, M. (1987). On robust Hurwitz polynomials, *IEEE Transactions on Automatic Control* **32**(10): 909–913.

Anderson, P. and Anderson, G. (1998). *Navigating C++ and Object-Oriented Design*, Prentice Hall, Upper Saddle River, NJ.

Arsouze, J., Ferrand, G. and Lallouet, A. (2000). Chaotic iterations for constraint propagation and labeling, *Research Report RR-LIFO-2000-04*, LIFO, Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, BP 6759, F-45067 Orléans Cedex 2, France. Available at: ftp://ftp-lifo.univ-orleans.fr/pub/RR/RR2000/RR2000-04.ps.

Balakrishnan, V., Boyd, S. and Balemi, S. (1991a). Branch and bound algorithm for computing the minimum stability degree of parameter-dependent linear systems, *International Journal of Robust and Nonlinear Control* **1**(4): 295–317.

Balakrishnan, V., Boyd, S. and Balemi, S. (1991b). Computing the minimum stability degree of parameter-dependent linear systems, *in* S. P. Bhattacharyya and L. H. Keel (eds), *Control of Uncertain Dynamic Systems*, CRC Press, Boca Raton, FL, pp. 359–378.

Barmish, B. R. (1984). Invariance of the strict Hurwitz property for polynomials with perturbed coefficients, *IEEE Transactions on Automatic Control* **29**(10): 935–936.

Barmish, B. R. (1988). New tools for robustness analysis, *Proceedings of the 27th IEEE Conference on Decision and Control*, Austin, TX, pp. 399–408.

Barmish, B. R. (1994). *New Tools for Robustness of Linear Systems*, MacMillan, New York, NY.

Barmish, B. R. and Tempo, R. (1995). On mappable nonlinearities in robustness analysis, *Proceedings of the 3rd European Control Conference*, Rome, Italy, pp. 1430–1435.

Bartlett, A. C., Hollot, C. V. and Huang, L. (1988). Root locations of an entire polytope of polynomials: It suffices to check the edges, *Mathematics of Control, Signals and Systems* **1**(1): 61–71.

Belforte, G., Bona, B. and Cerone, V. (1990). Parameter estimation algorithms for a set-membership description of uncertainty, *Automatica* **26**(5): 887–898.

Benhamou, F. and Goualard, F. (2000). Universally quantified interval constraints, *in* R. Dechter (ed.), *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming - CP 2000*, Vol. 1894 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany.

Benhamou, F., Goualard, F., Granvilliers, L. and Puget, J. F. (1999). Revising hull and box consistency, *Proceedings of the International Conference on Logic Programming*, Las Cruces, NM, pp. 230–244.

Benhamou, F. and Granvilliers, L. (1997). Automatic generation of numerical redundancies for nonlinear constraint solving, *Reliable Computing* **3**(3): 335–344.

Benhamou, F., McAllester, D. and van Hentenryck, P. (1994). CLP (intervals) revisited, *in* M. Bruynooghe (ed.), *Proceedings of the International Logic Programming Symposium*, Ithaca, NY, pp. 124–138.

Berger, J. (1985). *Statistical Decision Theory and Bayesian Analysis,* 2nd edition, Springer-Verlag, New York, NY.

Berger, M. (1979). *Espaces euclidiens, triangles, cercles et sphères*, Vol. 2 of *Géométrie*, Cedic/Fernand Nathan, Paris, France.

Bertsekas, D. P. and Rhodes, I. B. (1971). Recursive state estimation for a set-membership description of uncertainty, *IEEE Transactions on Automatic Control* **16**(2): 117–128.

Berz, M. and Makino, K. (1998). Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models, *Reliable Computing* **4**(4): 361–369.

Bialas, S. (1983). A necessary and sufficient condition for the stability of interval matrices, *International Journal of Control* **37**(4): 717–722.

Bialas, S. (1985). A necessary and sufficient condition for the stability of convex combinations of stable polynomials or matrices, *Bulletin of the Polish Academy of Sciences* **33**: 473–480.

Bialas, S. and Garloff, J. (1985). Convex combinations of stable polynomials, *Journal of the Franklin Institute* **319**(3): 373–377.

Bischof, C., Carle, A., Corliss, G. F., Griewank, A. and Hovland, P. (1992). ADIFOR – Generating derivative codes from Fortran programs, *Scientific Programming* **1**(1): 11–29.

Bischof, C. H. (1991). Issues in parallel automatic differentiation, *in* A. Griewank and G. F. Corliss (eds), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA, pp. 100–113.

Blondel, V. and Tsitsiklis, J. (1995). NP-hardness of some linear control design problems, *Proceedings of the 34th IEEE Conference on Decision and Control*, New Orleans, LA, pp. 2910–2915.

Borenstein, J., Everett, H. and Feng, L. (1996). *Navigating Mobile Robots*, A. K. Peters Ltd., Wellesley, MA.

Boyd, S., Ghaoui, L. E., Feron, E. and Balakrishnan, V. (1994). *Linear Matrix Inequalities in System and Control Theory*, Vol. 15 of *Studies in Applied Mathematics*, SIAM, Philadelphia, PA.

Braatz, R., Young, P., Doyle, J. and Morari, M. (1994). Computational complexity of μ calculation, *IEEE Transactions on Automatic Control* **39**(5): 1000–1002.

Braems, I., Berthier, F., Jaulin, L., Kieffer, M. and Walter, E. (2001). Guaranteed estimation of electrochemical parameters by set inversion using interval analysis, *Journal of Electroanalytical Chemistry* **495**(1): 1–9.

Brayton, R. K., Director, S. W., Hachtel, G. D. and Vidigal, L. M. (1979). A new algorithm for statistical circuit design based on quasi-Newton methods and function splitting, *IEEE Transactions on Circuits and Systems* **26**(9): 784–794.

Brooks, R. A. and Lozano-Pérez, T. (1985). A subdivision algorithm in configuration space for findpath with rotation, *IEEE Transactions on Systems Man and Cybernetics* **15**(2): 224–233.

Capper, D. M. (1994). *C++ for Scientists, Engineers and Mathematicians*, Springer-Verlag, London, UK.

Castellanos, J. A., Monteil, J., Neira, J. and Tardos, J. (1999). The SMAP: A probabilistic framework for simultaneous localization and map building, *IEEE Transactions on Robotics and Automation* **15**(5): 948–952.

