Nonconvex Optimization and Its Applications

Volume 13

The titles published in this series are listed at the end of this volume.

Rigorous Global Search: Continuous Problems

by

R. Baker Kearfott
University of Southwestern Louisiana

KLUWER ACADEMIC PUBLISHERS
DORDRECHT / BOSTON / LONDON

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN 0-7923-4238-0

Published by Kluwer Academic Publishers,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

Kluwer Academic Publishers incorporates
the publishing programmes of
D. Reidel, Martinus Nijhoff, Dr W. Junk and MTP Press.

Sold and distributed in the U.S.A. and Canada
by Kluwer Academic Publishers,
101 Philip Drive, Norwell, MA 02061, U.S.A.

In all other countries, sold and distributed
by Kluwer Academic Publishers Group,
P.O. Box 322, 3300 AH Dordrecht, The Netherlands.

Printed on acid-free paper

Printed in the Netherlands

CONTENTS

LIST OF FIGURES vii

LIST OF TABLES xi

PREFACE xiii

1 PRELIMINARIES 1
- 1.1 Interval Arithmetic 3
- 1.2 Interval Linear Systems 18
- 1.3 Derivatives and Slopes 26
- 1.4 Automatic Differentiation and Code Lists 36
- 1.5 Interval Newton Methods and Interval Fixed Point Theory 50
- 1.6 The Topological Degree 66

2 SOFTWARE ENVIRONMENTS 71
- 2.1 INTLIB 71
- 2.2 Fortran 90 Interval and Code List Support 78
- 2.3 Other Software Environments 102

3 ON PRECONDITIONING 113
- 3.1 The Inverse Midpoint Preconditioner 115
- 3.2 Optimal Linear Programming Preconditioners 120

4 VERIFIED SOLUTION OF NONLINEAR SYSTEMS 145
- 4.1 An Overall Branch and Bound Algorithm 146
- 4.2 Approximate Roots and Epsilon-Inflation 150

4.3 Tessellation Schemes 154
4.4 Description of Provided Software 159
4.5 Alternate Algorithms and Improvements 167

5 OPTIMIZATION 169
5.1 Background and Historical Algorithms 170
5.2 Handling Constraints 177
5.3 Description of Provided Software 199

6 NON-DIFFERENTIABLE PROBLEMS 209
6.1 Extensions of Non-Smooth Functions 210
6.2 Use in Interval Newton Methods 218

7 USE OF INTERMEDIATE QUANTITIES IN THE EXPRESSION VALUES 227
7.1 The Basic Approach 227
7.2 An Alternate Viewpoint – Constraint Propagation 230
7.3 Application to Global Optimization 230
7.4 Efficiency and Practicality 232
7.5 Provided Software 233
7.6 Exercises 234

REFERENCES 235

INDEX 255

LIST OF FIGURES

Chapter 1

1.1 The united solution set $\Sigma(A, B)$ for the system (1.19) 20

1.2 $\Sigma(A, B) \cap X$ can still be bounded when $AX = B$ is underdetermined. 23

1.3 The difference between the interval slope $\mathbf{S}^{\natural}(f, x, \check{x})$ and the derivative range f'_u for $x = [0, 2]$, $\check{x} = 1$ and $f(x) = x^2$ 28

1.4 Two ways of computing the change in ϕ between $(\check{x}_1, \check{x}_2)$ and (x_1, x_2) 31

1.5 Sample program to compute a floating point function value from a code list 49

1.6 Non-uniqueness with slopes with $f(x) = (x^2 - 1)(x + 2)$, $x = [-2, 3]$ and $\check{x} = 3$ 64

Chapter 2

2.1 A `FORTRAN-77` program using INTLIB 77

2.2 Illustration of use of the interval data type 79

2.3 A univariate interval Newton method program, used to generate the data in Table 1.3 82

2.4 Program that generates a code list for $f(x) = x^4 + x^3 + x$ 85

2.5 Operation lines of the code list produced by the program of Figure 2.4 85

2.6 Sample `OVERLOAD.CFG` file 85

2.7 Accumulating a product for a dependent variable 86

2.8 Reassignment of a dependent variable 87

2.9 The code list operation lines for the program in Figure 2.8 87

2.10 (a) Function `USR` for Example 2.1 90

2.10 (b) Function `USRD` for Example 2.1 90

2.11 A program to generate a code list for $f_1(x) = f(x) - 2$, where $f(x)$ is as in Example 2.1 91

2.12 Generation of a derivative code list corresponding to the code list in file EX1.CDL 92
2.13 Operation lines of the code list for the derivative of $f(x) = x^4 + x^3 + x$ 92
2.14 Algebraic interpretation of the code list in Figure 2.13 92
2.15 A code list for the function in Example 2.2 93
2.16 The derivative code list corresponding to Figure 2.15 93
2.17 A program to generate a code list for $\phi(x_1, x_2) = x_1^2 + x_2^2$, subject to $x_1 + x_2 - 1 = 0$ 100
2.18 Evaluation of the constraints in Example 2.3 100

Chapter 3

3.1 Action of a C-preconditioner 125
3.2 Action of an E-preconditioner 125
3.3 Appropriate situation for a left-optimal preconditioner 126
3.4 Action of a magnitude-optimal C-preconditioner: $\max\{a, b\}$ is minimized. 127
3.5 Action of a mignitude-optimal E-preconditioner: $\min\{a, b\}$ is maximized. 128

Chapter 4

4.1 Complementation of a box in a box 154
4.2 Complementation of a box in a list of boxes 156
4.3 Bisection of the second coordinate according to maximum smear, for Example 4.1 158
4.4 Program to produce a code list for a counterexample to a method of Branin 161
4.5 Box data file BRANIN.DT1 162
4.6 RUN_ROOTS_DELETE for BRANIN.CDL and BRANIN.DT1 162
4.7 Sample output to RUN_ROOTS_DELETE 163
4.8 Box data file BRANIN.DT2 163
4.9 Use of RUN_ROOTS_DELETE for uniqueness verification 164

Chapter 5

5.1 "Peeling " a box to produce lower-dimensional boundary elements 181

5.2 "Peeling" the box into lower-dimensional boundary elements 183
5.3 Proving that there exists a feasible point of an underdetermined constraint system 186
5.4 Proving existence in a reduced space when the approximate feasible point satisfies bound constraints 186
5.5 A common degenerate case, when $\check{X}$ must be perturbed 187
5.6 Geometry of a feasibility proof with LP preconditioners 190
5.7 Box data file `GOULD.DT1` 202

Chapter 6

6.1 Program that generates a code list for $f(x) = x^2$ if $x < 1$, $f(x) = 2x - 1$ if $x \geq 1$ 211
6.2 Operation lines of the code list produced by the program of Figure 6.1 212
6.3 Computation of slope bounds of a discontinuous function 214

LIST OF TABLES

Chapter 1

1.1 Some example operations and corresponding numerical codes 44
1.2 Tabular representation of the output to the program (1.36) 48
1.3 Illustration of quadratic convergence of the univariate interval Newton method for $f(x) = x^2 - 4$ 54
1.4 Illustration of quadratic convergence of the Krawczyk method for F as in Example 1.5 58

Chapter 2

2.1 Elementary arithmetic routines in INTLIB 74
2.2 Standard function routines in INTLIB 75
2.3 Utility functions in INTLIB 75
2.4 Special interval functions in module `INTERVAL_ARITHMETIC` 80
2.5 Logical and interval operators in module `INTERVAL_ARITHMETIC` 81
2.6 Operations in module `OVERLOAD` and program `MAKE_GRADIENT` 84
2.7 Additional operations supported in module `OVERLOAD` 85
2.8 Operational complexity – generic routines 98
2.9 Ratio of interval to floating point CPU times `CPURAT` for ACRITH-XSC on an IBM 3090 105
2.10 Ratio of interval to floating point CPU times `CPURAT` for `INTERVAL_ARITHMETIC` on a Sun SPARC 20 model 51 105

Chapter 5

5.1 Summary attributes of various global optimization algorithms 177
5.2 Summary of handling of constraints in various global optimization algorithms 178

5.3 Summary of 3 methods of handling constraints for Example 5.2 198

Chapter 6

6.1 Iterates of the interval Newton method for $f(x) = |x^2 - x| - 2x + 2$ 225

PREFACE

This work grew out of several years of research, graduate seminars and talks on the subject. It was motivated by a desire to make the technology accessible to those who most needed it or could most use it. It is meant to be a self-contained introduction, a reference for the techniques, and a guide to the literature for the underlying theory. It contains pointers to fertile areas for future research. It also serves as introductory documentation for a Fortran 90 software package for nonlinear systems and global optimization.

The subject of the monograph is deterministic, *automatically verified* or *rigorous* methods. In such methods, directed rounding and computational fixed-point theory are combined with exhaustive search (branch and bound) techniques. Completion of such an algorithm with a list of solutions constitutes a *rigorous mathematical proof* that all of the solutions within the original search region are within the output list.

The monograph is appropriate as an introduction to research and technology in the area, as a desk reference, or as a graduate-level course reference. Knowledge of calculus, linear algebra, and elementary numerical analysis is assumed. Interval computations are presented from the beginning, although the more advanced material is mainly to support development of ideas in nonlinear systems and global optimization. The style is meant to balance the need to inform newcomers with the need to provide concise treatment for experts. The emphasis in the advanced topics is on the author's own contributions to the field. The book contains numerous references and cross-references, as well as an extensive index. A logical thread is provided for easy initial reading, with gateways for subsequent in-depth study.

Although the major goal of the book is to explain techniques and software for rigorous solution of nonlinear systems and rigorous, deterministic global optimization, the extensive introduction in Chapter 1 is suitable as a general introduction or reference for interval arithmetic. Thus, this introduction can also be useful in studying verified quadrature or verified solutions of differential equations, for example.

Similarly, the software described in Chapter 2 includes general-use interval arithmetic software. This software includes a Fortran-90 module for interval arithmetic and Fortran 90 modules for automatic differentiation (termed INTLIB_90), freely available from the author (email: rbk@usl.edu), and also includes more special-purpose nonlinear equations and global optimization codes (termed INTOPT_90). Chapter 2 also contains a review of alternate programming languages and packages for interval computations, with a discussion of the advantages and disadvantages of each.

In Chapter 3, previously unpublished material on optimal preconditioners for interval linear systems is presented. Illustrations and examples are meant to point the reader to the contexts in which such preconditioners can be advantageous, as well as to give necessary explanation for implementation. (The author's software contains routines for one type of optimal preconditioner.)

Chapter 4 contains an introduction to algorithmic constructs for global search methods for solutions to nonlinear systems of equations. These constructs include overall algorithm structure, as well as methods for subdividing the region and taking advantage of accurate approximate solutions. Some of these techniques and algorithmic constructs are shared by global optimization algorithms, described in the next chapter. Chapter 4 also contains a description of the author's software for nonlinear equations, in an overall package (termed INTOPT_90).

Chapter 5 deals with actual global optimization techniques and algorithms. Beginning with an overall view, historical algorithms and recent, more sophisticated algorithms are reviewed. This is followed by an extensive review of how inequality and equality constraints are handled, including some of the author's proposed techniques and experience. The chapter concludes with a description of the routines available in INTOPT_90.

Chapter 6 deals with special techniques for non-differentiable problems, such as l_1 and l_∞ optimization. In particular, formulas are listed for interval extensions of non-smooth and discontinuous functions, as well as representation of derivative information for non-smooth and discontinuous functions. Surprisingly, such interval extensions, even for discontinuous functions, can be effective in interval Newton methods; examples are given, and a convergence analysis is reviewed. Support for the techniques is available in the author's software, INTLIB_90.

Chapter 7 deals with using the interval values of *subexpressions*, obtained while evaluating objective functions and gradients. These techniques are tied to the

preconditioner techniques of Chapter 3, and are also related to the established field of *constraint propagation.*

Throughout the book, techniques and software are presented not only to explain available work, but also to guide interested readers to future improvements or to ways of using individual techniques in different contexts.

Elements of the theory deemed most relevant are presented. However, theorems appear within the context of the meaning and practical impact of their results. Where proofs are presented, it is mainly for the purpose of giving additional insight. Where proofs are not presented, references to the original research are given.

Exercises after many sections amplify the preceding presentation.

The references are available electronically in BibTeX form, via anonymous FTP from

`interval.usl.edu`

in the directory

`pub/interval_math/bibliographies/optimization_book.bib`

I wish to thank my former student Dr. Xiaofa Shi, and my colleagues George Corliss, Panos Pardalos, Dietmar Ratz, and Siegfried Rump for the careful reading and suggestions that improved this book considerably. I also would like to thank Layne Watson for introducing me to TeX and for encouraging me, early on, to work hard.

1

PRELIMINARIES

The main purpose of this book is to introduce techniques and software for the verified solution of nonlinear systems of equations and for rigorous, deterministic unconstrained and constrained global optimization. Specifically, global optima or solutions of nonlinear equations will be sought within the *box*

$$\boldsymbol{X} = \left\{(x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n \;\middle|\; \underline{x}_i \le x_i \le \overline{x}_i, 1 \le i \le n\right\}, \tag{1.1}$$

for some set of lower bounds $\{\underline{x}_i\}_{i=1}^n$ and upper bounds $\{\overline{x}_i\}_{i=1}^n$. The fundamental nonlinear equations problem is

Given $F : \boldsymbol{X} \to \mathbb{R}^n$, *rigorously* enclose *all* solutions $X^* \in \boldsymbol{X}$. That is, for each

$$X^* = (x_1^*, \ldots, x_n^*)^T \in \boldsymbol{X}$$

with $F(X^*) = 0$, find bounds $a_i \le x_i^* \le b_i$ such that

- $b_i - a_i$ is small, $1 \le i \le n$, and
- it is mathematically but automatically proven that there is a unique root of F within each

$$\check{\boldsymbol{X}} = \left\{X = (x_1, \ldots, x_n) \;\middle|\; a_i \le x_i \le b_i, 1 \le i \le n\right\}.$$

(1.2)

We will phrase the corresponding constrained global optimization problem as

Given $\phi : \boldsymbol{X} \to \mathbb{R}$ and constraints

$$C(X) = (c_1(X), \ldots, c_m(X))^T : \mathbb{R}^n \to \mathbb{R}^m,$$

rigorously find upper and lower bounds to the values of ϕ that solve

$$\begin{array}{rlcll} \text{minimize} & \phi(X) & & & \\ \text{subject to} & c_i(X) & = & 0, & i = 1, \ldots, m, \\ & \underline{x}_{i_j} \leq & x_{i_j}, & & j = 1, \ldots q - \mu, \\ & x_{i_j} \leq & \overline{x}_{i_j}, & & j = \mu + 1, \ldots q, \end{array} \tag{1.3}$$

and, for each minimizer $X^* \in \boldsymbol{X}$, find bounds $a_i \leq x_i^* \leq b_i$ such that

- $b_i - a_i$ is small, $1 \leq i \leq n$, and
- it is mathematically but automatically proven that there is a unique critical point of ϕ within each

$$\check{\boldsymbol{X}} = \{X = (x_1, \ldots, x_n) \mid a_i \leq x_i \leq b_i, 1 \leq i \leq n\}.$$

Variants of the problems, such as when there are singularities, and uniqueness thus cannot be verified, will be discussed. Also, individual techniques will be treated separately, so that they can be used outside the main context. For example, a single approximate root $\widehat{X}$ of $F(X) = 0$ may have been found via a traditional nonlinear equations solver; it is relatively efficient to then construct bounds $\check{\boldsymbol{X}}$ as in Problem (1.2), within which it is automatically proven that there is a unique root of F.

The cornerstone of automatically verified solution of Problem (1.2) or Problem (1.3) is the ability to compute *rigorous bounds* on the range of functions and expressions (such as the range of ϕ over $\boldsymbol{X}$), and to use such bounds to verify the hypotheses of fixed-point theorems, coupled with branch and bound algorithms for exhaustive search. This chapter introduces mathematical theory and techniques underlying such range computations and fixed-point verification.

1.1 INTERVAL ARITHMETIC

Prior to widespread use of interval arithmetic, bounds on the range of a function were sometimes obtained with Lipschitz constants or moduli of continuity. Such computations can be considered special cases of computation of *interval enclosures*. Judicious use of *interval arithmetic* allows such range bounds to be computed efficiently, without extensive ad hoc analysis. Additionally, interval arithmetic, with *directed roundings*, can provide mathematically rigorous results from floating point operations on computers.

1.1.1 Real Interval Arithmetic

Introduced in its modern form by R. E. Moore [163], real interval arithmetic is based on arithmetic within the set of closed intervals of real numbers. If $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$, then the four elementary operations for such *idealized interval arithmetic* obey

$$\boldsymbol{x} \operatorname{op} \boldsymbol{y} = \{x \operatorname{op} y \mid x \in \boldsymbol{x} \text{ and } y \in \boldsymbol{y}\} \quad \text{for } \operatorname{op} \in \{+, -, \times, \div\} \tag{1.4}$$

Thus, the image of each of the four basic interval operations is the *exact range* of the corresponding real operation. Although Equation (1.4) characterizes these operations mathematically, interval arithmetic's usefulness is due to the *operational definitions*. For example,

$$\boldsymbol{x} + \boldsymbol{y} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \tag{1.5}$$

$$\boldsymbol{x} - \boldsymbol{y} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}], \tag{1.6}$$

$$\boldsymbol{x} \times \boldsymbol{y} = [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \tag{1.7}$$

$$\frac{1}{\boldsymbol{x}} = [1/\overline{x}, 1/\underline{x}] \quad \text{if } \underline{x} > 0 \text{ or } \overline{x} < 0 \tag{1.8}$$

$$\boldsymbol{x} \div \boldsymbol{y} = \boldsymbol{x} \times 1/\boldsymbol{y} \tag{1.9}$$

Here, reciprocation, and hence ordinary interval division $\boldsymbol{x}/\boldsymbol{y}$ is undefined when $0 \in \boldsymbol{y}$. However, under certain circumstances (as in §1.1.4 and §1.5.1 below), it makes sense to define such quotients in an *extended* or *Kahan–Novoa–Ratz arithmetic*. Also, when multiplication is actually implemented on a computer, a

somewhat more complicated alternative to Equation (1.7), such as [165, (2.20), p. 12], is often used, since it is faster on average.

Although the ranges of interval arithmetic operations are exactly the ranges of the corresponding real operations, this is not so if the operations are composed. For example, if

$$f(x) = x^2 - x,$$

then evaluating f over $\boldsymbol{x} = [0,1]$ with interval arithmetic gives

$$[0,1]^2 + [0,1] = [0,1] - [0,1] = [-1,1],$$

but the range of f over $[0,1]$ is $[-1/4, 0]$; thus, the interval computation *overestimates* the range. This is due to the fact that the variable x is implicitly assumed to vary independently in the term x^2 and the term $-x$, so that e.g. numbers such as $1^2 - 0$, in addition to $0^2 - 0$ and $1^2 - 1$, are included in the interval result. This phenomenon is termed *interval dependency*. Due to interval dependency, algebraic expressions that are equivalent in real arithmetic give different results when evaluated in interval arithmetic. For example, if f above is written as $x(x-1)$, then

$$[0,1]([0,1] - 1) = [0,1][-1,0] = [-1,0] \neq [-1,1].$$

In this case, the expression $x(1-x)$, obtained using Horner's method, gives a sharper bound on the range than the power representation, although that is not always the case[1]. Theorem 1.4 in §1.1.7 below states an additional property of interval dependency. Also, Theorem 1.3 below asserts that *evaluating a function with interval arithmetic always gives bounds on the range of the function.* Here, we can state

Theorem 1.1 *Interval arithmetic is* subdistributive *in the sense that, if* $\boldsymbol{x}$, $\boldsymbol{y}$, *and* $\boldsymbol{z}$ *are intervals, then*

$$\boldsymbol{x}(\boldsymbol{y} + \boldsymbol{z}) \subseteq \boldsymbol{x}\boldsymbol{y} + \boldsymbol{x}\boldsymbol{z}.$$

Thus, although addition or multiplication of intervals is commutative and associative, the distributive laws do not hold. Furthermore, although there is an additive identity $[0,0]$ and a multiplicative identity $[1,1]$, additive and multiplicative inverses do not exist. For instance, the example

$$[1,2] - [1,2] = [-1,3]$$

[1] One study related to this is [192].

illustrates that cancellation does not fully occur in subtraction as defined by Equation (1.4). However, a fifth operation, *cancellation subtraction*, is appropriate in various contexts. Cancellation subtraction is defined by the equation

$$[\underline{x}, \overline{x}] \ominus [\underline{y}, \overline{y}] = [\underline{x} - \underline{y}, \overline{x} - \overline{y}]. \tag{1.10}$$

For example, if interval sums $s_j = \sum_{i=1, i \neq j}^{n} \boldsymbol{x}_i$ are required for each j, the overall sum $s = \sum_{i=1}^{n} \boldsymbol{x}_i$ can first be computed, then the individual s_j can be obtained with cancellation subtraction.

Additional elementary properties of interval arithmetic are listed in [8], [77], [83], [165], and [175].

1.1.2 Notation

Throughout, boldface will denote intervals, lower case will denote scalar quantities, and upper case will denote vectors and matrices. Brackets "[·]" will delimit intervals while parentheses "(·)" will delimit vectors and matrices. Underscores will denote lower bounds of intervals and overscores will denote upper bounds of intervals. Corresponding lower case letters will denote components of vectors. For example, we may have:

$$\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)^T,$$

where $\boldsymbol{x}_i = [\underline{x}_i, \overline{x}_i]$. If $\boldsymbol{X}$ is an interval vector or matrix, then $\boldsymbol{X} = [\underline{X}, \overline{X}]$, where $\underline{X}$ is the vector or matrix whose components are lower bounds of corresponding components of $\boldsymbol{X}$, and $\overline{X}$ is the vector or matrix whose components are upper bounds of corresponding components of $\boldsymbol{X}$. For example, if $\boldsymbol{X} = ([1,2],[3,4])^T$, then $\underline{X} = (1,3)^T$ and $\overline{X} = (2,4)^T$.

The symbol $\check{x}$ will denote a representative point, usually in $\boldsymbol{x}$ and often its center. Similarly, $\check{X}$ will denote a representative point for the box $\boldsymbol{X}$. The actual center, or midpoint, of an interval $\boldsymbol{x}$ will be denoted by $\mathrm{m}(\boldsymbol{x})$, and the vector or matrix whose entries are midpoints of the entries of the vector or matrix $\boldsymbol{X}$ will be denoted by $\mathrm{m}(\boldsymbol{X})$. The *magnitude* of an interval is defined as $|\boldsymbol{x}| = \max\{|\underline{x}|, |\overline{x}|\}$. The magnitude of an interval vector or matrix will be interpreted componentwise:

$$|\boldsymbol{X}| = (|\boldsymbol{x}_1|, |\boldsymbol{x}_2|, \ldots, |\boldsymbol{x}_n|)^T,$$

while the norm of an interval vector is defined as $\|\boldsymbol{X}\| = \big\||\boldsymbol{X}|\big\|_\infty$. The *magnitude* of an interval $\boldsymbol{x}$ will be defined by $<\boldsymbol{x}> = \min_{x \in \boldsymbol{x}} |x|$.

The *width* of an interval $\boldsymbol{x}$ is denoted by $\mathrm{w}(\boldsymbol{x}) = \overline{x} - \underline{x}$. The width of an interval vector $\boldsymbol{X}$, denoted by $\mathrm{w}(\boldsymbol{X})$, is defined componentwise. We use $\mathrm{w}(\boldsymbol{X})$ in the context of $\|\mathrm{w}(\boldsymbol{X})\| = \|\mathrm{w}(\boldsymbol{X})\|_\infty$.

The set of intervals will be denoted $\mathbb{IR}$, the set of n-dimensional interval vectors, also called *boxes*, will be denoted by $\mathbb{IR}^n$, and the set of m by n matrices whose entries are intervals will be denoted by $\mathbb{IR}^{m\times n}$. The set corresponding to $\mathbb{IR}^2$, when identified with rectangles in the complex plane, will be denoted by $\mathbb{IC}$.

Arithmetic operations involving both real numbers and intervals will occur. In these, such as $[1,2] + 1 = [2,3]$, the real number 1 is interpreted as a *thin interval* $[1,1]$. (Thin intervals are simply intervals of width zero.)

The topological interior of a set $\mathbf{D}$ will be denoted by $\mathrm{int}(\mathbf{D})$. Unless otherwise stated, such sets (and in particular, intervals) will be considered to be topologically closed. The boundary of a set $\mathbf{D}$ will be denoted by $\partial\mathbf{D}$.

If f is a function defined over an interval $\boldsymbol{x}$, then $\boldsymbol{f}^{\boldsymbol{u}}(\boldsymbol{x})$ denotes[2] the range of f over $\boldsymbol{x}$. The ranges of functions $\phi : \mathbb{R}^n \to \mathbb{R}$ and $F : \mathbb{R}^n \to \mathbb{R}^m$ are similarly denoted as $\boldsymbol{\phi}^{\boldsymbol{u}}$ and $\boldsymbol{F}^{\boldsymbol{u}}$, respectively. The gradient of ϕ is denoted by $\nabla\phi$, while the generic notation $F'(X)$ represents the m by n Jacobi matrix of a function $F : \mathbb{R}^n \to \mathbb{R}^m$.

Comparison of intervals will be as sets. For example $\boldsymbol{x} < \boldsymbol{y}$ will mean that every element of $\boldsymbol{x}$ is less than every element of $\boldsymbol{y}$. Comparison of interval vectors $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{IR}^n$ will be componentwise, in the sense of a relation as in [133, pp. 129–130]. For example, $\boldsymbol{X} \leq \boldsymbol{Y}$ will mean that $\boldsymbol{x}_i \leq \boldsymbol{y}_i$ for i between 1 and n, while $\boldsymbol{X} < \boldsymbol{Y}$ will mean $\boldsymbol{x}_i \leq \boldsymbol{y}_i$ for i between 1 and n, and $\boldsymbol{x}_i \neq \boldsymbol{y}_i$ for at least one i. The convex hull of two intervals $\boldsymbol{x}$ and $\boldsymbol{y}$ is an interval that will be denoted by $\boldsymbol{x} \cup \boldsymbol{y}$.

Often, sequences of intervals $\boldsymbol{x}_i$, as well as interval vectors and interval matrices, will be considered. This leads to

Definition 1.1 *Let* $\{\boldsymbol{x}_i\}_{i=1}^{\infty} = \{[\underline{x}_i, \overline{x}_i]\}_{i=1}^{\infty}$ *be a sequence of intervals. The sequence is said to* converge *to an interval* $\boldsymbol{x} = [\underline{x}, \overline{x}]$ *provided* $\underline{x}_i \to \underline{x}$ *and* $\overline{x}_i \to \boldsymbol{x}_i$ *in the standard topology on the real line. Similarly, a sequence of intervals is said to converge to a real number* r *provided* $\underline{x}_i \to r$ *and* $\overline{x}_i \to r$. *A sequence of vectors or matrices is said to converge to a vector or matrix if each*

[2]The notation is suggestive of "united" interval extension, a term first used by Moore.

of its components or coefficients converges to the corresponding component or coefficient of the limit vector or matrix.

For example, we may say that a set of interval matrices $\boldsymbol{A}_k$ converges to the identity matrix I.

1.1.3 Rounded Interval Arithmetic

Computations in §1.1.1 illustrated how real interval arithmetic could compute rigorous bounds on the ranges of functions, assuming infinitely precise computations. However, if such an arithmetic is implemented on the computer using, for example, equations (1.5) through (1.9), then the default rounding may result in non-rigor. For example, the range of the expression $[0.123, 0.456] + [0.0116, 0.0214]$ is $[0.1346, 0.4774]$. But this would be rounded to $[0.135, 0.477]$ with three digit decimal arithmetic and rounding to nearest, and $[0.1346, 0.4774] \not\subseteq [0.135, 0.477]$. Nonetheless, with *directed rounding*, such bounds can be computed rigorously. In particular, if instead of rounding to nearest, the lower bound of the interval is rounded down to the largest machine number less than the exact result and the upper bound is rounded up to the smallest machine number greater than the actual result, then the computed interval necessarily contains the exact range. In our example, the result would be $[0.134, 0.475]$, and $[0.1346, 0.4664] \subset [0.134, 0.475]$. This process is called *outward rounding*, and the resulting widening of the intervals is called *roundout error*.

Thus, with directed rounding, a *machine interval arithmetic* can be defined, such that the result of the four elementary operations contains the result that would be obtained with real interval arithmetic. Mathematical development of this concept is found in [147] and other works of Kulisch *et al.* For notational simplicity, a distinction between the space $\mathbb{IR}$ and the corresponding set of machine intervals will not be made explicit, but should be inferred from the context.

The IEEE binary floating point standard [227] prescribes three rounding modes: nearest, round down, and round up. Thus, rounding modes suitable for interval arithmetic are available on a wide variety of machines. However, standardized programming language support for switching of rounding modes is not available, although the Fortran 90 intrinsic NEAREST [4, p. 636] comes close. Transportable implementations of interval arithmetic thus often resort to mixed assembly language programming. Notwithstanding, if the tightest possible bounds on ranges are not required, then rounding down and rounding

up can be simulated. The main requirement is that it be known by how many units in the last place[3] (ULP's) the results of the elementary operations $+$, $-$, $\times$ and $\div$ can be in error. (Note that the IEEE standard [227, §4] specifies that all stored digits be correct, which translates to 1/2 ULP accuracy.) The ACM TOMS algorithm[4] INTLIB implements such a *simulated directed rounding*, and provides rigorous interval arithmetic in portable **FORTRAN-77** on a wide variety of machines, including some common computers without IEEE arithmetic. See §2.1.2 below.

Computing dot products with the formula $\boldsymbol{X} \circ \boldsymbol{Y} = \sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{y}_i$ and interval addition sometimes results in significant overestimation of the range of $\boldsymbol{X} \circ \boldsymbol{Y}$, especially when the widths of the components of $\boldsymbol{X}$ and $\boldsymbol{Y}$ are small. Because such dot products are important in obtaining *a posteriori* rigorous error bounds to solutions of linear systems of equations, Kulisch *et al.* [147] have proposed considering $\circ$ as a fifth elementary operation, having the same accuracy as $+$, $-$, $\times$ and $\div$. This *maximally accurate dot product* is implemented in the excellent "SC" and "XSC" languages [243] such as FORTRAN-SC [21, 237] (known commercially as ACRITH-XSC), Pascal-SC [188], C-XSC [134], and Pascal-XSC [71]. It enables computation of tight and rigorous bounds to the solutions of very ill-conditioned linear systems. However, the accurate dot product gives less benefit when the widths of the component intervals of $\boldsymbol{X}$ and $\boldsymbol{Y}$ are large in relation to roundoff error, as is the case in most (but not all) of the interval computation in optimization and solution of nonlinear systems.

1.1.4 Kahan–Novoa–Ratz Arithmetic

Although the interval quotient $\boldsymbol{x}/\boldsymbol{y}$ is undefined in ordinary interval arithmetic when $0 \in \boldsymbol{y}$, an extension to interval arithmetic for this case is useful e.g. in eliminating regions in branch and bound algorithms in which critical points cannot exist. The arithmetic on infinite intervals presented here differs from previous arithmetics on infinite intervals, since it corrects inconsistencies [198]. We call this arithmetic *Kahan–Novoa–Ratz arithmetic*, since Kahan first proposed an arithmetic on infinite intervals in the summer school lecture notes [102], since Kahan used extended interval arithmetic effectively in his continued fraction research, and since Novoa [180] and Ratz [198] separately proposed the corrections that make it consistent for use in nonlinear equations and optimization. In Kahan's arithmetic, the set of real intervals $[a, b] \in \mathbb{IR}$ is augmented by the

[3]For example, in a three-digit decimal system, with numbers of the form 0.123×10^4, one unit in the last place is $.001 \times 10^0 = 10^{-3}$.