Caviness, B. F. and Johnson, J. R. (eds) (1998). *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Springer-Verlag, Vienna, Austria.

Cerone, V. (1991). Parameter bounds for models with bounded errors in all variables, *Preprints of the 9th IFAC/IFORS Symposium on Identification and System Parameter Estimation*, Budapest, Hungary, pp. 1518–1523.

Cerone, V. (1996). Errors-in-variables models in parameter bounding, *in* M. Milanese, J. Norton, H. Piet-Lahanier and E. Walter (eds), *Bounding Approaches to System Identification*, Plenum, New York, NY, pp. 289–306.

Chernousko, F. L. (1994). *State Estimation for Dynamic Systems*, CRC Press, Boca Raton, FL.

Chiriaev, D. and Walster, G. W. (1998). Interval arithmetic specification. Available at: http://www.mscs.edu/globsol/walster-papers.html.

Cleary, J. C. (1987). Logical arithmetic, *Future Computing Systems* **2**(2): 125–149.

Clément, T. and Gentil, S. (1990). Recursive membership set estimation for output-error models, *Mathematics and Computers in Simulation* **32**(5–6): 505–513.

Clowes, M. B. (1971). On seeing things, *Artificial Intelligence* **2**: 179–185.

Collin, Z., Dechter, R. and Katz, S. (1991). On the feasibility of distributed constraint satisfaction, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, pp. 318–324.

Connell, A. and Corless, R. (1993). An experimental interval arithmetic package in Maple, *Interval Computation* **2**: 120–134.

Corliss, G. F. (1988). Applications of differentiation arithmetic, *in* R. E. Moore (ed.), *Reliability in Computing*, Academic Press, London, UK, pp. 127–148.

Corliss, G. F. (1991). Proposal for a basic interval arithmetic subroutines library (BIAS), Technical Report, Department of Mathematics, Statistics, and Computer Science, Marquette University, Milwaukee, WI.

Corliss, G. F. (1992). Automatic differentiation bibliography, Technical Memorandum ANL/MCS–TM–167, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

Crowley, J. (1989). World modeling and position estimation for a mobile robot using ultrasonic ranging, *Proceedings of the IEEE International Conference on Robotics and Automation*, Scottsdale, AZ, pp. 674–680.

Crowley, J. L., Wallner, F. and Schiele, B. (1998). Position estimation using principal components of range data, *Proceedings of the IEEE International Conference on Robotics and Automation*, Leuven, Belgium, pp. 3121–3128.

Daumas, M., Mazenc, C., Merrheim, X. and Muller, J. M. (1995). Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions, *Journal of Universal Computer Science* **1**(3): 162–175.

Davis, E. (1987). Constraint propagation with interval labels, *Artificial Intelligence* **32**(3): 281–331.

Dechter, A. and Dechter, R. (1987). Removing redundancies in constraint networks, *Proceedings of the 6th AAAI National Conference on Artificial Intelligence*, Vol. 1, Seattle, WA, pp. 105–109.

Dechter, R. and Pearl, J. (1989). Tree-clustering for constraint networks, *Artificial Intelligence* **38**(3): 353–366.

Deller, J., Nayeri, M. and Odeh, S. (1993). Least-square identification with error bounds for real-time signal processing and control, *Proceedings of the IEEE* **81**(6): 815–849.

Deo, N. (1974). *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ.

Deville, Y., Barette, O. and van Hentenryck, P. (1997). Constraint satisfaction over connected row convex constraints, *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 1, Nagoya, Japan, pp. 403–410.

Didrit, O. (1997). *Analyse par intervalles pour l'automatique; résolution globale et garantie de problèmes non linéaires en robotique et commande robuste*, PhD dissertation, Université Paris-Sud, Orsay, France.

Didrit, O., Jaulin, L. and Walter, E. (1997). Guaranteed analysis and optimization of parametric systems, with application to their stability degree, *European Journal of Control* **3**(1): 68–80.

Didrit, O., Petitot, M. and Walter, E. (1998). Guaranteed solution of direct kinematic problems for general configurations of parallel manipulators, *IEEE Transactions on Robotics and Automation* **14**(2): 259–266.

Dijkstra, E. (1959). A note on two problems in connection with graphs, *Numerische Mathematik* **1**: 269–271.

Dorato, P. (2000). Quantified multivariate polynomial inequalities, *IEEE Control Systems Magazine* **20**(5): 48–58.

Dorato, P., Tempo, R. and Muscato, G. (1993). Bibliography on robust control, *Automatica* **29**(1): 201–213.

Doyle, J. (1982). Analysis of feedback systems with structured uncertainties, *IEE Proceedings* **129D**(6): 242–250.

Drumheller, M. (1987). Mobile robot localization using sonar, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **9**(2): 325–332.

Du, D.-Z. (1995). Minmax and its application, *in* R. Horst and P. M. Pardalos (eds), *Handbook of Global Optimization*, Kluwer, Dordrecht, the Netherlands, pp. 339–367.

Durieu, C., Polyak, B. and Walter, E. (1996). Trace versus determinant in ellipsoidal outer bounding with application to state estimation, *Proceedings of the 13th IFAC World Congress*, Vol. I, San Francisco, CA, pp. 43–48.

El Kahoui, M. and Weber, A. (2000). Deciding Hopf bifurcations by quantifier elimination in a software-component architecture, *Journal of Symbolic Computation* **30**(2): 161–179.

Evtushenko, Y. G. (1991). Automatic differentiation viewed from optimal control, *in* A. Griewank and G. F. Corliss (eds), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA, pp. 25–30.

Falk, J. E. (1973). Global solutions for signomial programs, Technical Report T-274, George Washington University, Washington, DC.

Faugère, J.-C. and Lazard, D. (1995). The combinatorial classes of parallel manipulators, *Mechanism and Machine Theory* **30**(6): 765–776.

Ferreres, G. and Magni, J.-F. (1996). Robustness analysis using the mapping theorem without frequency gridding, *Proceedings of the 13th IFAC Triennial World Congress*, San Francisco, pp. 7–12.

Fogel, E. and Huang, Y. F. (1982). On the value of information in system identification - bounded noise case, *Automatica* **18**(2): 229–238.