[4]cf. §2.1

set of complements $]a,b[= [-\infty,a] \cup [b,\infty]$. In Kahan–Novoa–Ratz arithmetic, division of two ordinary intervals $\boldsymbol{x}$ and $\boldsymbol{y}$ with $0 \in \boldsymbol{y}$ is still defined by the principle in 1.4, operationally as follows:

$$\frac{[\underline{x},\overline{x}]}{[\underline{y},\overline{y}]} = \begin{cases} \boldsymbol{x}[1/\overline{y}, 1/\underline{y}] & \text{if } 0 \notin \boldsymbol{y} \\ [-\infty,\infty] & \text{if } 0 \in \boldsymbol{x} \text{ and } 0 \in \boldsymbol{y}, \\ [\overline{x}/\underline{y},\infty] & \overline{x} < 0 \text{ and } \underline{y} < \overline{y} = 0, \\ [-\infty,\overline{x}/\overline{y}] \cup [\overline{x}/\underline{y},\infty] & \text{if } \overline{x} < 0 \text{ and } \underline{y} < 0 < \overline{y} \\ [-\infty,\overline{x}/\overline{y}] & \text{if } \overline{x} < 0 \text{ and } 0 = \underline{y} < \overline{y}, \\ [-\infty,\underline{x}/\underline{y}] & \text{if } 0 < \underline{x} \text{ and } \underline{y} < \overline{y} = 0. \\ [-\infty,\underline{x}/\underline{y}] \cup [\underline{x}/\overline{y},\infty] & \text{if } 0 < \underline{x} \text{ and } \underline{y} < 0 < \overline{y} \\ [\underline{x}/\overline{y},\infty] & \text{if } \underline{x} < 0 \text{ and } 0 = \underline{y} < \overline{y} \\ \emptyset & \text{if } 0 \notin \boldsymbol{x} \text{ and } 0 = \underline{y}. \end{cases} \tag{1.11}$$

For example, according to Formula (1.11), $[2,3]/[-3,4] = [-\infty,-2/3] \cup [1/2,\infty]$, where $[-\infty,-2/3] \cup [1/2,\infty]$ represents the actual range of x/y, $x \in [2,3]$, $y \in [-3,4]$.

Based on the lecture notes [102], arithmetic on intervals and their complements based on formulas similar to (1.11) is published, fully developed, in [149]. This arithmetic is useful in continued fraction computations, since the reciprocal of an interval complement[5] is an ordinary interval. However, for most[6] purposes here it suffices merely to use Formula (1.11), then to intersect the result with an ordinary interval, to obtain zero, one or two ordinary intervals; an example occurs in Example 1.4 on page 53 below.

Kahan–Novoa–Ratz arithmetic, defined on intervals and their complements, should not be confused with other extensions of real interval arithmetic, designed for different purposes. For example, *Kaucher arithmetic* is used in [217] to obtain both *inner estimates* and *outer estimates*[7] for the solution sets of linear interval systems. Yet a third structure, Markov arithmetic, purported to be useful in reducing the overestimation of ranges with ordinary interval arithmetic, is described in [157, 158]. Each of these arithmetics is at times called *extended interval arithmetic.*

[5] also termed an *extended interval*

[6] except perhaps for evaluation of some special objective functions

[7] An inner estimate of the range of a function is a set contained within the range, while an outer estimate contains the range.

1.1.5 Complex Interval Arithmetic

It is possible to identify rectangles of complex numbers

$$z = \{x + iy \mid x \in \boldsymbol{x} \text{ and } y \in \boldsymbol{y}\}$$

with boxes in $\mathbb{IR}^2$, and to define interval operations componentwise. Thus, a complex interval $\boldsymbol{z}$ is identified with the interval vector $[\boldsymbol{x}, \boldsymbol{y}]^T \sim \boldsymbol{x} + i\boldsymbol{y}$. So interpreted, complex interval operations become

$$\begin{aligned} \boldsymbol{z}_1 + \boldsymbol{z}_2 &= \boldsymbol{x}_1 + \boldsymbol{x}_2 + i(\boldsymbol{y}_1 + \boldsymbol{y}_2) && (1.12) \\ &= [\underline{x}_1 + \underline{x}_2, \overline{x}_1 + \overline{x}_2] + i[\underline{y}_1 + \underline{y}_2, \overline{y}_1 + \overline{y}_2], && (1.13) \\ \boldsymbol{z}_1 \times \boldsymbol{z}_2 &= \boldsymbol{x}_1\boldsymbol{x}_2 - \boldsymbol{y}_1\boldsymbol{y}_2 + i(\boldsymbol{x}_1\boldsymbol{y}_2 + \boldsymbol{x}_2\boldsymbol{y}_1), && (1.14) \end{aligned}$$

where $\boldsymbol{z}_1 = \boldsymbol{x}_1 + i\boldsymbol{y}_1$ and $\boldsymbol{z}_2 = \boldsymbol{x}_2 + i\boldsymbol{y}_2$, where multiplication of the component intervals is according to Formula (1.7), and where $-$ and $\div$ are defined similarly. The problem is that, although the set $\boldsymbol{x}_1\boldsymbol{x}_2 - \boldsymbol{y}_1\boldsymbol{y}_2 + i(\boldsymbol{x}_1\boldsymbol{y}_2 + \boldsymbol{x}_2\boldsymbol{y}_1)$ corresponds to a rectangle in $\mathbb{R}^2$, the actual range

$$\{(x_1 + iy_1)(x_2 + iy_2) \mid x_1 \in \boldsymbol{x}_1,\, y_1 \in \boldsymbol{y}_1,\, x_2 \in \boldsymbol{x}_2,\, y_2 \in \boldsymbol{y}_2\}$$

is less regularly shaped. Thus, the product $\boldsymbol{z}_1\boldsymbol{z}_2$ does *not* represent the exact range, and there is overestimation even in a single elementary operation. This can be viewed as complex interval dependency arising from the relationship given by the Cauchy–Riemann equations.

An alternate arithmetic can be defined on the set of ***disks***, rather than rectangles, in the complex plane. First introduced by Henrici [59], this ***circular arithmetic*** results in less overestimation in multiplication[8], but may be more difficult to implement. It has been proposed [246] that a combination of both rectangular and circular arithmetic can be used, to reduce overall overestimation. Both rectangular and circular arithmetic are explained in [8, Ch. 5].

Despite the intrinsic overestimation, complex interval arithmetic can be useful in practice. Although, with interval arithmetic, problems should be solved in the real domain if possible (rather than recast in the complex plane), rectangular arithmetic, recommended for its simplicity, can be used effectively. If rectangular arithmetic is not implemented as a data type, either intrinsically or through operator overloading[9], functions $w = f(z)$, $f : \mathbb{IC} \to \mathbb{IC}$

[8] in the sense of ratio of area of the interval product to the area of the range of the product operation

[9] cf. §1.4.4

may be explicitly rewritten in terms of their components as $F : \mathbb{IR}^2 \to \mathbb{IR}^2$. For example, to use a multidimensional interval branch and bound method to find all roots of $f(z) = z^2 - 2$ in the complex plane, f could be written as $F(x, y) = [f_1(x, y), f_2(x, y)]^T = [0, 0]^T$, where $f_1(x, y) = x^2 - y^2 - 2$ and $f_2(x, y) = xy$.

1.1.6 Extensions of Standard Functions

An example in §1.1.1 illustrated how interval arithmetic could be used to obtain bounds on the range of a function that could be evaluated as a sequence of the four elementary operations. This section introduces, in elementary terms, some of the fundamental techniques for obtaining the ranges of functions, for libraries to compute interval extensions.

Throughout, the term *standard function* will appear. It will mean a set of functions, such as those specified in the FORTRAN-77 standard, that are bundled with a particular compiler or software package. Libraries for evaluation of such functions must be supplied by the package provider. Such standard functions generally include exponentials, logarithms, and a selection of trigonometric functions and their inverses.

Definition 1.2 *If $f : \mathbb{R} \to \mathbb{R}$ is a function computable as an expression, algorithm or computer program involving the four elementary arithmetic operations, then a* natural interval extension *of f, whose value over an interval $\boldsymbol{x}$ is denoted by $\boldsymbol{f}(\boldsymbol{x})$, is obtained by replacing each occurrence of x by the interval $\boldsymbol{x}$ and by executing all operations according to formulas (1.5) through (1.9).*

It is not hard to show

Theorem 1.2 *If $\boldsymbol{f}$ is any natural interval extension of f, and $\boldsymbol{x} \in \mathbb{IR}$ is contained within the domain of $\boldsymbol{f}$, then $\boldsymbol{f}(\boldsymbol{x})$ contains the range $f^u(\boldsymbol{x})$ of f over $\boldsymbol{x}$.*

This property is crucial to interval computations, so it should be a part of the *definition* of an interval extension.

Definition 1.3 *A function* $\boldsymbol{f} : \mathbb{IR} \to \mathbb{IR}$ *is said to be an* interval extension *of* $f : \mathbb{R} \to \mathbb{R}$ *provided*

$$\{f(x) | x \in \boldsymbol{x}\} \subseteq \boldsymbol{f}(\boldsymbol{x})$$

for all intervals $\boldsymbol{x} \subset \mathbb{IR}$ *within the domain of* $\boldsymbol{f}$. *(Interval extensions are defined similarly for* $F : \mathbb{R}^n \to \mathbb{R}^m$.*)*

In fact, natural interval extensions can be obtained for virtually *any* function that can be represented with a computer program. To this end, sharp bounds on the ranges of the standard functions are desired. Such bounds can be computed from any expansion (rational, Taylor series, etc.) that has an explicit formula for the error term. For example, suppose $\boldsymbol{x} \in \mathbb{IR}$ and $\check{x} \in \boldsymbol{x}$. Then, for any $x \in \boldsymbol{x}$,

$$\cos(x) = \cos(\check{x}) - (x - \check{x})\sin(\check{x}) - \frac{(x - \check{x})^2}{2}\cos(c)$$

for some c between $\check{x}$ and x. Translating to intervals, we obtain the inclusion

$$\cos(x) \in \cos(\check{x}) - (\boldsymbol{x} - \check{x})\sin(\check{x}) - \frac{(\boldsymbol{x} - \check{x})^2}{2}\left[\min_{c \in \boldsymbol{x}}\{\cos(c)\}, \max_{c \in \boldsymbol{x}}\{\cos(c)\}\right].$$

Specifically, if $\boldsymbol{x} = [-0.1, 0.1]$, we may use $\check{x} = 0$ to obtain the interval extension value

$$\begin{aligned} \cos^u([-0.1, 0.1]) &\subseteq \cos(0) - [-0.1, 0.1]\sin(0) - \frac{([-0.1, 0.1]^2)}{2}[0.9, 1] \\ &= 1 - [0, 0.005][0.9, 1] = 1 - [0, 0.005] \\ &= [0.995, 1] = \cos([-0.1, 0.1]), \end{aligned}$$

whereas the range of cos over $[-0.1, 0.1]$, given to ten digits and rounded out, is $[0.995004164, 1]$.

Remark 1.1 *Above,* $[-0.1, 0.1]^2$ *is given as* $[0, 0.01]$, *not as* $[-0.1, 0.1][-0.1, 0.1] = [-0.01, 0.01]$. *Generally, integer powers should be viewed as standard functions with exact ranges (to within roundout), not as repeated multiplication.*

Ranges of the standard functions can thus be approximated with any desired accuracy, within the limits of the floating point system. Monotonicity can also

be used, as appropriate, to piece together the range from efficiently obtained narrow enclosures for the endpoints of $\boldsymbol{x}$. Argument reductions and other considerations, such as those explained in [32], are also important in construction of libraries of interval values of functions. Taylor series are often used, due to their simplicity and ease of including the truncation error term. In fact, Braune [23] and Krämer [138] describe how both roundoff and truncation error can be taken into account in Taylor series values, given machine parameters, so that only floating point arithmetic is necessary when accumulating the series terms, to obtain rigorous bounds on ranges. Along other lines, [15] contains recipes for computing ranges of the various standard functions, based on rational approximations.

There are additional issues associated with interval values of certain standard functions. For example, the range of the inverse trigonometric functions should be viewed as an infinite sequence of intervals in some applications, and the range of some trigonometric functions contains branch points. However, most of the standard functions available with programming language compilers for scientific computation can be considered as functions $\omega : \mathbb{IR} \to \mathbb{IR}$ such that $\omega(\boldsymbol{x})$ is, to within roundout, the exact range of ω over $\boldsymbol{x}$.

Detailed discussion of techniques for computing the ranges of functions appears in [39]. Particulars on construction of the `FORTRAN-77` library described in §2.1 appears in [86, 124, 123].

1.1.7 Properties of Interval Extensions

Some elementary properties of interval extensions were already introduced as Theorems 1.1 and 1.2. Amplification and additional properties useful in formulating and coding objective functions and systems of equations appear here.

Theorem 1.3 *Suppose that an interval extension $\boldsymbol{F}(\boldsymbol{X})$ of a function $F : \mathbb{R}^n \to \mathbb{R}^m$ is obtained by a sequence of computation of rigorous bounds on ranges $\omega_j(\boldsymbol{x})$ of standard functions $\{\omega_j\}_{j=1}^{p}$ as in §1.1.6 interspersed with outwardly rounded interval arithmetic involving the four elementary operations, as in formulas (1.5) through (1.9). Then this natural interval extension $\boldsymbol{F}(\boldsymbol{X})$ contains the range of F over the box $\boldsymbol{X}$.*

An early criticism of interval arithmetic was that the bounds on the ranges so obtained were too wide or pessimistic to be of value. This *is* frequently the

case when interval arithmetic is applied *naively*, but a good understanding of its properties leads to appropriate and effective algorithms. The next several facts deal with *how sharply* interval extensions enclose the range of a function.

Theorem 1.4 *Suppose $\phi : \mathbb{R}^n \to \mathbb{R}$, is formally written as a sequence of computations of standard functions $\{\omega_j\}_{j=1}^{q}$ interspersed with the four basic arithmetic operations. Suppose that in this expression, each of the n variables $\{x_i\}_{i=1}^{n}$ occurs formally only once. Then, if $\phi(\boldsymbol{X})$ is evaluated with exact interval arithmetic and computation of the exact ranges of each ω_j, the resulting interval enclosure will be the exact range $\phi(\boldsymbol{X})$.*

Thus, the range e.g. of $f(x) = x^2 - 2$ over $[-2, 2]$ is $[-2, 2]^2 - 2 = [0, 4] - [2, 2] = [-2, 2]$, and the range of $\phi(x_1, x_2) = x_1 x_2$ over the box $\big([-1, 1], [-1, 1]\big)^T$ is $[-1, 1][-1, 1] = [-1, 1]$.

Theorem 1.4 and subdistributivity (Theorem 1.1) suggest that, to maximize the sharpness of an interval extension, the defining expressions or algorithms should be rewritten to minimize the number of occurrences of each variable or subexpression. Indeed, that is a reasonable heuristic in many[10] instances.

Computing the exact range of an arbitrary function ϕ over a box $\boldsymbol{X}$ is similar to optimizing ϕ over $\boldsymbol{X}$. However, in both cases asymptotic properties of interval bounds on the range of ϕ as $\|\mathrm{w}(\boldsymbol{X})\| \to 0$ are helpful.

Definition 1.4 *Let $\boldsymbol{F}(\boldsymbol{X})$ denote an interval extension of $F : \mathbb{R}^n \to \mathbb{R}^m$ evaluated over a box $\boldsymbol{X}$, and let $\boldsymbol{F}^u(\boldsymbol{X})$ denote the exact range of f over $\boldsymbol{X}$. If there is a constant K, independent of the box $\boldsymbol{X}$ such that*

$$\mathrm{w}\big(\boldsymbol{F}(\boldsymbol{X})\big) - \mathrm{w}\big(\boldsymbol{f}^u(\boldsymbol{X})\big) \leq K \mathrm{w}(\boldsymbol{X})^\alpha \tag{1.15}$$

for all boxes $\boldsymbol{X}$ with $\mathrm{w}(\boldsymbol{X})$ sufficiently small and fixed $\alpha > 0$, then we say that $\boldsymbol{F}$ is an order α inclusion function for F. *When α is 1 or 2, we call the inclusion* first order *or* second order, *respectively.*

For clarity in what follows, we now restate Definition 1.2 more generally.

[10] but not all

Definition 1.5 *If $F : \mathbb{R}^n \to \mathbb{R}^m$ is a function computable as an expression, algorithm or computer program involving the four elementary arithmetic operations interspersed with evaluations of standard functions $\{\omega_j\}_{j=1}^q$, then a* natural interval extension *of F, whose value over an interval vector $\boldsymbol{X}$ is denoted by $\boldsymbol{F}(\boldsymbol{X})$, is obtained by replacing each occurrence of each component x_i of X by the corresponding interval component $\boldsymbol{x}_i$ of $\boldsymbol{X}$, by executing all operations according to formulas (1.5) through (1.9), and by computing the exact ranges of the standard functions.*

Remark 1.2 *We will refer to natural interval extensions even when machine arithmetic is used. However, the standard function ranges are then merely enclosures of the actual ranges that are sharp to within roundout error and the bounds on the truncation error. The order of such an interval extension is then only approximate, and not valid when the widths of the independent variables are on the order of the distance between machine numbers.*

Theorem 1.5 *([8], [165], [190, §1.5]) Natural interval extensions are first order.*

Second order extensions are somewhat more desirable, and indeed, almost indispensable in some contexts. This has been pointed out in [114] for solving nonlinear systems and in [52], [125] and [51] for optimization. Second-order extensions may be obtained, as in the examples in §1.1.6, by series expansions and bounding the range of derivatives.

Definition 1.6 *Suppose $\phi : \mathbf{D} \subseteq \mathbb{R}^n \to \mathbb{R}$ has continuous derivatives, $\boldsymbol{X} \subseteq \mathbf{D}$ and $\check{X} \in \boldsymbol{X}$. Then the* mean value extension *for ϕ over $\boldsymbol{X}$ and centered at $\check{X}$ is defined by*

$$\boldsymbol{\phi}_2(\boldsymbol{X}, \check{X}) := \phi(\check{X}) + \boldsymbol{\nabla\phi}(\boldsymbol{X})(\boldsymbol{X} - \check{X}), \tag{1.16}$$

where $\boldsymbol{\nabla\phi}(\boldsymbol{X})$ is a componentwise interval enclosure for the range of $\nabla\phi$ over $\boldsymbol{X}$.

It follows from the mean value theorem and properties of interval arithmetic that $\boldsymbol{\phi}_2(\boldsymbol{X}, \check{X})$ is an enclosure for the range of ϕ over $\boldsymbol{X}$. Furthermore:

Theorem 1.6 *(originally in [142]; see also [190, Ch. 2, esp. Theorem 2.3]) Suppose that the components of $\boldsymbol{\nabla\phi}$ are interval extensions of $\nabla\phi$ of order at least one. Then $\boldsymbol{\phi}_2$ is an order-2 interval extension of ϕ.*

For example, suppose $\phi(x) = x^2 - 2$, $x = [0,1]$ and $\check{x} = 1/2$, so that $\nabla\phi(x) = 2x$ is the exact range of the derivative of ϕ over x. Then the natural interval extension obtained from the particular expression $x^2 - 2$ is, as in §1.1.1, $\phi_n([0,1]) = [-1,1]$, while the mean value extension is

$$\begin{aligned} \phi_2([0,1],1/2) &= -\frac{1}{4} + [0,2]\left([0,1] - \frac{1}{2}\right) \\ &= [-5/4, 3/4] \end{aligned}$$

Thus $w(\phi_n([0,1])) = w(\phi_2([0,1],1/2)) = 2$, whereas the width of the range is $w(\phi^u([0,1])) = 1/4$. This illustrates that a second order extension may not be superior to a first order extension (and may actually be substantially wider than a natural extension) when the widths of the arguments are large. It also illustrates that the extensions are *different*: $\phi_n([0,1]) \cap \phi_2([0,1],1/2) = [-1,1] \cap [-5/4, 3/4] = [-1, 3/4]$ is also an enclosure for the range of ϕ over $[0,1]$ that is sharper than either $\phi([0,1])$ or $\phi_2([0,1],1/2)$ separately.

If on the other hand $x = [0.49, 0.51]$ and $\check{x} = 0.5$, then the natural interval extension is $\phi_n([0.49,0.51]) = [-0.2699, -0.2299]$, $\phi_2([0.49,0.51],0.5) = [-0.2602, -0.2398]$, $\phi^u([0.49,0.51]) = [-0.25, -0.2499]$, and a hint of the convergence behavior of ϕ_n and ϕ_2 is seen.

Unfortunately, although it would be desirable in global optimization algorithms, it appears difficult to obtain interval extensions of order higher than 2 for arbitrary functions; cf. [41] and the discussion in [51, Ch. 5]: finding a third order extension may require solving a quadratic programming problem which may not always have all positive eigenvalues.

The mean value form is a special case of a *centered form* for an interval inclusion for a function. Alternate centered forms and a discussion of the theory appear in [190]. Such forms may be appropriate to obtain tighter bounds in specific cases. For our purposes[11] the natural extension ϕ and mean value extension ϕ_2 will be adequate. However, we will use interval vectors other than $\nabla\phi(X)$ in Equation (1.16) to obtain tighter but still-rigorous bounds; cf. §1.3 below. In algorithms, a particular natural interval extension will be denoted by ϕ (or f or F), while a generalized mean value extension will be denoted by ϕ_2, to imply that any first order and any second order extension will do.

[11] since we will supply generic interpreters to compute ϕ and ϕ_2, since order higher than 2 cannot be achieved merely with more terms, and since other mechanisms in global optimization algorithms can supply tighter bounds

A final property, often assumed in the literature to facilitate convergence proofs, is

Definition 1.7 *An interval extension ϕ is* inclusion monotonic *or* inclusion isotonic *provided $\boldsymbol{Y} \subseteq \boldsymbol{X}$ implies $\phi(\boldsymbol{X}) \subseteq \phi(\boldsymbol{Y})$.*

Theorem 1.7 *Natural interval extensions as in Definition 1.5 are inclusion monotonic, provided exact interval arithmetic is used and exact ranges are computed for the standard functions.*

Inclusion monotonicity has been suggested as a stopping criterion for interval iterative algorithms: violation of it indicates that roundout error is predominating.

Although inclusion monotonicity is convenient, in our opinion the asymptotic properties (i.e. the order) are more crucial. Some of our constructions and algorithmic processes, although effective, lead to interval extensions that are not inclusion monotonic.

1.1.8 Exercises

1. On page 4, it was seen that the natural interval extension corresponding to $f_1(x) = x(x-1)$ was sharper than the natural interval extension corresponding to $f_2(x) = x^2 - x$ over the interval $[0,1]$, even if the exact range of x^2 is used, i.e. $\boldsymbol{f}_1([0,1]) \subset \boldsymbol{f}_2([0,1])$. Is this also true for the interval $[-1,1]$, i.e. is $\boldsymbol{f}_1([-1,1]) \subset \boldsymbol{f}_2([-1,1])$? What can you say about Horner's method and factoring interval extensions? Can you make recommendations concerning which natural interval extensions are appropriate or how they should be used? You may wish to consult [192].

2. *(roundout error)* In rounded interval arithmetic, an increase in interval widths due to directed roundings occurs under the same circumstances as roundoff error in traditional floating point computations. Consider an outwardly rounded interval arithmetic based on directed roundings with a four decimal digit floating point system. (For example, the outwardly rounded result of $[.1234 \times 10^0, .1234 \times 10^0] + [.1111 \times 10^{-1}, .1111 \times 10^{-1}]$ would be $[.1345 \times 10^0, .1346 \times 10^0] \ni .13451$.) Now examine the difference quotient $Q(h) = \big(f(x_0 + h) - f(x_0)\big)/h$ with $f(x) = x^2$ and $x_0 = 1$.

Representing x_0 as a four-digit decimal interval $[.1000 \times 10^0, .1000 \times 10^0]$ and using outwardly rounded arithmetic, form a table whose columns are h, $Q(h)$ and $\mathrm{w}(Q(h))$ and whose rows correspond to $h = .1$, $h = .05$, $h = .01$, and $h = .005$. Perform a directed rounding into four digit intervals after each of the three operations ($x \to x^2$, $y - x$, and y/x).

3. For the example on page 16, form a table with columns $\boldsymbol{x}$, ϕ, ϕ_2, ϕ^u, $\mathrm{w}(\phi)$, $\mathrm{w}(\phi_2)$ and $\mathrm{w}(\phi^u)$ for $\boldsymbol{x}$ centered about $\check{x} = 1/2$ and with widths 1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}. Referring to Definition 1.4, form and label additional columns of the table containing ratios of differences of widths to illustrate from the computations that ϕ is order 1 and ϕ_2 is order 2.

4. Prove Theorem 1.7.

   Hint: This may be done by induction on the number of elementary operations (including evaluation of standard functions).

1.2 INTERVAL LINEAR SYSTEMS

Within the scope of this book, interval linear systems of equations are important in interval Newton methods, the primary device for verifying existence and uniqueness of roots and critical points. In particular, interval Newton methods can be viewed as computational analogues of the Brouwer fixed point theorem, and the solution set of an interval linear system must be bounded to carry out an interval Newton method. Also, interval linear systems of equations are useful in computing rigorously verified bounds on the actual solutions of real systems of equations; see [70, Chapter 10] and the references therein.

Generally, interval linear systems are of the form

$$\boldsymbol{A}X = \boldsymbol{B}, \tag{1.17}$$

where $\boldsymbol{A} \in \mathbb{IR}^{n \times n}$ and $\boldsymbol{B} \in \mathbb{IR}^n$, although the form most commonly encountered in interval Newton methods is

$$\boldsymbol{A}(X - \check{X}) = \boldsymbol{B}_0, \tag{1.18}$$

where $\check{X} \in \mathbb{R}^n$ and, usually, $\|\mathrm{w}(\boldsymbol{B}_0)\|$ is small; the special form of Equation (1.18) can be exploited, as explained in Chapter 3 below.

The concept of regularity is important in proving uniqueness of solutions.

Definition 1.8 *An interval matrix $\boldsymbol{A}$ is said to be* regular *provided each $A \in \boldsymbol{A}$ is non-singular.*

The solution set of an interval linear system (a concept to be made precise below) is bounded if and only if $\boldsymbol{A}$ is regular.

Also, interval versions of inverse positivity, M-matrices, and H-matrices (a generalization of M-matrices) provide a rich theory for interval linear systems; see [175] for an overview.

1.2.1 Types of Solution Sets

The *solution set* can be defined in various ways. Shary ([215, p. 7], [217]) defines and analyzes three types of solution sets, all potentially useful in applications, as follows.

Definition 1.9 *(Solution sets)*

1. *The* united solution set *of Equation (1.17) is that set $\sum_{\exists\exists}(\boldsymbol{A}, \boldsymbol{B}) \subseteq \mathbb{R}^n$ such that, if $X \in \sum_{\exists\exists}(\boldsymbol{A}, \boldsymbol{B})$, there exists an $A \in \boldsymbol{A}$ and a $B \in \boldsymbol{B}$ such that $AX = B$. The united solution set will also be denoted simply by $\Sigma(\boldsymbol{A}, \boldsymbol{B})$.*
2. *The* controlled solution set *of Equation (1.17) is that set $\sum_{\exists\forall}(\boldsymbol{A}, \boldsymbol{B}) \subseteq \mathbb{R}^n$ such that, if $X \in \sum_{\exists\forall}$, for every $B \in \boldsymbol{B}$ there exists an $A \in \boldsymbol{A}$ such that $AX = B$.*
3. *The* tolerable solution set *of Equation (1.17) is that set $\sum_{\forall\exists}(\boldsymbol{A}, \boldsymbol{B}) \subseteq \mathbb{R}^n$ such that, if $X \in \sum_{\forall\exists}$, for every $A \in \boldsymbol{A}$ there exists a $B \in \boldsymbol{B}$ such that $AX = B$.*

In most of the literature, the united solution set $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ is called simply the *solution set.* Since this is the solution set of main interest in this book, this terminology will also be adopted here.

In general $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ is a non-convex polygonal region; pictures of particular $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ appear in [217], on the jacket of [175], in [175, fig. 3.2, p. 92], in [77,

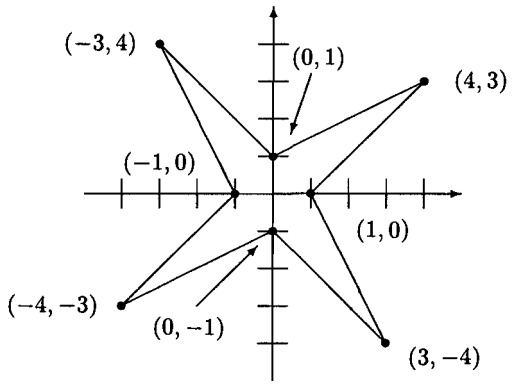


Figure 1.1 The united solution set $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ for the system (1.19)

fig. 4.1, p. 27], etc. For example, consider the interval matrix

$$\boldsymbol{A} = \begin{pmatrix} [\ 2, 4] & [-2, 1] \\ [-1, 2] & [\ 2, 4] \end{pmatrix}$$

and right-hand-side vector (1.19)

$$\boldsymbol{B} = \begin{pmatrix} [-2, 2] \\ [-2, 2] \end{pmatrix}.$$

The actual united solution set $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ to this system is shown in Figure 1.1.

Computation of $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ is in general an NP-complete problem; see [203], but is inexpensive when the system has certain special properties, as for example in [175, Theorem 3.6.7, p. 108].

Although $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ is not an interval vector, the interval vector formed from the bounds on the coordinates of the solution set is usually considered. It is called the *solution hull* or *interval hull*, and is denoted by $\square\Sigma(\boldsymbol{A}, \boldsymbol{B})$. Computing the exact solution hull is also in general NP-complete [203], but fortunately, it is computationally inexpensive to obtain interval vectors $\boldsymbol{X} \supseteq \square\Sigma(\boldsymbol{A}, \boldsymbol{B})$. Such vectors are called *outer estimates* to $\Sigma(\boldsymbol{A}, \boldsymbol{B})$. Similarly, *inner estimates*, interval vectors $\boldsymbol{X}$ with the property $\boldsymbol{X} \subseteq \square\Sigma(\boldsymbol{A}, \boldsymbol{B})$, are also useful to compute. For example, Rump [208, §4.2] computes both inner estimates and outer estimates to determine the sharpness of each in approximating the hull, and for sensitivity analysis. A different, interesting way of computing inner estimates appears in [148] and [217].

1.2.2 Solution Algorithms

The three most common methods for computing outer estimates to $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ are *interval Gaussian elimination*, the *interval Gauss–Seidel method*, and the *Krawczyk method*. The interval Gauss–Seidel method is sometimes called the *single-step method* or *Einzelschrittverfahren*. The Krawczyk method and interval Gauss–Seidel method each require an initial guess vector, whereas interval Gaussian elimination does not.

It usually is necessary to *precondition* the System (1.17) by a point matrix $Y \in \mathbb{R}^{n \times n}$, for the Krawczyk, interval Gaussian elimination, or interval Gauss–Seidel method to be effective. Thus, the algorithms are applied, not to the original System (1.17) but to

$$YAX = YB, \text{ i.e. } \boldsymbol{M}X = \boldsymbol{C} \text{ where } \boldsymbol{M} = YA \text{ and } \boldsymbol{C} = YB, \tag{1.20}$$

where Y is chosen to make the solution set of $\boldsymbol{M}X = \boldsymbol{C}$ somehow easier to bound (such as when Y in some sense is like an inverse A^{-1}, so $\boldsymbol{M}$ somehow approximates a diagonal matrix).