Francis, B. A. and Khargonekar, P. P. (eds) (1995). *Robust Control Theory*, Vol. 66 of *IMA Volumes in Mathematics and Its Applications*, Springer-Verlag, New York, NY.

Frazer, R. A. and Duncan, W. J. (1929). On the criteria for the stability of small motion, *Proceedings of the Royal Society of London* **124**: 642–654.

Freuder, E. C. (1978). Synthesizing constraint expressions, *Communications of the ACM* **21**(11): 958–966.

Gardenes, E., Mielgo, H. and Trepat, A. (1985). Modal intervals: Reasons and ground semantics, *in* K. Nickel (ed.), *Interval Mathematics 1985*, Vol. 212 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 27–35.

Garloff, J. (2000). Application of Bernstein expansion to the solution of control problems, *Reliable Computing* **6**(3): 303–320.

Garloff, J. and Walter, E. (eds) (2000). Special Issue on Applications to Control, Signals and Systems. *Reliable Computing* **6**(3):229-362.

Gelb, A. (1974). *Applied Optimal Estimation*, MIT Press, Cambridge, MA.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys* **23**(1): 5–47.

Gough, V. E. (1956). Contribution to discussion of papers on research in automobile stability, control and tyre performance, by Cornell staff, *Proceedings of the Automotive Division of the Institution of Mechanical Engineers*, pp. 392–395.

Granvilliers, L. (1998). *Consistances locales et transformations symboliques de contraintes d'intervalles*, PhD dissertation, Université d'Orléans, France.

Grimson, W. E. and Lozano-Pérez, T. (1987). Localizing overlapping parts by searching the interpretation tree, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **9**(4): 469–482.

Gyssens, M., Jeavons, P. G. and Cohen, D. A. (1994). Decomposing constraint satisfaction problems using database techniques, *Artificial Intelligence* **66**(1): 57–89.

Halbwachs, E. and Meizel, D. (1997). Multiple hypothesis management for mobile vehicule localization, *CD-ROM of the European Control Conference*, Louvain, Belgium.

Hammer, R., Hocks, M., Kulish, U. and Ratz, D. (1995). *C++ Toolbox for Verified Computing*, Springer-Verlag, Berlin, Germany.

Hanebeck, U. and Schmidt, G. (1996). Set theoretic localization of fast mobile robots using an angle measurement technique, *Proceedings of the IEEE International Conference on Robotics and Automation*, Minneapolis, MN, pp. 1387–1394.

Hansen, E. R. (1965). Interval arithmetic in matrix computations - part I, *SIAM Journal on Numerical Analysis: Series B* **2**(2): 308–320.

Hansen, E. R. (1968). On solving systems of equations using interval arithmetic, *Mathematical Computing* **22**: 374–384.

Hansen, E. R. (1992a). Bounding the solution of interval linear equations, *SIAM Journal on Numerical Analysis* **29**(5): 1493–1503.

Hansen, E. R. (1992b). *Global Optimization Using Interval Analysis*, Marcel Dekker, New York, NY.

Hansen, E. R. and Sengupta, S. (1980). Global constrained optimization using interval analysis, *in* K. Nickel (ed.), *Interval Mathematics 1980*, Academic Press, New York, NY, pp. 25–47.

Hanson, R. J. (1968). Interval arithmetic as a closed arithmetic system on a computer, Technical Memorandum 197, Jet Propulsion Laboratory, Section 314, California Institute of Technology, Pasadena, CA.

Happel, J. (1986). *Isotopic Assessment of Heterogeneous Catalysis*, Academic Press, Orlando, FL.

Haroud, D., Boulanger, S., Gelle, E. M. and Smith, I. F. C. (1995). Management of conflicts for preliminary engineering design tasks, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **9**(4): 313–323.

Hecht, E. (1987). *Optics,* 2nd edition, Addison-Wesley, Reading, MA.

Hong, H., Liska, R. and Steinberg, S. (1997). Testing stability by quantifier elimination, *Journal of Symbolic Computation* **24**(2): 161–187.

Horowitz, I. M. (1963). *Synthesis of Feedback Systems*, Academic Press, New York, NY.

Husty, M. L. (1996). An algorithm for solving the direct kinematics of general Stewart–Gough platforms, *Mechanism and Machine Theory* **31**(4): 365–379.

Hyvönen, E. (1992). Constraint reasoning based on interval arithmetic; the tolerance propagation approach, *Artificial Intelligence* **58**(1-3): 71–112.

IEEE Computer Society (1985). IEEE standard for binary floating-point arithmetic, *Technical Report IEEE Std. 754-1985*, American National Standards Institute. Available at: http://standards.ieee.org/.

Innocenti, C. and Parenti-Castelli, V. (1991). Direct kinematics of the reverse Stewart platform mechanism, *Proceedings of the 3rd IFAC/IFIP/IMACS Symposium on Robot Control*, Vienna, Austria, pp. 75–80.

Ioakimidis, N. I. (1997). Quantifier elimination in applied mechanics problems with cylindrical algebraic decomposition, *International Journal of Solids and Structures* **34**(30): 4037–4070.

Iri, M. (1991). History of automatic differentiation and rounding estimation, *in* A. Griewank and G. F. Corliss (eds), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA, pp. 1–16.

Jacquez, J. A. (1972). *Compartmental Analysis in Biology and Medecine*, Elsevier, Amsterdam.

Jansson, C. and Knüppel, O. (1995). A branch and bound algorithm for bound constrained optimization problems without derivatives, *Journal of Global Optimization* **7**: 297–331.

Jaulin, L. (1994). *Solution globale et garantie de problèmes ensemblistes ; application à l'estimation non linéaire et à la commande robuste*, PhD dissertation, Université Paris-Sud, Orsay, France. Available at: http://www.istia.univ-angers.fr/~jaulin/thesejaulin.zip.

Jaulin, L. (2000a). Interval constraint propagation with application to bounded-error estimation, *Automatica* **36**(10): 1547–1552.

Jaulin, L. (2000b). *Le calcul ensembliste par analyse par intervalles*, Habilitation à diriger des recherches, Université Paris-Sud, Orsay, France. Available at: http://www.istia.univ-angers.fr/~jaulin/hdrjaulin.zip.

Jaulin, L. (2001a). Path planning using intervals and graphs, *Reliable Computing* **7**(1): 1–15.