Theorem 1.8 (cf. [175, §4.1, esp. fig. 4.1]) *Suppose* $Y \in \mathbb{R}^{n \times n}$, $\boldsymbol{A} \in \mathbb{IR}^{n \times n}$, $\boldsymbol{B} \in \mathbb{IR}^n$ *and* $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ *is as in Definition 1.9. Then*

$$\Sigma(Y\boldsymbol{A}, Y\boldsymbol{B}) = \Sigma(\boldsymbol{M}, \boldsymbol{C}) \supseteq \Sigma(\boldsymbol{A}, \boldsymbol{B}).$$

Even though preconditioning in general increases the size of the actual solution set, and thus makes bounds obtained on the solution of the original System (1.17) less sharp, it nonetheless allows the Krawczyk, Gauss–Seidel, or interval Gaussian elimination method to compute tighter interval bounds for the components of the solution set. Preconditioning is thus in general used, except in special cases mentioned in §3 below.

Interval Gaussian elimination, also called the *interval Gauss algorithm*, proceeds similarly to a variant of floating point Gaussian elimination.

Algorithm 1 (Interval Gaussian Elimination)

INPUT: The matrix $\boldsymbol{M} = [m_{i,j}]_{i,j=1}^n \in \mathbb{IR}^{n \times n}$ and right-hand-side vector $\boldsymbol{C} = [c_1, \ldots, c_n]^T \in \mathbb{IR}^n$ for the preconditioned interval system $\boldsymbol{M}X = \boldsymbol{C}$.

OUTPUT: Either the information "failure" or bounds $\mathbf{GE}(\boldsymbol{A}, \boldsymbol{B}) = \tilde{\boldsymbol{X}} = (\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n) \supseteq \Sigma(\boldsymbol{M}, \boldsymbol{C}) \supseteq \Sigma(\boldsymbol{A}, \boldsymbol{B})$.

1. (Factorization phase) DO *for* $i = 1$ *to* $n - 1$*:*

   (a) IF $0 \in m_{i,i}$ THEN EXIT *with failure.*

   (b) DO *for* $j = i + 1$ *to* n *and for* $k = i + 1$ *to* n*:*

   $$m_{j,k} \leftarrow m_{j,k} - (m_{j,i}/m_{i,i})\, m_{i,k}.$$

   END DO

   (c) *For* $j = i + 1$ *to* n*:* $c_j \leftarrow c_j - (m_{j,i}/m_{i,i})\, c_i$

   END DO

2. (Solution phase)

   (a) $c_{n,n} \leftarrow c_n/m_{n,n}$.

   (b) DO *for* $i = n - 1$ *to* 1 *by* -1*:*

   $$c_i \leftarrow \left(c_i - \sum_{j=i+1}^{n} m_{i,j} c_j\right) \Big/ m_{i,i}$$

   END DO

3. EXIT

End Algorithm 1

Although the interval Gauss–Seidel algorithm requires an initial guess box $\boldsymbol{X}$, it sometimes leads to sharper enclosures of the solution set than interval Gaussian elimination. Also, since the interval Gauss–Seidel method proceeds coordinate by coordinate, it may produce sharper bounds on certain coordinates when interval Gaussian elimination fails, even when the interval matrix $\boldsymbol{A}$ contains singular matrices[12], or when $\boldsymbol{A} \in \mathbb{R}^{p \times n}$ with $p \neq n$. This capability for rectangular systems is useful in handling e.g. constraints, or parametrized systems, as in [130]. In fact, even though $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ must be unbounded when $\boldsymbol{A}$ has more columns than rows, bounds on the portion of $\Sigma(\boldsymbol{A}, \boldsymbol{B})$ lying within a particular box can still often be sharpened. See Figure 1.2.

Algorithm 2 (Preconditioned interval Gauss–Seidel method)

INPUT: The matrix $\boldsymbol{A} \in \mathbb{IR}^{n \times n}$ and right-hand-side vector $\boldsymbol{B} \in \mathbb{IR}^n$ for the interval system $\boldsymbol{A}X = \boldsymbol{B}$. Also input bound constraints $\boldsymbol{X} = (x_1, \ldots, x_n)^T$ bounding the desired portion of the solution set.

[12]Interval Gaussian elimination must fail in this case, since ordinary Gaussian elimination fails on singular matrices, and the intermediate results of the interval computation must contain all of the corresponding point results.

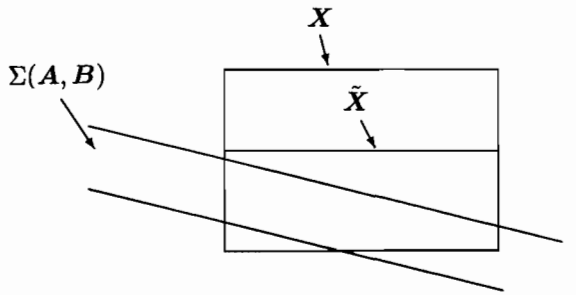


Figure 1.2 $\Sigma(\boldsymbol{A},\boldsymbol{B}) \cap \boldsymbol{X}$ can still be bounded when $AX = B$ is underdetermined.

OUTPUT: Either the information "solution does not intersect $\boldsymbol{X}$" or new bounds $\mathbf{GS}(\boldsymbol{X}) = \tilde{\boldsymbol{X}} = [\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n]^T \supseteq \Sigma(Y\boldsymbol{A}, Y\boldsymbol{B}) \cap \boldsymbol{X} \supseteq \Sigma(\boldsymbol{A},\boldsymbol{B}) \cap \boldsymbol{X}$.

DO *for* $i = 1$ *to* n.

1. *Compute the* i*-th row* Y_i *of the preconditioner.*
2. *Let* $\boldsymbol{A}_j$ *denote the* j*-th column of* $\boldsymbol{A}$. *Compute*

$$\tilde{\boldsymbol{x}}_i = \frac{Y_i\boldsymbol{B} + \sum_{j=1}^{i-1}(Y_i\boldsymbol{A}_j)\tilde{\boldsymbol{x}}_j + \sum_{j=i+1}^{n}(Y_i\boldsymbol{A}_j)\boldsymbol{x}_j}{Y_i\boldsymbol{A}_i}$$

3. IF $\tilde{\boldsymbol{x}}_i \cap \boldsymbol{x}_i = \emptyset$,
   THEN
   EXIT, *signaling "solution does not intersect* $\boldsymbol{X}$*."*
   ELSE
   Replace $\tilde{\boldsymbol{x}}_i$ *by* $\tilde{\boldsymbol{x}}_i \cap \boldsymbol{x}_i$.
   END IF

END DO

End Algorithm 2

If the underlying system is Equation (1.18) instead of Equation (1.17), then $\tilde{\boldsymbol{x}}_j$ and $\boldsymbol{x}_j$ are replaced by $\tilde{\boldsymbol{x}}_j - \check{\boldsymbol{x}}_j$ and $\boldsymbol{x}_j - \check{\boldsymbol{x}}_j$ in the right member in step 2 of Algorithm 2, respectively. In that case, considered in §3 below, much advantage can be taken of the system structure.

The sequential nature in Algorithm 2 is not critical. In particular, the algorithm will still function qualitatively similarly (although perhaps with a different convergence speed) if some of the $\tilde{\boldsymbol{x}}$ are exchanged for $\boldsymbol{x}$, and visa versa, in the right member of Step 2. Thus, as with floating point Jacobi, Gauss–Seidel or SOR iterations, the individual passes through the loop on i may be done in parallel, asynchronously. Wherever we refer to the "interval Gauss–Seidel" method, unless analyzing exact convergence rates, we will also be referring to such parallel variants.

The third method of note for computing rigorous bounds on $\square\Sigma(\boldsymbol{A}, \boldsymbol{B})$, the *Krawczyk method*, is based on considering the fixed point iteration $X \leftarrow X - YF(X)$, where Y is presumably an approximation to the inverse of the Jacobi matrix of F at some point [164]. Its convergence properties are particularly simple to analyze, and much of the theoretical literature concerns this method. Its iteration formula is

$$\mathbf{K}(\boldsymbol{X}) = Y\boldsymbol{B} + [I - Y\boldsymbol{A}]\,\boldsymbol{X}, \tag{1.21}$$

where, in the literature, Y is often (but not necessarily) taken to be the inverse of the matrix of midpoints of the elements of $\boldsymbol{A}$.

The convergence rate of the Krawczyk method depends on $\|\boldsymbol{\Delta}\| = \|\,[I - Y\boldsymbol{A}]\,\|$, and Rump uses $\boldsymbol{\Delta}$ [208, §4.2] to obtain both inner bounds and outer bounds on $\square\Sigma(\boldsymbol{A}, \boldsymbol{B})$. Such bounds are attractive, since the actual amount of overestimation of $\square\Sigma(\boldsymbol{A}, \boldsymbol{B})$ by the method is then bracketed.

However,

$$\mathbf{GS}(\boldsymbol{X}) \subseteq \mathbf{K}(\boldsymbol{X}) \tag{1.22}$$

(see [175, 4.3.5 and 4.3.6]), so, except for special applications, the Gauss–Seidel method is preferable in practice.

The following theorem, originally proved by various authors (see [175] and the references therein), forms the basis of most automatic verification procedures.

Theorem 1.9 (Computational existence and uniqueness)

1. $[]\Sigma(\boldsymbol{A}, \boldsymbol{B}) \subseteq \mathbf{GE}(\boldsymbol{A}, \boldsymbol{B})$.
2. $[]\Sigma(\boldsymbol{A}, \boldsymbol{B}) \cap \boldsymbol{X}$ *is contained in each of* $\mathbf{GS}(\boldsymbol{X})$ *and* $\mathbf{K}(\boldsymbol{X})$.
3. *If* $\mathbf{GS}(\boldsymbol{X}) \subset \mathrm{int}(\boldsymbol{X})$ *or* $\mathbf{K}(\boldsymbol{X}) \subset \mathrm{int}(\boldsymbol{X})$, *then for each* $A \in \boldsymbol{A}$ *and* $B \in \boldsymbol{B}$, *the system* $AX = B$ *has a unique solution in* $\boldsymbol{X}$.

Considerations related to part 3 of Theorem 1.9 will be considered in more detail on p. 63 ff.

When discussing interval Newton methods in §3 below, variants of interval Gaussian elimination and the interval Gauss–Seidel method corresponding to Equation (1.18) will be used.

1.2.3 Exercises

The following problems refer to System (1.19), with united solution set in Figure 1.1. The preconditioning matrix Y to be used here is the inverse of $Y^{-1} = \begin{pmatrix} 3 & -0.5 \\ 0.5 & 3 \end{pmatrix}$. ($Y^{-1}$ is the matrix of midpoints of elements of $\boldsymbol{A}$.)

1. Use preconditioned interval Gaussian elimination (Algorithm 1) to compute bounds on the components of the united solution set in Fig. 1.1. Compare the widths of the component intervals and the total volume contained within the computed box to $[]\Sigma(\boldsymbol{A}, \boldsymbol{B})$ and $\Sigma(\boldsymbol{A}, \boldsymbol{B})$.
2. Repeat Problem 1 with the preconditioned interval Gauss–Seidel method (Algorithm 2) in place of interval Gaussian elimination. Start with the interval vector $\boldsymbol{X} = ([-10, 10], [-10, 10])^T$; do three sweeps of the methods.
3. Apply the Krawczyk method (Equation (1.21)) to the preconditioned system $Y\boldsymbol{A}\boldsymbol{X} = Y\boldsymbol{B}$. Start with initial interval vector

$$\boldsymbol{X} = ([-10, 10], [-10, 10])^T.$$

Compare with the results of Problem 2.

4. *(This problem is more involved, and may require some extra study.)* Compute and graph the exact solution set $\Sigma(Y\boldsymbol{A}, Y\boldsymbol{B})$.

   (Hint: The paper [181], the description and references in [201], or the description in [175, Ch. 6 and §3.4] may be helpful.)

1.3 DERIVATIVES AND SLOPES

A primary use of interval derivative information is in bounding ranges or variations in functions over regions. This was seen directly in §1.1.7, Definition 1.6, in which a second order interval extension was derived. Such information is also crucial in *interval Newton methods*, which in turn are central to fast and memory-efficient verified global optimization algorithms.

However, it is not always necessary to use interval extensions of the derivative. In fact, there are two techniques that actually lead to tighter bounds. To understand these techniques, it is useful to introduce the concept of a *Lipschitz matrix* and a *slope matrix* for a function $\phi : \mathbb{R}^n \to \mathbb{R}$. There is also a subsidiary technique due to Hansen, leading to slope matrices, that gives narrower bounds for multivariate problems.

1.3.1 Interval Derivatives and Slope Matrices

Definition 1.10 ([175, p. 174], etc.) *Let $F : \mathbb{R}^n \to \mathbb{R}^m$. The matrix $\boldsymbol{A}$ is said to be a* Lipschitz matrix *for F over $\boldsymbol{X}$ provided, for every $X \in \boldsymbol{X}$ and $Y \in \boldsymbol{X}$, $F(X) - F(Y) = A(X - Y)$ for some $A \in \boldsymbol{A}$.*

Consider

Example 1.1 *Let*

$$\phi(x_1, x_2) = x_1^3/3 + x_2^3/3 + x_1.$$

Then a Lipschitz matrix for ϕ over the box $\boldsymbol{X} = ([-.5, .5], [-.5, .5])^T$ is any natural interval extension for the gradient: $\boldsymbol{A} = \left([-.5, .5]^2 + 1, [-.5, .5]^2\right) = \left([1, 1.25], [0, .25]\right)$, while the corresponding mean value extension for ϕ, centered at $\check{X} = (0, 0)^T$, is

$$\boldsymbol{\phi}_2(\boldsymbol{X}, \check{X}) = 0 + \left([1, 1.25], [0, .25]\right) \begin{pmatrix} [-.5, .5] \\ [-.5, .5] \end{pmatrix} = [-.75, .75].$$

In fact, if $F : \boldsymbol{X} \subset \mathbb{IR}^n \to \mathbb{R}^m$, then a component-by-component interval extension of the Jacobi matrix of F constitutes a Lipschitz matrix for F over $\boldsymbol{X}$.

Definition 1.11 *If $F : \boldsymbol{X} \subset \mathbb{IR}^n \to \mathbb{R}^m$, then $F'(\boldsymbol{X})$ will denote any interval matrix whose components are natural interval extensions over $\boldsymbol{X}$ of corresponding components of the Jacobi matrix of F. Any such matrix will be called an* interval Jacobi matrix.

An alternate, weaker property is the following.

Definition 1.12 *Let $F : \mathbb{R}^n \to \mathbb{R}^m$. The matrix $\boldsymbol{A}$ is said to be an* interval slope matrix *(or, more generally, a slope set) for F over $\boldsymbol{X}$ and centered on the interval vector $\check{\boldsymbol{X}}$ if, for every $X \in \boldsymbol{X}$ and $\check{X} \in \check{\boldsymbol{X}}$,*

$$F(X) - F(\check{X}) = A(X - \check{X}) \quad \textit{for some } A \in \boldsymbol{A}.$$

Any smallest such set of matrices satisfying this condition will be denoted by $\mathbf{S}^\natural(F, \boldsymbol{X}, \check{\boldsymbol{X}})$. An interval vector that contains $\mathbf{S}^\natural(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ (generally a good computed outer estimate) will be denoted by $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$.

Often, but not always, $\check{\boldsymbol{X}}$ is a point or a very small box.

Univariate Slopes

The difference between derivatives and slopes is most easily introduced with a one-dimensional example. For illustration purposes only, suppose a second-order extension to $f(x) = x^2$ is desired over the interval $[0, 2]$. Then the range of the derivative over $\boldsymbol{x} = [0, 2]$ is $\boldsymbol{a} = [0, 4]$, and the mean value extension, an enclosure for the range of $f(x) = x^2$ over $[0, 2]$, is:

$$f_2([0, 2]) = f(1) + \boldsymbol{f}'([0, 2])([0, 2] - 1) = 1 + [0, 4][-1, 1] = [-3, 5].$$

In contrast,

$$\boldsymbol{a} = \left\{ \mu | \mu = \frac{x^2 - 1}{x - 1} \text{ for } x \in [0, 2] \right\} = [1, 3] = \mathbf{S}^\natural(f, [0, 2], 1)$$

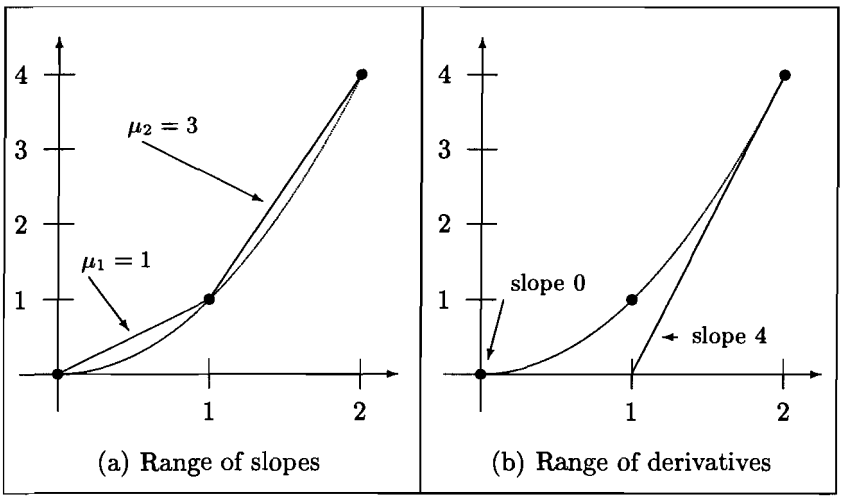


Figure 1.3 The difference between the interval slope $\mathbf{S}^{\natural}(f, \boldsymbol{x}, \check{x})$ and the derivative range $\boldsymbol{f}'_u$ for $\boldsymbol{x} = [0, 2]$, $\check{x} = 1$ and $f(x) = x^2$

is a slope set for $f(x) = e^x$ over $\boldsymbol{x} = [-1, 1]$, centered[13] at $\check{x} = 0$. Because of that, $f_s(\boldsymbol{x}) = f(0) + [1, 2](\boldsymbol{x} - 1)$ is a second order extension of $f(x) = x^2$ for $\boldsymbol{x} \subseteq [0, 2]$, and an enclosure for the range of f over $[0, 2]$ is

$$f_s([0, 2]) = 1 + [1, 3][-1, 1] = [-2, 4] \subset [-3, 5].$$

Figure 1.3 illustrates this relationship.

Although an interval extension would not actually be used for $f(x) = x^2$ (since the exact range is computable), this example illustrates a general relationship between derivatives and slopes, stated as

Theorem 1.10 *Suppose $f : \mathbb{R} \to \mathbb{R}$, $\boldsymbol{x} \in \mathbb{IR}$ is arbitrary, and $\check{x} \in \boldsymbol{x}$. Suppose $\boldsymbol{f}'_u$ is the range of f' over $\boldsymbol{x}$. Then*

$$\lim_{\mathrm{w}(\boldsymbol{x}) \to 0} \frac{\mathrm{w}\big(\boldsymbol{f}'_u(\boldsymbol{x})\big)}{\mathrm{w}\big(\mathbf{S}^{\natural}(f, \boldsymbol{x}, \check{x})\big)} = 2.$$

Thus, slopes lead to narrower intervals than derivatives, where they are applicable.

[13] In fact, this particular a is the smallest such set satisfying Definition 1.12.

Theorem 1.10 is well known; its proof is not deep, and can be constructed from the following: $\mu \in \mathbf{S}(f, x, \check{x})$ implies $\mu = (f(xf(\check{x}))/(x - \check{x}) = f''(c_2)(x - \check{x})$ for some $x \in \boldsymbol{x}$ and $c_2 \in \boldsymbol{x}$, while, for the same x, $f'(x) = f''(c_1)(x - \check{x})$ for some $c_1 \in \boldsymbol{x}$. Thus, $\mu/f'(x) = \frac{1}{2}\frac{f''(c_2)}{f''(c_1)} \approx \frac{1}{2}$, so $\mathbf{S}(f, x, \check{x}) \approx 2f'_u(x)$, and $\mathrm{w}(c[\underline{x}, \overline{x}]) = c\mathrm{w}([\underline{x}, \overline{x}])$.

In general, only enclosures for the range of the derivative and enclosures for the slope range $\mathbf{S}^\sharp(F, \boldsymbol{X}, \check{\boldsymbol{X}})$, not the exact ranges, can be computed, and the relationship between derivatives and slopes for multivariate functions is more complicated. However, with automatic differentiation as in §1.4 below, computation of derivative and slope enclosures are equally easy, and the computed slope enclosures are generally tighter than the derivative enclosures. Some comments and examples appear in [141], while a reference especially relevant for polynomials appears in [6]. In fact, Theorem 1.10 is a good rule of thumb for predicting the relative behavior of interval derivative and slope-based computations in a variety of settings.

Multivariate Slopes

In Example 1.1 on page 26, a slope enclosure corresponding to a particular partial derivative of ϕ can be computed by simply treating the other variable as constant. For example, the second component of the slope matrix for ϕ over $\boldsymbol{X} = ([-.5, .5], [-.5, .5])^T$ can be computed as $\mathbf{S}(x^3/3, [-.5, .5], 0) \subset [0, .04167]$, rounded out to four digits. Similarly, with techniques of §1.4 below, the first component of the slope matrix can be computed as $[1, 1.04167]$, so $\boldsymbol{A} = \big([1, 1.25], [0, .25]\big)$ is a slope matrix for ϕ centered at $\check{X} = (0, 0)^T$ over $\boldsymbol{X}$, and a second-order interval extension of ϕ is

$$\begin{aligned} \boldsymbol{\phi}_s(\boldsymbol{X}) = 0 + \boldsymbol{A}(\boldsymbol{X} - \check{X}) &= ([1, 1.08334], [0, .08334]) \begin{pmatrix} [-.5, .5] \\ [-.5, .5] \end{pmatrix} \\ &\subset [-.5834, .5834]. \end{aligned} \tag{1.23}$$

Here, $[-.5834, .5834] \subset [-.75, .75]$, where $[-.75, .75]$ is the enclosure obtained with the interval derivative, while the actual range of ϕ, rounded out to four digits, is $\boldsymbol{\phi}^u(\boldsymbol{X}) \subset [-.5834, .5834]$, equal in this case to the enclosure given by slopes.

However, additional care must be taken when computing slopes of general multivariate functions. For example, if $\phi(X) = x_1^3 x_2^3$, with $\boldsymbol{X} = ([-.5, .5], [-.5, .5])^T$ and $\check{X} = (0, 0)^T$, one would be tempted to compute the first element of the slope enclosure by setting $x_2 \leftarrow \check{x}_2 = 0$, then using a univariate slope computation

for x_1, and to similarly compute the second element by setting $x_1 \leftarrow \check{x}_1 = 0$. However, this would lead to the clearly incorrect[14] slope matrix of $\boldsymbol{A} = (0, 0)$. Thus, *in multivariate problems, variables other than the active one in computing a component of a slope matrix must be set to their interval, and not point, values.* This is crystallized in

Algorithm 3 (Computation of a slope matrix)

INPUT: The function $\phi : \mathbb{R}^n \to \mathbb{R}$, the box $\boldsymbol{X}$ and the center $\check{\boldsymbol{X}}$, and a method of computing a slope bound $\mathbf{S}(f, x, \check{x})$ of a univariate function $f : \mathbb{R} \to \mathbb{R}$.

OUTPUT: A slope matrix $\boldsymbol{A} = \mathbf{S}(\phi, \boldsymbol{X}, \check{\boldsymbol{X}})$.

DO *for* $i = 1$ *to* n*:*

1. *Define* $f_i(x) = \phi(\widehat{\boldsymbol{X}})$, *where* $\widehat{x}_j = x_j$ *for* $1 \leq j \leq n$, $j \neq i$ *and* $\widehat{x}_i = x$, *where* ϕ *is an interval extension of* ϕ. *(Note that* f_i *is in general an interval, even if* x *is a point.)*
2. *Set* $\boldsymbol{a}_i$ *to the slope of the univariate function* f_i*:* $\boldsymbol{a}_i \leftarrow \mathbf{S}(f_i, x_i, \check{x}_i)$.

END DO

End Algorithm 3

1.3.2 Hansen's Slope Technique

In [73], and later in [77, §6.3–§6.4], Hansen explains a technique that allows use of point values for some of the variables, when computing derivative matrices for mean value type interval extensions. Hansen's slope technique is based on decomposing the change in $\phi : \mathbb{R}^n \to \mathbb{R}$ into changes in due to displacements along the coordinate directions, and on simple application of the mean value theorem. Different interval extensions are obtained depending on which displacements are considered first. For example, suppose $\phi : \mathbb{R}^2 \to \mathbb{R}$, and $\check{X} \in \boldsymbol{X} \subset \mathbb{IR}^2$. If displacement along the x_1 direction is considered first, cf. Fig. 1.4(a), then, for $(x_1, x_2) \in \boldsymbol{X}$,

$$\phi(x_1, x_2) = \phi(\check{x}_1, \check{x}_2) + \frac{\partial \phi}{\partial x_1}(c_1, \check{x}_2)(x_1 - \check{x}_1) + \frac{\partial \phi}{\partial x_2}(x_1, c_2)(x_2 - \check{x}_2), \quad (1.24)$$

[14] Use of this faulty slope matrix in an interval extension would lead to the conclusion that ϕ is constant on $\boldsymbol{X}$.

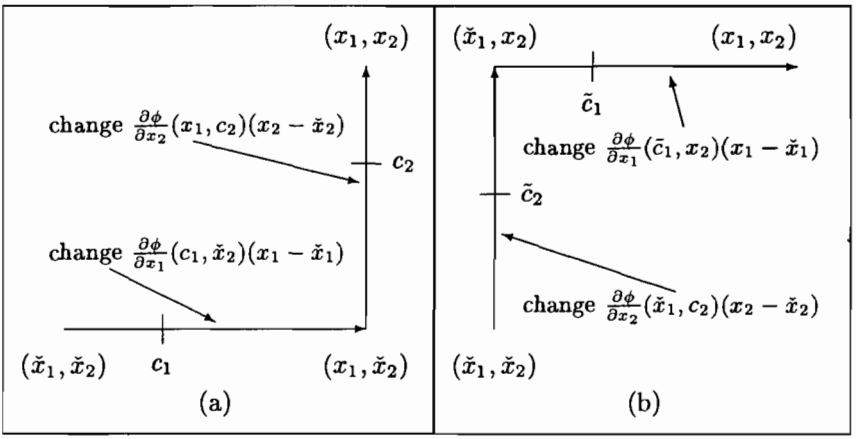


Figure 1.4 Two ways of computing the change in ϕ between $(\check{x}_1, \check{x}_2)$ and (x_1, x_2)

for some $c_1 \in \boldsymbol{x}_1$ and $c_2 \in \boldsymbol{x}_2$. Equation (1.24) in turn implies

$$\phi(x_1, x_2) \in \phi(\check{x}_1, \check{x}_2) + \frac{\partial \boldsymbol{\phi}}{\partial \boldsymbol{x}_1}(\boldsymbol{x}_1, \check{x}_2)(\boldsymbol{x}_1 - \check{x}_1) + \frac{\partial \boldsymbol{\phi}}{\partial \boldsymbol{x}_2}(\boldsymbol{x}_1, \boldsymbol{x}_2)(\boldsymbol{x}_2 - \check{x}_2), \quad (1.25)$$

where $\partial \boldsymbol{\phi}/\partial \boldsymbol{x}_1$ and $\partial \boldsymbol{\phi}/\partial \boldsymbol{x}_2$ represent any interval extensions of $\partial \phi/\partial x_1$ and $\partial \phi/\partial x_2$. The right member of Equation (1.25) is thus an interval extension of ϕ over $\boldsymbol{X}$ that can be proven to be second order. Similarly, if displacement along the x_2 direction is considered first, cf. Figure 1.4(b), then we obtain

$$\phi(x_1, x_2) \in \phi(\check{x}_1, \check{x}_2) + \frac{\partial \phi}{\partial x_1}(\boldsymbol{x}_1, \boldsymbol{x}_2)(\boldsymbol{x}_1 - \check{x}_1) + \frac{\partial \phi}{\partial x_2}(\check{x}_1, \boldsymbol{x}_2)(\boldsymbol{x}_2 - \check{x}_2), \quad (1.26)$$

where the right member of Equation (1.26) is a second-order interval extension of ϕ that is different from that in Equation (1.25).

Comparing equations (1.25) and (1.26) to Equation (1.16) (page 15) for the mean value extension, i.e. to

$$\begin{aligned} \boldsymbol{\phi}_2\big((\boldsymbol{x}_1, \boldsymbol{x}_2), (\check{x}_1, \check{x}_2)\big) = \phi(\check{x}_1, \check{x}_2) \quad &+ \quad \frac{\partial \boldsymbol{\phi}}{\partial \boldsymbol{x}_1}(\boldsymbol{x}_1, \boldsymbol{x}_2)(\boldsymbol{x}_1 - \check{x}_1) \\ &+ \quad \frac{\partial \boldsymbol{\phi}}{\partial \boldsymbol{x}_2}(\boldsymbol{x}_1, \boldsymbol{x}_2)(\boldsymbol{x}_2 - \check{x}_2), \quad (1.27) \end{aligned}$$

shows that equations (1.25) and (1.26) generally give sharper inclusions, since, in Hansen's extensions, some of the arguments to the partial derivatives are points instead of intervals.

For a concrete example, take

Example 1.2 *Let*

$$\phi(x_1, x_2) = x_1 x_2^2 + x_2,$$

$\boldsymbol{X} = ([-.5, .5], [-.5, .5])^T$, *and* $\check{X} = (0, 0)^T$. *Then the mean value extension corresponding to Equation (1.27) is*

$$\begin{aligned} \phi_2(\boldsymbol{X}) &= 0 + \left([-.5, .5]^2\right)[-.5, .5] + \left(2[-.5, .5][-.5, .5] + 1\right)[-.5, .5] \\ &= [-0.875, 0.875], \end{aligned}$$

the extension corresponding to Equation (1.25) is

$$\begin{aligned} \phi^H_{(1,2)}(\boldsymbol{X}, \check{X}) &= \left(0^2\right)[-.5, .5] + \left(2[-.5, .5][-.5, .5] + 1\right)[-.5, .5] \\ &= [-0.75, 0.75], \end{aligned}$$

and the extension corresponding to Equation (1.26) is

$$\begin{aligned} \phi^H_{(2,1)}(\boldsymbol{X}, \check{X}) &= \left([-.5, .5]^2\right)[-.5, .5] + \left(2(0)[-.5, .5] + 1\right)[-.5, .5] \\ &= [-0.75, 0.75]. \end{aligned}$$

The optimization procedures[15] *described in Chapter 5 give an actual range* $\phi^u(\boldsymbol{X}) = [-0.625, 0.625]$.

Although $\phi^H_{(1,2)}(\boldsymbol{X}) = \phi^H_{(2,1)}(\boldsymbol{X})$ in Example 1.2, this is in general not the case; see Exercise 1 on page 35.