Jaulin, L. (2001b). Reliable minimax parameter estimation, *Reliable Computing* **7**(3): 231–246.

Jaulin, L., Braems, I., Kieffer, M. and Walter, E. (2001). Nonlinear state estimation using forward-backward propagation of intervals, to appear in *Proceedings of SCAN 2000*.

Jaulin, L. and Godon, A. (1999). Motion planning using interval analysis, *Proceedings of the MISC'99 Workshop on Applications of Interval Analysis to Systems and Control*, Girona, Spain, pp. 335–346.

Jaulin, L., Kieffer, M., Braems, I. and Walter, E. (2001). Guaranteed nonlinear estimation using constraint propagation on sets, *International Journal of Control* **74(18)**: 1772–1782.

Jaulin, L. and Walter, E. (1993a). Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis, *Mathematics and Computers in Simulation* **35**(2): 123–137.

Jaulin, L. and Walter, E. (1993b). Guaranteed nonlinear parameter estimation via interval computations, *Interval Computations* **3**: 61–75.

Jaulin, L. and Walter, E. (1993c). Set inversion via interval analysis for nonlinear bounded-error estimation, *Automatica* **29**(4): 1053–1064.

Jaulin, L. and Walter, E. (1996). Guaranteed tuning, with application to robust control and motion planning, *Automatica* **32**(8): 1217–1221.

Jaulin, L. and Walter, E. (1997). Global numerical approach to nonlinear discrete-time control, *IEEE Transactions on Automatic Control* **42**(6): 872–875.

Jaulin, L. and Walter, E. (1999). Guaranteed bounded-error parameter estimation for nonlinear models with uncertain experimental factors, *Automatica* **35**(5): 849–856.

Jaulin, L., Walter, E. and Didrit, O. (1996). Guaranteed robust nonlinear parameter bounding, *Proceedings of CESA'96 IMACS Multiconference (Symposium on Modelling, Analysis and Simulation)*, Lille, France, pp. 1156–1161.

Jaulin, L., Walter, E., Lévêque, O. and Meizel, D. (2000). Set inversion for χ-algorithms, with application to guaranteed robot localization, *Mathematics and Computers in Simulation* **52**(3-4): 197–210.

Jirstrand, M. (1997). Nonlinear control system design by quantifier elimination, *Journal of Symbolic Computation* **24**(2): 137–152.

Kahan, W. (1968). A more complete interval arithmetic, Lecture notes for a summer course, University of Toronto, Canada.

Kahan, W. (1996). Lecture notes on the status of IEEE-754, Available at: http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems, *Transactions of the AMSE, Part D, Journal of Basic Engineering* **82**: 35–45.

Kam, M., Zhu, X. and Kalata, P. (1997). Sensor fusion for mobile robot navigation, *Proceeedings of the IEEE* **85**(1): 108–119.

Kang, W. and Krener, A. (1998). Nonlinear observer design: A backstepping approach, *Proceedings of the 13th International Symposium on the Mathematical Theory of Networks and Systems*, Padova, Italy, pp. 245–248.

Kearfott, R. B. (1989a). Interval arithmetic methods for nonlinear systems and nonlinear optimization: an introductory review, *in* R. Sharda, B. L. Golden, E. Wasil, O. Balci and W. Stewart (eds), *Impact of Recent Computer Advances on Operations Research*, North-Holland, New York, NY, pp. 533–542.

Kearfott, R. B. (1989b). Interval mathematics techniques for control theory computations, *in* K. Bowers and J. Lund (eds), *Computation and Control. Proceedings of the Bozeman Conference*, Vol. 20 of *Progress in Systems and Control Theory*, Birkhäuser, Boston, MA, pp. 169–178.

Kearfott, R. B. (1996a). Interval extensions of non-smooth functions for global optimization and nonlinear system solvers, *Computing* **57**(2): 149–162.

Kearfott, R. B. (1996b). *Rigorous Global Search: Continuous Problems*, Kluwer, Dordrecht, the Netherlands.

Kearfott, R. B., Dawande, M., Du, K. S. and Hu, C. (1992). INTLIB: a portable FORTRAN 77 elementary function library, *Interval Computations* **3**(5): 96–105.

Kharitonov, V. L. (1978). Asymptotic stability of an equilibrium position of a family of systems of linear differential equations, *Differentsial'nye Uravnenya* **14**(11): 2086–2088.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots, *International Journal of Robotics Research* **5**(1): 90–98.

Khlebalin, N. A. (1992). Interval automatic systems - theory, computer-aided design and applications, *Interval Computations* **3**: 106–115.

Kieffer, M. (1999). *Estimation ensembliste par analyse par intervalles, application à la localisation d'un véhicule*, PhD thesis, Université Paris-Sud, Orsay, France.

Kieffer, M., Jaulin, L. and Walter, E. (1998). Guaranteed recursive nonlinear state estimation using interval analysis, *Proceedings of the 37th IEEE Conference on Decision and Control*, Tampa, FL, pp. 3966–3971.

Kieffer, M., Jaulin, L., Walter, E. and Meizel, D. (1999). Guaranteed mobile robot tracking using interval analysis, *Proceedings of the MISC'99 Workshop on Applications of Interval Analysis to Systems and Control*, Girona, Spain, pp. 347–359.

Kieffer, M., Jaulin, L., Walter, E. and Meizel, D. (2000). Robust autonomous robot localization using interval analysis, *Reliable Computing* **6**(3): 337–362.

Kieffer, M., Jaulin, L., Walter, E. and Meizel, D. (2001). Localisation et suivi robustes d'un robot mobile grâce à l'analyse par intervalles, *Traitement du Signal* **17**(3): 207–219.

Kieffer, M. and Walter, E. (1998). Interval analysis for guaranteed nonlinear parameter estimation, *in* A. C. Atkinson, L. Pronzato and H. P. Wynn (eds), *MODA 5-Advances in Model-Oriented Data Analysis and Experiment Design*, Physica-Verlag, Heidelberg, pp. 115–125.

Kiendl, H. and Michalske, A. (1992). Robustness analysis of linear control systems with uncertain parameters by the method of convex decomposition, *in* M. Mansour, S. Balemi and W. Truol (eds), *Robustness of Dynamic Systems with Parameter Uncertainties*, Birkhäuser, Boston, MA, pp. 189–198.

Klatte, R., Kulisch, U., Neaga, M., Ratz, D. and Ullrich, C. (1992). PASCAL-XSC – *Language References with Examples*, Springer-Verlag, New York, NY.

Klatte, R., Kulisch, U., Wieth, A., Lawo, C. and Rauch, M. (1993). C-XSC*: A C++ Class Library For Extended Scientific Computing*, Springer-Verlag, Berlin, Germany.

Knofel, A. (1993). Hardware kernel for scientific/engineering computations, *in* E. Adams and U. Kulisch (eds), *Scientific Computing with Automated Result Verification*, Academic Press, Orlando, FL, pp. 549–570.

Knüppel, O. (1993). BIAS - basic interval arithmetic subroutines, Technical Report 93.3, Institut für Informatik III, Technische Universität Hamburg-Harburg, Germany.

Knüppel, O. (1994). PROFIL/BIAS – A fast interval library, *Computing* **53**: 277–287.

Kokame, H. and Mori, T. (1992). A branch and bound method to check the stability of a polytope of matrices, *in* M. Mansour, S. Balemi and W. Truöl (eds), *Robustness of Dynamic Systems with Parameter Uncertainties*, Birkhäuser, Boston, MA, pp. 125–137.

Kolev, L. V. (1993a). An interval first-order method for robustness analysis, *Procedings of the IEEE International Symposium on Circuits and Systems*, Chicago, IL, pp. 2522–2524.

Kolev, L. V. (1993b). *Interval Methods for Circuit Analysis*, World Scientific, Singapore.

Kolev, L. V. (1998). A new method for global solution of nonlinear equations, *Reliable Computing* **4**(2): 125–146.

Kolev, L. V., Mladenov, V. M. and Vladov, S. S. (1988). Interval mathematics algorithms for tolerance analysis, *IEEE Transactions on Circuits and Systems* **35**(8): 967–974.

Kolla, R., Vodopivec, A. and Wolff von Gudenberg, J. (1999). The IAX architecture interval arithmetic extension, Technical Report 225, Institut für Informatik, Universität Würzburg, Germany.

Kuc, R. and Siegel, M. W. (1987). Physically based simulation model for acoustic sensor robot navigation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **9**(6): 766–778.

Kühn, W. (1998). Rigorously computed orbits of dynamical systems without the wrapping effect, *Computing* **61**: 47–67.

Kulisch, U. and Miranker, W. L. (1981). *Computer Arithmetic in Theory and Practice*, Academic Press, New York, NY.

Kurzhanski, A. and Valyi, I. (1997). *Ellipsoidal Calculus for Estimation and Control*, Birkhäuser, Boston, MA.

Kwakernaak, H. (ed.) (1993). Special Issue on Robust Control. *Automatica* **29**(1):1–252.

Lahanier, H., Walter, E. and Gomeni, R. (1987). OMNE: a new robust membership-set estimator for the parameters of nonlinear models, *Journal of Pharmacokinetics and Biopharmaceutics* **15**: 203–219.

Laumond, J. P. (1986). Feasible trajectories for mobile robots with kinematic and environment constraints, *Proceedings of the International Conference on Intelligent Autonomous Systems*, Amsterdam, the Netherlands, pp. 246–354.

Lazard, D. (1992). Stewart platform and Gröbner basis, *in* V. Parenti-Castelli and J. Lenarcic (eds), *Proceedings of the 3rd International Workshop on Advances in Robot Kinematic*, Ferrare, Italy, pp. 136–142.

Lazard, D. (1993). Generalized Stewart platform: how to compute with rigid motions?, *Proceedings of the IMACS Symposium on Symbolic Computation*, Lille, France, pp. 85–88.

Lee, H. and Roth, B. (1993). A closed-form solution of the forward displacement analysis of a class of in-parallel mechanisms, *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 1, Atlanta, GA, pp. 720–724.

Lefèvre, V., Muller, J. and Tisserand, A. (1998). Toward correctly rounded transcendentals, *IEEE Transactions on Computers* **47**(11): 1235–1243.

Leonard, J. J. and Durrant-Whyte, H. F. (1991). Mobile robot localization by tracking geometric beacons, *IEEE Transactions on Robotics and Automation* **7**(3): 376–382.

Lerch, M. and Wolff von Gudenberg, J. (2000). `fi_lib++` : Specification, implementation and test of a library for extended interval arithmetic, *Proceedings of the 4th Conference on Real Numbers and Computers*, Dagstuhl, Germany. Available at: http://www.imada.sdu.dk/~kornerup/RNC4/papers/p03.ps.

Lévêque, O. (1998). *Méthodes ensemblistes pour la localisation de véhicules*, PhD dissertation, Université de Technologie, Compiègne, France.

Levine, W. (ed.) (1996). *The Control Handbook*, CRC Press, Boca Raton, FL.

Lhomme, O. (1993). Consistency techniques for numeric CSPs, *Proceedings of the International Joint Conference on Artificial Intelligence*, Chambéry, France, pp. 232–238.

Lhomme, O. and Rueher, M. (1997). Application des techniques CSP au raisonnement sur les intervalles, *Revue d'Intelligence Artificielle* **11**(3): 283–311.

Liska, R. and Steinberg, S. (1996). Solving stability problems using quantifier elimination, *in* R. Jeltsch and M. Mansour (eds), *Stability Theory: Hurwitz Centenary Conference*, Vol. 121 of *International Series on Numerical Mathematics*, Birkhäuser, Basel, Switzerland, pp. 205–210.

Lohner, R. (1987). Enclosing the solutions of ordinary initial and boundary value problems, *in* E. Kaucher, U. Kulisch and C. Ullrich (eds), *Computer Arithmetic: Scientific Computation and Programming Languages*, BG Teubner, Stuttgart, pp. 255–286.

Lottaz, C. (2000). *Collaborative design using solution spaces*, PhD dissertation 2119, Swiss Federal Institute of Technology in Lausanne, Switzerland.

Lottaz, C., Sam-Haroud, D., Faltings, B. V. and Smith, I. F. C. (1998). Constraint techniques for collaborative design, *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Taipei, Taiwan, ROC, pp. 34–41.

Lozano-Pérez, T. (1981). Automatic planning of manipulator transfer movements, *IEEE Transactions on Systems Man and Cybernetics* **11**(10): 681–698.