Definition 1.13 *Suppose* $\phi : \mathbb{D} \subseteq \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{X} \subseteq \mathbb{D}$, $\check{X} \in \boldsymbol{X}$, *and* $\{i_j\}_{j=1}^n = \{1, 2, \ldots, n\}$. *Then* $\phi^H_{(i_1, i_2, \ldots i_n)}(\boldsymbol{X}, \check{X})$ *is defined to be Hansen's interval extension as above, with displacement in the* i_1 *direction first, in the* i_2 *direction second, etc.*

In the main context in optimization (interval Newton methods, §1.5.1 below), the intervals corresponding to partial derivative values are as important as the

[15] Also, it can be seen directly from the fact that $\partial\phi/\partial x_1 = x_2^2 \geq 0$ and $\partial\phi/\partial x_2 = 2x_1x_2 + 1 \geq 0$ that the minimum of ϕ occurs at $(-.5, -.5)$ and the maximum occurs at $(.5, .5)$.

actual extensions. In Example 1.2, the interval gradient is

$$\nabla\phi(\boldsymbol{X}) = \big([0,.25],[.5,1.5]\big)^T,$$

the vector corresponding to $\phi^H_{(1,2)}$ is

$$\nabla\phi^H_{(1,2)}(\boldsymbol{X},\check{X}) = \big(0,[.5,1.5]\big)^T,$$

and the vector corresponding to $\phi^H_{(2,1)}$ is

$$\nabla\phi^H_{(2,1)}(\boldsymbol{X},\check{X}) = \big([0,.25],1\big)^T.$$

This example shows that Hansen's slope technique gives better inclusions than the interval gradient, even though a particular ordering may not be optimal for a particular component.

In the optimization context, the Lipschitz property (Definition 1.10 on page 26) and the slope matrix property (Definition 1.12 on page 27) are important in the design of interval Newton methods to verify existence and uniqueness of critical points. The Hansen extensions do not correspond to Lipschitz sets, but clearly correspond to the definition of slope matrices. In fact, Hansen's technique can be combined with the multivariate slope computation procedure as a minor modification to Algorithm 3.

Definition 1.14 *A* Hansen slope *of a function* $\phi : \mathbb{R}^n \to \mathbb{R}$*, denoted by* $\mathbf{S}^{H,S}_{(i_1,i_2,\ldots i_n)}(\phi,\boldsymbol{X},\check{X})$*, is defined to be the output* $\boldsymbol{A}$ *of the following algorithm.*

Algorithm 4 (Computation of a Hansen slope matrix)

INPUT: The function $\phi : \mathbb{R}^n \to \mathbb{R}$, the box $\boldsymbol{X}$ and the center $\check{\boldsymbol{X}}$, and a method of computing a slope bound $\mathbf{S}(f,\boldsymbol{x},\check{x})$ of a univariate function $f : \mathbb{R} \to \mathbb{R}$.

OUTPUT: A Hansen slope matrix $\boldsymbol{A} = \mathbf{S}(\phi,\boldsymbol{X},\check{\boldsymbol{X}})$.

DO *for* $i = 1$ *to* n*:*

1. *As in Algorithm 3, define* $f_i(x) = \phi(\check{\boldsymbol{X}})$ *with* $\check{x}_i = x$*, where* ϕ *is an interval extension of* ϕ*. However, set* $\check{x}_j = \check{x}_j$ *for* $1 \le j \le n$*,* $j \ne i$*, and* $j \in \{i_k\}_{k=i+1}^{n}$*, and* $\check{x}_j = x_j$ *for* $j \in \{i_k\}_{k=1}^{i-1}$*, where* $\{i_k\}_{k=1}^{n} = \{k \mid 1 \le k \le n\}$ *is a permutation of the first* n *positive integers.*

2. Set $\boldsymbol{a}_i$ to the slope of the univariate function f_i: $\boldsymbol{a}_i \leftarrow \mathbf{S}(f_i, \boldsymbol{x}_i, \check{x}_i)$.

END DO

End Algorithm 4

In the absence of other information, i_k can be set to k for $1 \leq k \leq n$. The expansion can also be done last with respect to those variables x_i for which the scaled partial derivative widths

$$\frac{\partial\phi(\check{x}_1, \ldots, \check{x}_{i-1}, \boldsymbol{x}_i, \check{x}_{i+1}, \ldots, \check{x}_n)}{\partial x_j}(\boldsymbol{x}_j - \check{x}_j)$$

differ most from the corresponding "point approximation" widths

$$\frac{\partial\phi(\check{x}_1, \ldots, \check{x}_{i-1}, \check{x}_i, \check{x}_{i+1}, \ldots, \check{x}_n)}{\partial x_j}(\boldsymbol{x}_j - \check{x}_j),$$

although this is a heuristic. A less expensive heuristic would be to order the variables to expand in order of increasing

$$\frac{\partial\phi}{\partial x_i}(\boldsymbol{x}_i - \check{x}_i).$$

In any case, for sharper inclusions, results corresponding to several different orders can be computed and intersected.

For example, Algorithm 3, applied to compute $\boldsymbol{A}$ for the ϕ in Example 1.2, gives $f_1(x_1) = [0, .25]x_1 + [-.5, .5]$. With techniques of §1.4.3 below, a bound $\mathbf{S}(f_1, \boldsymbol{x}_1, \check{x}_1)$ is computed to be $[0, .25]$. Similarly, $f_2(x_2) = [-.5, .5]x_2^2 + x_2$, and a bound $\mathbf{S}(f_2, \boldsymbol{x}_2, \check{x}_2)$ is computed to be $[-.5, .5][-.5, .5] + 1 = [.75, 1.25]$. The corresponding computed slope matrix is:

$$\mathbf{S}(\phi, \boldsymbol{X}, \check{X}) = \big([0, .25], [.75, 1.25]\big)^T, \tag{1.28}$$

which is the same as the interval gradient $\boldsymbol{\nabla\phi}(\boldsymbol{X})$ (computed below Example 1.2 on page 33). Similarly, for this ϕ, the fact that relevant terms in ϕ are linear causes $\mathbf{S}^{H,S}_{(1,2)}(\phi, \boldsymbol{X}, \check{X}) = \boldsymbol{\nabla\phi}^H_{(1,2)}(\boldsymbol{X}, \check{X})$ and $\mathbf{S}^{H,S}_{(2,1)}(\phi, \boldsymbol{X}, \check{X}) = \boldsymbol{\nabla\phi}^H_{(2,1)}(\boldsymbol{X}, \check{X})$, where $\boldsymbol{\nabla\phi}^H_{(1,2)}(\boldsymbol{X}, \check{X})$ and $\boldsymbol{\nabla\phi}^H_{(2,1)}(\boldsymbol{X}, \check{X})$ are computed below $\boldsymbol{\nabla\phi}(\boldsymbol{X})$ on page 33.

On the other hand, in Example 1.1 on page 26, with $\phi(x_1, x_2) = x_1^3/3 + x_2^3/3 + x_1$,

$$\boldsymbol{\nabla\phi}(\boldsymbol{X}) = \big([1, 1.25], [0, .25]\big)^T,$$

whereas a computable slope enclosure, given in Equation (1.23) on page 29, is

$$\mathbf{S}(x_1^3/3 + x_2^3/3 + x_1, \boldsymbol{X}, \check{X}) = \left([1, 1.08334], [0, .08334]\right)^T.$$

For this example, the slope enclosures from Algorithm 3 and the Hansen-slope enclosures from Algorithm 4 are the same, but the multivariate slope enclosures are better than the gradient.

We emphasize:

Theorem 1.11 *The Hansen slope* $\mathbf{S}^{H,S}_{(i_1,i_2,\ldots i_n)}(\phi, \boldsymbol{X}, \check{X})$ *given by Algorithm 4 is a slope matrix in the sense of Definition 1.12.*

This theorem follows from careful consideration of the definitions.

Also:

Theorem 1.12 *Assume all interval computations and interval extensions are inclusion monotonic. Then* $\mathbf{S}^{H,S}_{(i_1,i_2,\ldots i_n)}(\phi, \boldsymbol{X}, \check{X}) \subseteq \mathbf{S}(\phi, \boldsymbol{X}, \check{X})$, *where* $\mathbf{S}(\phi, \boldsymbol{X}, \check{X})$ *is computed by Algorithm 3.*

Because of Theorems 1.11 and 1.12, in almost all situations Algorithm 4 should be used instead of Algorithm 3, unless, for some reason, Algorithm 3 can be implemented more efficiently.

1.3.3 Exercises

1. If $\phi(\boldsymbol{X}) = x_1^2 x_2 + x_1 x_2^3$, $\boldsymbol{X} = ([-.5, .5], [-.5, .5])^T$ and $\check{X} = (0, 0)^T$, then compute and compare $\phi^H_{(1,2)}(\boldsymbol{X}, \check{X})$ and $\phi^H_{(2,1)}(\boldsymbol{X}, \check{X})$. Also compare to the mean value extension $\phi_2(\boldsymbol{X}, \check{X})$ and to the natural interval extension. Is it easy to obtain the exact range $\phi^u(\boldsymbol{X})$ for this ϕ?

   Hint: $\phi^u(\boldsymbol{X}) \subset [-.068042, .068042]$. *Techniques of calculus may be used in this case.*

2. Prove Theorem 1.11.

3. Prove Theorem 1.12.

1.4 AUTOMATIC DIFFERENTIATION AND CODE LISTS

Description of a global optimization problem requires specification of an objective function and constraints. However, algorithms for verified global optimization may require both floating point and interval values (i.e. interval enclosures or slope enclosures) of the objective function, gradient and Hessian matrices, as well as, possibly, interval values of the intermediate quantities obtained in computing a natural interval extension[16]. With floating point and interval values of the constraints and their gradients, this makes at least ten different routines[17] that would be required for each problem, an unacceptable burden if many problems are to be solved.

The earliest approach, in floating point codes for approximations to local optimizers, avoided a separate programming effort for Jacobian or Hessian matrices. They used numerical differentiation, i.e. computation of approximations with forward or central differences. An alternate possibility is to pre-process with symbolic manipulation packages such as MACSYMA, Maple, Reduce or Mathematica. A third approach, followed in several packages such as LANCELOT [35], provides ad hoc programs to interpret a standard input format, such as MPS format [174], or SIF format [35], or a proposed format of Neumaier [176], and actually writes target language (such as Fortran) subroutines for each required computation.

Each of these approaches have failings. Numerical differentiation can be overwhelmed by both truncation and roundoff error, can result in n times the amount of effort to evaluate the gradient of a function of n variables[18], and is unsuitable for interval computations. Symbolic differentiation requires substantial separate machinery, and can also result in expressions for the derivatives that are many times larger than optimal. Special input formats and special packages must be learned by the user, and are not universal. An alternate technique is *automatic differentiation.*

[16] See Chapter 7 below.

[17] The routines must include: (1) floating point objective function, (2) floating point gradient, (3) interval objective function, (4) interval gradient, (5) floating point derivative matrix, (6) interval derivative matrix, (7) floating point constraints, (8) interval constraints, (9) floating point constraint gradients, (10) interval constraint gradients. Several others may also be used, depending on the algorithm.

[18] depending on system structure and how it is utilized: see [34].

1.4.1 The Forward Mode

The forward mode of differentiation is analogous to interval arithmetic in the sense that an arithmetic is defined on an extended set of objects, intervals in the case of interval arithmetic and function / derivative pairs in the case of automatic differentiation. That is, an object for automatic differentiation is an ordered pair of the form $\langle u, u' \rangle$, where the elements u and u' are real numbers or intervals[19]. The rules for *differentiation arithmetic* are based on elementary rules of differentiation. For the four elementary operations, these rules are:

$$\begin{aligned} \langle u, u' \rangle + \langle v, v' \rangle &= \langle u + v, u' + v' \rangle, \\ \langle u, u' \rangle - \langle v, v' \rangle &= \langle u - v, u' - v' \rangle, \\ \langle u, u' \rangle \cdot \langle v, v' \rangle &= \langle uv, uv' + vu' \rangle \\ \langle u, u' \rangle / \langle v, v' \rangle &= \langle u/v, (u'v - uv')/v^2 \rangle. \end{aligned} \tag{1.29}$$

Similarly, rules for standard functions are defined according to formulas of differential calculus. For example:

$$\begin{aligned} \sin(\langle u, u' \rangle) &= \langle \sin u, u' \cos u \rangle, \\ \langle u, u' \rangle^2 &= \langle u^2, 2uu' \rangle, \\ \langle u, u' \rangle^3 &= \langle u^3, 3u^2 u' \rangle, \text{ etc.} \end{aligned}$$

In arithmetic expressions, constants c are identified with pairs $\langle c, 0 \rangle$ while independent variables x are identified with pairs $\langle x, 1 \rangle$.

Example 1.3 *Suppose*

$$f(x) = x^4 + x^3 + x,$$

and $f(x)$ and $f'(x)$ are to be evaluated with $x = [.99, 1.01]$. Evaluation of this function can then be decomposed into the following operations

$$\begin{aligned} x_1 &= x \\ x_2 &\leftarrow x_1^4 \\ x_3 &\leftarrow x_1^3 \\ x_4 &\leftarrow x_2 + x_3 \\ x_5 &\leftarrow x_4 + x_1 \\ x_5 & \quad \textit{is dependent,} \end{aligned} \tag{1.30}$$

and the arithmetic, rounded out to four digits, can proceed as follows:

$$\langle x_2, x_2' \rangle \leftarrow \langle x, 1 \rangle^4 = \langle [.99, 1.01]^4, 4[.99, 1.01]^3 \rangle$$

[19] or floating point numbers and floating point intervals

$$
\begin{aligned}
& \subset \langle [.9606, 1.041], [3.881, 4.122] \rangle, \\
\langle x_3, x_3' \rangle & \leftarrow \langle x, 1 \rangle^3 = \langle [.99, 1.01]^3, 3[.99, 1.01]^2 \rangle \\
& \subset \langle [.9702, 1.031], [2.940, 3.061] \rangle \\
\langle x_4, x_4' \rangle & \leftarrow \langle x_2, x_2' \rangle + \langle x_3, x_3' \rangle \\
& \subset \langle [1.930, 2.072], [6.821, 7.183] \rangle \\
\langle x_5, x_5' \rangle & \leftarrow \langle x_4, x_4' \rangle + \langle x, 1 \rangle \\
& \subset \langle [2.920, 3.082], [7.821, 8.183] \rangle.
\end{aligned}
$$

Thus, an interval enclosure for $\{f(x) \mid x \in [.99, 1.01]\}$ *is* $[2.920, 3.082]$, *while an interval enclosure for* $\{f'(x) \mid x \in [.99, 1.01]\}$ *is* $[7.821, 8.183]$.

Although automatic differentiation is generally viewed as propagating values rather than expressions, a similar technique can be used to obtain derivatives in a symbolic form; see page 91 below. In either case, automatic differentiation is different from symbolic differentiation. There is no *expression swell* (that is, the size of the symbolic representations does not become excessive) in automatic differentiation.

As seen in Example 1.3, in the *forward mode* of automatic differentiation, the intermediate quantities in the expressions for the values and derivatives are computed in the order in which the expression is evaluated.

The earliest explanation of the forward mode is [239], while an early but thorough explanation is [187].