Lozano-Pérez, T. (1983). Spatial planning: A configuration space approach, *IEEE Transactions on Computers* **32**(2): 108–120.

Lozano-Pérez, T. and Wesley, M. A. (1979). An algorithm for planning collision-free paths among polyhedral obstacles, *Communications of the ACM* **22**(10): 560–570.

Luenberger, D. (1966). Observers for multivariable systems, *IEEE Transactions on Automatic Control* **11**(2): 190–197.

Mackworth, A. K. (1977a). Consistency in networks of relations, *Artificial Intelligence* **8**(1): 99–118.

Mackworth, A. K. (1977b). On reading sketch maps, *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 2, Cambridge, MA, pp. 598–606.

Mackworth, A. K. and Freuder, E. C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25**(1): 65–74.

Maksarov, D. and Norton, J. P. (1996). State bounding with ellipsoidal set description of the uncertainty, *International Journal of Control* **65**(5): 847–866.

Malan, S. A., Milanese, M. and Taragna, M. (1997). Robust analysis and design of control systems using interval arithmetics, *Automatica* **33**(7): 1363–1372.

Malan, S. A., Milanese, M., Taragna, M. and Garloff, J. (1992). B^3 algorithm for robust performance analysis in presence of mixed parametric and dynamic pertubations, *Proceedings of the 31st IEEE Conference on Decision and Control*, Tucson, AZ, pp. 128–133.

Marden, M. (1966). *Geometry of Polynomials*, American Mathematical Society, Providence, RI.

Marti, P. and Rueher, M. (1995). A distributed cooperating constraint solving system, *International Journal of Artificial Intelligence Tools* **4**(1-2): 93–113.

Matula, D. W. and Kornerup, P. (1985). Finite precision rational arithmetic: Slash number system, *IEEE Transactions on Computers* **34**(1): 3–18.

McKendall, R. (1990). *Minimax estimation of a discrete location parameter for a continuous distribution*, PhD dissertation, Computer and Information Science Department, University of Pennsylvania, Philadelphia, MA. Available at: http://www.cis.upenn.edu/~mcken/pub/om.ps.gz.

McKendall, R. and Mintz, M. (1992). Robust sensor fusion with statistical decision theory, *in* M. A. Abidi and R. C. Gonzalez (eds), *Data Fusion in Robotics and Machine Intelligence*, Academic Press, Boston, MA, pp. 211–244.

Meiri, I., Pearl, J. and Dechter, R. (1990). Tree decomposition with applications to constraint processing, *in* T. Dietterich and W. Swartout (eds), *Proceedings of the 8th AAAI National Conference on Artificial Intelligence*, Boston, MA, pp. 10–16.

Meizel, D., Preciado-Ruiz, A. and Halbwachs, E. (1996). Estimation of mobile robot localization: geometric approaches, *in* M. Milanese, J. Norton,

H. Piet-Lahanier and E. Walter (eds), *Bounding Approaches to System Identification*, Plenum Press, New York, NY, pp. 463–489.

Merlet, J. P. (1990). *Les robots parallèles*, Hermes, Paris, France.

Milanese, M. and Belforte, G. (1982). Estimation theory and uncertainty intervals evaluation in presence of unknown but bounded errors: Linear families of models and estimators, *IEEE Transactions on Automatic Control* **27**(2): 408–414.

Milanese, M., Fioro, G., Malan, S. and Taragna, M. (1991). Robust performance design of nonlinearly perturbed control systems, *Proceedings of the International Workshop on Robust Control*, San Antonio, TX, pp. 113–124.

Milanese, M., Norton, J., Piet-Lahanier, H. and Walter, E. (eds) (1996). *Bounding Approaches to System Identification*, Plenum Press, New York, NY.

Milanese, M. and Vicino, A. (1991). Estimation theory for nonlinear models and set membership uncertainty, *Automatica* **27**(2): 403–408.

Mishra, B. (1993). *Algorithmic Algebra*, Springer-Verlag, New York, NY.

Montanari, U. and Rossi, F. (1991). Constraint relaxation may be perfect, *Artificial Intelligence* **48**(2): 143–170.

Moore, R. E. (1959). Automatic error analysis in digital computation, Technical Report LMSD-48421 Lockheed Missiles and Space Co, Palo Alto, CA.

Moore, R. E. (1966). *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ.

Moore, R. E. (1976). On computing the range of values of a rational function of n variables over a bounded region, *Computing* **16**: 1–15.

Moore, R. E. (1979). *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, PA.

Moore, R. E. (1992). Parameter sets for bounded-error data, *Mathematics and Computers in Simulation* **34**(2): 113–119.

Moore, R. E. and Ratschek, H. (1988). Inclusion function and global optimization II, *Mathematical Programing* **41**(3): 341–356.

Mourrain, B. (1993). The 40 generic positions of a parallel robot, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, Kiev, Ukraine, pp. 173–182.

Muller, J. M. (1997). *Elementary Functions, Algorithms and Implementation*, Birkhäuser, Boston, MA.

Murdock, T. M., Schmitendorf, W. E. and Forrest, S. (1991). Use of a genetic algorithm to analyse robust stability problems, *Proceedings of the American Control Conference*, Boston, MA, pp. 886–889.

Nanua, P., Waldron, K. J. and Murthy, V. (1990). Direct kinematic solution of a Stewart platform, *IEEE Transactions on Robotics and Automation* **6**(4): 438–443.

Nemirovskii, A. (1993). Several NP-hard problems arising in robust stability analysis, *Mathematics of Control, Signals and Systems* **6**(2): 99–105.

Neubacher, A. (1997). *Parametric robust stability by quantifier elimination*, PhD dissertation, Research Institute for Symbolic Computation - Universität Linz, Austria.

Neumaier, A. (1985). Interval iteration for zeros of systems of equations, *BIT* **25**(1): 256–273.

Neumaier, A. (1990). *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, UK.

Nickel, K. (1966). Über die Notwendigkeit einer Fehlerschranken-Arithmetik für Rechenautomaten, *Numerische Mathematik* **9**: 69–79.

Nilsson, N. (1969). A mobile automaton: an application of artificial intelligence techniques, *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, Washington, DC, pp. 509–520.

Norton, J. P. (1987). Identification of parameter bounds for ARMAX models from records with bounded noise, *International Journal of Control* **45**(2): 375–390.

Norton, J. P. (1996). Roles for deterministic bounding in environmental modeling, *Ecological Modelling* **86**: 157–161.

Norton, J. P. (ed.) (1994). Special Issue on Bounded-Error Estimation: Issue 1. *International Journal of Adaptive Control and Signal Processing* **8**(1):1–118.

Norton, J. P. (ed.) (1995). Special Issue on Bounded-Error Estimation: Issue 2. *International Journal of Adaptive Control and Signal Processing* **9**(1):1–132.

O'Dunlaing, C. and Yap, C. K. (1982). A retraction method for planning the motion of a disc, *Journal of Algorithms* **6**(1): 104–111.

Olson, C. (2000). Probabilistic self-localization for mobile robots, *IEEE Transactions on Robotics and Automation* **16**(1): 55–66.

Pardalos, P. and Rosen, J. (1987). *Constrained Global Optimization: Algorithms and Applications*, Vol. 268 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany.

Piazzi, A. and Visioli, A. (1998). Global minimum-time trajectory planning of mechanical manipulators using interval analysis, *International Journal of Control* **71**(4): 631–652.

Poljak, S. and Rohn, J. (1993). Checking robust nonsingularity is NP-hard, *Mathematics of Control, Signals and Systems* **6**: 1–9.

Pronzato, L. and Walter, E. (1994). Minimal volume ellipsoids, *International Journal of Adaptative Control and Signal Processing* **8**(1): 15–30.

Pruski, A. (1996). *La robotique mobile, planification de trajectoire*, Hermes, Paris, France.

Pruski, A. and Rohmer, S. (1997). Robust path planning for non-holonomic robots, *Journal of Intelligent and Robotic Systems* **18**: 329–350.

Psarris, P. and Floudas, C. A. (1995). Robust stability analysis of systems with real parametric uncertainty: a global optimization approach, *International Journal of Robust and Nonlinear Control* **5**(8): 699–717.

Raghavan, M. (1993). The Stewart platform of general geometry has 40 configurations, *ASME Journal of Mechanical Design* **115**(2): 277–282.

Raghavan, M. and Roth, B. (1995). Solving polynomial systems for the kinematic analysis and synthesis of mechanisms and robot manipulators, *ASME Journal of Mechanical Design* **117**: 71–79.

Rall, L. B. (1980). Applications of software for automatic differentiation in numerical computation, *in* G. Alefeld and R. D. Grigorieff (eds), *Fundamentals of Numerical Computation (Computer Oriented Numerical Analysis)*, Computing Supplement No. 2, Springer-Verlag, Berlin, Germany, pp. 141–156.

Rall, L. B. (1981). *Automatic Differentiation: Techniques and Applications*, Vol. 120 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Rall, L. B. and Corliss, G. F. (1999). Automatic differentiation: Point and interval AD, *in* P. M. Pardalos and C. A. Floudas (eds), *Encyclopedia of Optimization*, Kluwer, Dordrecht, the Netherlands.

Ratschan, S. (2000a). Approximate quantified constraint solving (AQCS), Available at: http://www.risc.uni-linz.ac.at/research/software/AQCS.

Ratschan, S. (2000b). Uncertainty propagation in heterogeneous algebras for approximate quantified constraint solving, *Journal of Universal Computer Science* **6**(9): 861–880.

Ratschek, H. and Rokne, J. (1984). *Computer Methods for the Range of Functions*, Ellis Horwood, Chichester, UK.

Ratschek, H. and Rokne, J. (1988). *New Computer Methods for Global Optimization*, Ellis Horwood, Chichester, UK.

Ratschek, H. and Rokne, J. (1995). Interval methods, *in* R. Horst and P. Pardalos (eds), *Handbook of Global Optimization*, Kluwer, Dordrecht, the Netherlands, pp. 751–828.

Ratz, D. and Csendes, T. (1995). On the selection of subdivision directions in interval branch-and-bound methods for global optimization, *Journal of Global Optimization* **7**(2): 183–207.

Reboulet, C. (1988). Modélisation des robots parallèles, *in* J.-D. Boissonat, B. Faverjon and J.-P. Merlet (eds), *Techniques de la robotique, architecture et commande*, Hermes, Paris, France, pp. 257–284.

Rimon, E. and Koditschek, D. E. (1992). Exact robot navigation using artificial potential fields, *IEEE Transactions on Robotics and Automation* **8**(5): 501–518.

Rohn, J. (1994). NP-hardness results for linear algebraic problems with interval data, *in* J. Herzberger (ed.), *Topics in Validated Computations*, Elsevier, Amsterdam, the Netherlands, pp. 463–471.

Rump, S. M. (1999). INTLAB - INTerval LABoratory, *in* T. Csendes (ed.), *Developments in Reliable Computing*, Kluwer, Dordrecht, the Netherlands, pp. 77–104.

Rump, S. M. (2001). INTLAB - INTerval LABoratory, *in* J. Grabmeier, E. Kaltofen and V. Weispfennig (eds), *Handbook of Computer Algebra:*

Foundations, Applications, Systems, Springer-Verlag, Heidelberg, Germany.

Saeki, M. (1986). A method of robust stability analysis with highly structured uncertainties, *IEEE Transactions on Automatic Control* **31**(10): 935–940.

Safonov, M. G. and Athans, M. (1981). A multiloop generalisation of the circle criterion for stability margin analysis, *IEEE Transactions on Automatic Control* **26**(2): 415–422.

Sam-Haroud, D. (1995). *Constraint consistency techniques for continuous domains*, PhD dissertation 1423, Swiss Federal Institute of Technology in Lausanne, Switzerland.

Sam-Haroud, D. J. and Faltings, B. (1996). Consistency techniques for continuous constraints, *Constraints* **1**(1-2): 85–118.

Samet, H. (1982). Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing* **18**(1): 37–57.

Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA.

Schulte, M. J. and Swartzlander, Jr., E. E. (2000). A family of variable-precision, interval arithmetic processors, *IEEE Transactions on Computers* **49**(5): 387–397.

Schweppe, F. C. (1968). Recursive state estimation: unknown but bounded errors and system inputs, *IEEE Transactions on Automatic Control* **13**(1): 22–28.

Schwetlick, H. and Tiller, V. (1985). Numerical methods for estimating parameters in nonlinear models with errors in the variables, *Technometrics* **27**(1): 17–24.

Severance, C. (1998). IEEE 754: An interview with William Kahan, *IEEE Computer* **31**(3): 114–115.

Seybold, B., Metzger, F., Ogan, G. and Simon, K. (1998). Using blocks for constraint satisfaction, *in* M. J. Maher and J. Puget (eds), *Principles and Practice of Constraint Programming - Proceedings of CP98*, Vol. 1520 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, p. 474.

Sideris, A. (1991). An efficient algorithm for checking the robust stability of a polytope of polynomials, *Mathematics of Control, Signals, and Systems* **4**(3): 315–337.

Skelboe, S. (1974). Computation of rational interval functions, *BIT* **14**: 87–95.

Sondergeld, K. P. (1983). A generalization of the Routh–Hurwitz stability criteria and an application to a problem in robust controller design, *IEEE Transactions on Automatic Control* **28**(10): 965–970.

Sorenson, H. (ed.) (1983). Special Issue on Applications of Kalman Filtering. *IEEE Transactions on Automatic Control* **28**(3):253–434.

Steinberg, S. and Liska, R. (1996). Stability analysis by quantifier elimination, *Mathematics and Computers in Simulation* **42**(4-6): 629–638.

Stewart, D. (1965). A platform with six degrees of freedom, *Proceedings of the Institut of Mechanical Engineering* **180**(1): 371–386.

Stine, J. E. and Schulte, M. J. (1998a). A combined interval and floating-point divider, *Proceedings of the 32nd Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, pp. 218–222.

Stine, J. E. and Schulte, M. J. (1998b). A combined interval and floating-point multiplier, *Proceedings of the 8th Great Lakes Symposium on VLSI*, Lafayette, LA, pp. 208–213.

Stroustrup, B. (1991). *The C++ Programming Language,* 2nd edition, Addison-Wesley, Reading, MA.

Swartzlander, E. E. and Alexopoulos, G. A. (1975). The sign/logarithm number system, *IEEE Transactions on Computers* **24**(12): 1238–1242.

Tesi, A. and Vicino, A. (1989). A new fast algorithm for robust stability analysis of control systems with linearly dependent parametric uncertainties, *Systems and Control Letters* **13**(4): 321–329.

van Hentenryck, P., Deville, Y. and Michel, L. (1997). *Numerica: A Modeling Language for Global Optimization*, MIT Press, Boston, MA.

van Hentenryck, P., Michel, L. and Benhamou, F. (1998). `Newton`: Constraint programming over nonlinear constraints, *Science of Computer Programming* **30**(1–2): 83–118.

Veres, S. M. and Norton, J. P. (1996). Parameter-bounding algorithms for linear errors-in-variables models, *in* M. Milanese, J. Norton, H. Piet-Lahanier and E. Walter (eds), *Bounding Approaches to System Identification*, Plenum, New York, NY, pp. 275–288.

Vicino, A., Tesi, A. and Milanese, M. (1990). Computation of nonconservative stability perturbation bounds for systems with nonlinearly correlated uncertainties, *IEEE Transactions on Automatic Control* **35**(7): 835–841.

Walster, G. W. (1998). The extended real interval system. Available at: http://www.mscs.edu/globsol/walster-papers.html.

Walster, G. W., Hansen, E. R. and Sengupta, S. (1985). Test results for a global optimization algorithm, *in* P. Boggs, R. H. Byrd and R. B. Schnaubel (eds), *Numerical Optimization 1984*, SIAM, Philadelphia, PA, pp. 272–287.

Walter, E. (ed.) (1990). Special Issue on Parameter Identification with Error Bounds. *Mathematics and Computers in Simulation* **32**(5-6):447–607.

Walter, E. and Jaulin, L. (1994). Guaranteed characterization of stability domains via set inversion, *IEEE Transactions on Automatic Control* **39**(4): 886–889.

Walter, E. and Piet-Lahanier, H. (1989). Exact recursive polyhedral description of the feasible parameter set for bounded-error models, *IEEE Transactions on Automatic Control* **34**(8): 911–915.

Walter, E. and Pronzato, L. (1997). *Identification of Parametric Models from Experimental Data*, Springer-Verlag, London.

Waltz, D. (1975). Generating semantic descriptions from drawings of scenes with shadows, *in* P. H. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, NY, pp. 19–91.

Wampler, C. W. (1996). Forward displacement analysis of general six-in-parallel SPS (Stewart) platform manipulators using soma coordinates, *Mechanism and Machine Theory* **31**(3): 331–337.

Wang, L.-C. T. and Chen, C. C. (1993). On the numerical kinematic analysis of general parallel robotic manipulators, *IEEE Transactions on Robotics and Automation* **9**(3): 272–285.

Wei, K. H. and Yedavalli, R. K. (1989). Robust stabilizability for linear systems with both parameter variation and unstructured uncertainty, *IEEE Transactions on Automatic Control* **34**(2): 149–156.

Willems, J. C. (1986a). From time series to linear systems, part 1, finite dimensional linear time invariant systems, *Automatica* **22**(5): 561–580.

Willems, J. C. (1986b). From time series to linear systems, part 2, exact modelling, *Automatica* **22**(6): 675–694.

Witsenhausen, H. S. (1968). Sets of possible states of linear systems given perturbed observations, *IEEE Transactions on Automatic Control* **13**(5): 556–558.

Wolfe, M. A. (1996). Interval methods for global optimization, *Applied Mathematics and Computation* **75**: 179–206.

Wolfe, M. A. (1999). On discrete minimax problems in the set of real numbers using interval arithmetic, *Reliable Computing* **5**(4): 371–383.

Wolff von Gudenberg, J. (1994). Comparison of accurate dot product algorithms, *Technical Report 881*, IRISA, Rennes, France. Available at: http://www.irisa.fr/bibli/publi/pi/1994/881/881.html.

Wolff von Gudenberg, J. (1996). Hardware support for interval computation, *in* G. Alefeld, A. Frommer and B. Lang (eds), *Scientific Computing and Validated Numerics*, Akademie-Verlag, Berlin, Germany, pp. 32–37.

Zhou, J. (1996). A permutation-based approach for solving the job-shop problem, *Constraints* **1**: 1–30.

Zuhe, S., Neumaier, A. and Eiermann, M. (1990). Solving minimax problems by interval methods, *BIT* **30**: 742–751.

Index