1.4.2 The Reverse Mode

An alternate mode of computation is the *reverse mode*. In this scheme, each of the intermediate quantities computed when evaluating the expression for the derivative term is assigned a variable. In Example 1.3, there are four operations, so four intermediate variables are introduced: x_2 through x_5; x is denoted x_1, and x_5 corresponds to the dependent variable $f(x)$. The system of equations for Example 1.3 is thus

$$
\begin{aligned}
x_1 - c &= 0 \\
x_1^4 - x_2 &= 0 \\
x_1^3 - x_3 &= 0 \\
(x_2 + x_3) - x_4 &= 0 \\
(x_4 + x_1) - x_5 &= 0.
\end{aligned}
\tag{1.31}
$$

Throughout, this system of equations will be called the *expanded system.*

If each term in System (1.31) is now differentiated with respect to the variables occurring in it[20], repeated application of the chain rule leads to the following system of equations for the derivatives of the intermediate quantities with respect to x.

$$\begin{pmatrix} 1 & & & & \\ 4x_1^3 & -1 & & & \\ 3x_1^2 & & -1 & & \\ & 1 & 1 & -1 & \\ 1 & & & 1 & -1 \end{pmatrix} \begin{pmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \\ x_5' \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{1.32}$$

The forward mode corresponds to setting $x_1' = 1$ and applying forward substitution to this system. The reverse mode corresponds to using the fifth equation to eliminate x_4' from the fourth equation, using the resulting fourth equation to eliminate x_3' from the third equation, then using the resulting third equation to eliminate x_2' from the second equation. Finally, the first equation can be used to eliminate x_1' from the second equation. The first equation may then be solved for $x_5 = f'(x)$. Note that, in general, interval enclosures for the intermediate quantities (x_i, $2 \leq i \leq 5$, in this example) must be computed first. This may be done e.g. by forward substitution on System (1.31) or by computing second order extensions for each equation in System (1.31). For example, if $x = [.99, 1.01]$ as in Example 1.3, then, rounded out to four digits,

[20] For example, in the second equation, the first expression is differentiated with respect to x_1 and the second expression is differentiated with respect to x_2.

the augmented matrix for (1.32) becomes

$$
\left(\begin{array}{ccccc|c}
1 & & & & & 1 \\
[3.881,4.122] & -1 & & & & \\
[2.940,3.061] & & -1 & & & \\
 & 1 & 1 & -1 & & \\
1 & & & 1 & -1 &
\end{array}\right)
$$

$$
\sim \left(\begin{array}{ccccc|c}
1 & & & & & 1 \\
[3.881,4.122] & -1 & & & & \\
[2.940,3.061] & & -1 & & & \\
1 & 1 & 1 & 0 & -1 & \\
1 & & & 1 & -1 &
\end{array}\right)
$$

$$
\sim \left(\begin{array}{ccccc|c}
1 & & & & & 1 \\
[3.881,4.122] & -1 & & & & \\
[3.940,4.061] & 1 & 0 & & -1 & \\
1 & 1 & 1 & 0 & -1 & \\
1 & & & 1 & -1 &
\end{array}\right)
$$

$$
\sim \left(\begin{array}{ccccc|c}
1 & & & & & 1 \\
[7.822,8.183] & 0 & & & -1 & \\
[3.940,4.061] & 1 & 0 & & -1 & \\
1 & 1 & 1 & 0 & -1 & \\
1 & & & 1 & -1 &
\end{array}\right)
$$

$$
\sim \left(\begin{array}{ccccc|c}
1 & & & & & 1 \\
0 & 0 & & & -1 & [-8.183,-7.822] \\
[3.940,4.061] & 1 & 0 & & -1 & \\
1 & 1 & 1 & 0 & -1 & \\
1 & & & 1 & -1 &
\end{array}\right)
$$

Thus, $f'(x) \subseteq [-8.183,-7.822]/(-1) = [7.822,8.183]$ for $x \in \boldsymbol{x} = [.99,1.01]$. In general, the reverse mode does not give identical intervals to the forward mode. Observations indicate that the intervals obtained from the reverse mode are often narrower. Other elimination orders, different from both the forward and reverse modes and leading to different interval inclusions, are also possible [66].

The reverse mode works similarly for multivariate functions and partial differentiation: There is an additional equation and column for each independent variable. All but one of these rows and columns is ignored for a particular partial derivative. Because of the extreme sparsity[21] of System (1.32), it is not hard to show that the reverse mode can compute all partial derivatives in a time that is proportional[22] to the amount of time required to compute the original function value $\phi(x_1,\ldots,x_n)$, independently of n.

More generally, interval enclosures for the solution to the System (1.32) may be computed with optimal preconditioning, as in §3 below.

[21] If all operations are unary or binary, each row of the System (1.32) has at most three non-zero entries.

[22] Griewank has observed the constant of proportionality to be 5.

Early works dealing with the reverse mode are [225] and [92] (see also [93]), while Griewank [63, 64] has studied the process more recently. Shiriaev [223] has included vector operations in the expanded system, thus reducing the number of intermediate results and increasing the efficiency considerably for many problems.

1.4.3 Slope Arithmetic

Any interval Lipschitz matrix, such as $\boldsymbol{F}'(\boldsymbol{X})$ of Definition 1.11, is also a slope bound matrix $\mathbf{S}(F, \boldsymbol{X}, \check{X})$ for any $\check{X} \in \boldsymbol{X}$. However, tighter bounds can be computed automatically for particular $\check{X}$ or for particular $\check{\boldsymbol{X}} \subset \boldsymbol{X}$. This is done with slope arithmetic that is similar to the differentiation arithmetic of §1.4. Formulas analogous to those of §1.4 will be presented here; slopes of multivariate functions can be computed from these and from Algorithm 3.

As with differentiation arithmetic, *slope arithmetic* is based on defining operations and standard functions on ordered triplets of the form $\ll \check{u}, u, u^{(s)} \gg$. Addition and subtraction are defined the same way as with differentiation arithmetic, but multiplication and division differ somewhat: the rules[23] corresponding to formulas (1.29) are

$$\begin{aligned}
\ll \check{u}, u, u^{(s)} \gg + \ll \check{v}, v, v^{(s)} \gg &= \ll \check{u} + \check{v}, u + v, u^{(s)} + v^{(s)} \gg, \\
\ll \check{u}, u, u^{(s)} \gg - \ll \check{v}, v, v^{(s)} \gg &= \ll \check{u} - \check{v}, u - v, u^{(s)} - v^{(s)} \gg, \\
\ll \check{u}, u, u^{(s)} \gg \times \ll \check{v}, v, v^{(s)} \gg &= \ll \check{u}\check{v}, uv, uv^{(s)} + u^{(s)}\check{v} \gg, \\
\ll \check{u}, u, u^{(s)} \gg \div \ll \check{v}, v, v^{(s)} \gg &= \ll \check{u}/\check{v}, u/v, \frac{u^{(s)} - \check{u}v^{(s)}/\check{v}}{v} \gg.
\end{aligned} \tag{1.33}$$

As with differentiation arithmetic, rules could be defined for the standard functions according to the rules of differentiation arithmetic. However, the resulting slope bounds are not the sharpest possible. For standard functions $\omega(x)$ for which $\omega''(x) \neq 0$ for $x \in \boldsymbol{x}$, a difference formula computes the *exact* slope range $\mathbf{S}^{\natural}(\omega, \boldsymbol{x}, \check{x})$ to within roundout error:

Theorem 1.13 (A slight generalization of a theorem of Rump in [208, 211]) *Suppose* $\omega : \mathbb{R} \to \mathbb{R}$, $\boldsymbol{x} \in \mathbb{IR}$, $\check{x} \in \mathbb{IR}$, *and* ω *is either convex or concave (i.e.* $\omega''(x) \neq 0$ *if* ω'' *is continuous) for all* x *in the convex hull* $\boldsymbol{x} \cup \check{x}$. *Then*

[23] These rules were presented in [141], with additional insight in [208].

$\mathbf{S}^{(d)}(\omega, x, \check{x})$ *defined by*

$$\mathbf{S}^{(d)}(\omega, x, \check{x}) = h(\underline{x}) \cup h(\overline{x}) \quad \textit{with} \quad h(x) = \begin{cases} \frac{\omega(x)-\omega(\check{x})}{x-\check{x}} & \textit{for } x \notin \check{x} \\ \omega'(\check{x}) & \textit{otherwise} \end{cases} \tag{1.34}$$

is a slope set for ω *over* x *and centered on* $\check{x}$. *If* $\check{x}$ *is a point, then* $\mathbf{S}^{(d)}(\omega, x, \check{x})$ *is the smallest possible such slope set.*

It was proven in [208] that $\mathbf{S}^{(d)}(\omega, x, \check{x})$ is a slope set if $\omega'' \neq 0$, and it was pointed out in [211] that $\mathbf{S}^{(d)}(\omega, x, \check{x})$ is a slope set when ω is convex over $x \cup \check{x}$. The fact that this slope set is sharp for points $\check{x}$ follows from the fact that any slope set must contain $h(\underline{x})$ and $h(\overline{x})$. The fact that $\mathbf{S}^{(d)}(\omega, x, \check{x})$ is a slope set when $\check{x}$ is non-degenerate interval follows from elementary properties of interval arithmetic, assuming $\omega(\check{x})$ contains the range of ω over $\check{x}$.

In actual implementations, h is not defined exactly as in Equation (1.34), since roundout error becomes severe when x is close to $\check{x}$, even though $x \notin \check{x}$. In such implementations, $h(x)$ can be defined to be $\omega'(\check{x})$ whenever

$$<\check{x} - x> < \sqrt{\epsilon_m}, \tag{1.35}$$

where ϵ_m is the machine epsilon.

Theorem 1.13 can be applied directly when $\omega \in \{\exp, \log, \sqrt{}\}$. It can also be applied directly to $\omega(x) = x^n$ with n even, since such ω are limiting cases of ω satisfying the hypotheses. It can be applied to $\omega(x) = x^n$ with n odd by considering the algebraic signs of the endpoints of x and $\check{x}$, and it can be applied to functions such as $\omega(x, y) = x^y$, since they can be computed by composing log and exp. Theorem 1.13 can be applied when ω is an inverse trigonometric function by checking for the positions of the inflection points relative to x and $\check{x}$.

It is somewhat more difficult to apply Theorem 1.13 when $\omega \in \{\sin, \cos\}$. For these functions, the interval derivative or other techniques can be used.

For example, a slope bound for $f(x) = x^4 + x^3 + x$, $x = [.99, 1.01]$, $\check{x} = \check{x} = 1$ can be evaluated as in Example 1.3 on page 37:

$$\begin{aligned}
\ll \check{x}_2, x_2, x_2^{(s)} \gg &\leftarrow \ll \check{x}, x, 1 \gg^4 = \ll 1, [.99, 1.01]^4, \mathbf{S}^\sharp(x^4, [.99, 1.01], 1) \gg \\
&\subset \ll 1, [.9606, 1.041], [3.940, 4.061] \gg, \\
\ll \check{x}_3, x_3, x_3^{(s)} \gg &\leftarrow \ll \check{x}, x, 1 \gg^3 = \ll 1, [.99, 1.01]^3, \mathbf{S}^\sharp(x^3, [.99, 1.01], 1) \gg \\
&\subset \ll 1, [.9702, 1.031], [2.970, 3.031] \gg, \\
\ll \check{x}_4, x_4, x_4^{(s)} \gg &\leftarrow \ll \check{x}_2, x_2, x_2^{(s)} \gg + \ll \check{x}_3, x_3, x_3^{(s)} \gg \\
&\subset \ll 2, [1.930, 2.072], [6.910, 7.092] \gg, \\
\ll \check{x}_5, x_5, x_5^{(s)} \gg &\leftarrow \ll \check{x}_4, x_4, x_4^{(s)} \gg + \ll \check{x}, x, 1 \gg \\
&\subset \ll 3, [2.920, 3.082], [7.910, 8.092] \gg.
\end{aligned}$$

Thus, an interval enclosure for $\{f(x) \mid x \in [.99, 1.01]\}$ is $[2.920, 3.082]$, the value $f(1)$ is 3, and an interval enclosure for the slope is:

$$\{\mathbf{S}^\sharp(f, x, \check{x}) \mid x \in [.99, 1.01], \check{x} = 1\} \subset [7.910, 8.092].$$

Note that $[7.910, 8.092] \subset [7.821, 8.183]$, where $[7.821, 8.183]$ is the interval bound on the derivative obtained in Example 1.3. In fact, $\mathrm{w}([7.910, 8.092]) = 0.182 \approx \frac{1}{2}\mathrm{w}([7.821, 8.183]) = 0.362$, as theory of $\mathbf{S}^\sharp$ would predict.

A final note: Neumaier has suggested that slope arithmetic can be done with matrices, rather than with numbers. This would obviate the need for Hansen's slope technique, and also be more efficient that computing derivative matrices.

1.4.4 Operator Overloading and Code List Generation

In the software described in this work, both the forward mode and reverse mode will be used, as well as a *symbolic* version of each. To do so, methods explained early in [187] (and earlier in [131]) will be used to first create an internal representation, termed a *code list*, of the function from a user-created program. Generic routines are then used, with this code list as input, to compute interval and floating point function and derivative values, and to symbolically differentiate a code list. In this work[24], the code list will be created with *operator overloading*.

[24] Alternatives are to use operator overloading directly in automatic differentiation, without creating a code list, to implement a special programming language with a derivative data type, or to create a special computer program that generates a code list. Each has advantages and disadvantages.

Considered one of the four capabilities defining an object-oriented programming language ([184], etc.), operator overloading consists of extension of the definitions of arithmetic and logical operators ("+", "−", "×", "÷", "<", ">", standard functions such as "sin", etc.) to user-defined data types. In this technique, the data type is first defined (such as an interval as an array of two double precision words). Then subroutines are written to implement the arithmetic and logical operations on this data type. For example, if an interval were defined in FORTRAN-77 as a double precision array with two elements, then a subroutine for addition of two intervals could be of the general form[25]

```
SUBROUTINE ADD(A, B, RESULT)
   DOUBLE PRECISION A(2), B(2), RESULT(2)
   RESULT(1) = A(1) + B(1)
   RESULT(2) = A(2) + B(2)
   CALL RNDOUT(RESULT)
   RETURN
END
```

The final step in the operator overloading process is to connect the subroutine names (such as ADD) to the operator symbols (such as "+"). This is done e.g. in Fortran 90 with a "module."

An early package allowing this technique was the Augment precompiler [42], a FORTRAN-66 preprocessor with which an interval data type [247] was provided. Current common modern programming languages with operator overloading are C++, Fortran 90 and Ada. Also, the SC languages, e.g. [70, §2.2], besides containing intrinsic[26] interval data types, provide language constructs for operator overloading.

The subroutines for particular operations need not do actual arithmetic operations, but can write lines in a table. This is how code lists are generated. Here, we sketch briefly how this is implemented, introducing only enough detail to describe Example 1.3 on page 37. More detail appears in §2.2.2 on page 83.

A partial set of operation codes used in the system of [117] appears in Table 1.1.

Operation	Code
$x_p \leftarrow x_q^r$	19
$x_p \leftarrow x_q + x_r$	20

Table 1.1 Some example operations and corresponding numerical codes

[25] This is a simplification of the corresponding INTLIB [123] routine.
[26] i.e. already defined in the language

New CDLVAR ("code list variable") and CDLLHS ("code list left-hand-side") data types, essentially integer storage addresses, can then be defined[27] in Fortran 90 as follows:

```
TYPE CDLVAR
   INTEGER LOCATION
END TYPE CDLVAR
TYPE CDLLHS
   INTEGER LOCATION
END TYPE CDLLHS
```

Arrays OPA, PA, QA, and RA containing the operation code as in Table 1.1 and the indices of the variables p, q, and r as in Table 1.1, respectively, are defined. Simplified[28] Fortran 90 subroutines for code list generation, corresponding to exponentiation and addition, can then be written:

```
!  The following is for overloading exponentiation A**N.
  FUNCTION CLPOWN(A, N) RESULT(R)
      TYPE(CDLVAR), INTENT(IN):: A
      INTEGER, INTENT(IN):: N
      TYPE(CDLVAR):: R
      NROW = NROW+1
      NVAR = NVAR+1
      OPA(NROW) = 19
      PA(NROW)  = NVAR
      QA(NROW)  = A%LOCATION
      RA(NROW)  = N
      R%LOCATION=NVAR
  END FUNCTION CLPOWN
```

[27] Here, concepts are illustrated with Fortran 90 constructs; see e.g. [22] for an introduction to Fortran 90.

[28] i.e. somewhat simplified versions of actual routines of INTLIB_90 described in §2.2.2 on page 83 below

```
!  The following is for overloading addition A+B.
   FUNCTION CLADD(A, B) RESULT(R)
       TYPE(CDLVAR), INTENT(IN)::  A, B
       TYPE(CDLVAR)::  R
       NROW = NROW+1
       NVAR = NVAR+1
       OPA(NROW) = 20
       PA(NROW)  = NVAR
       QA(NROW)  = A%LOCATION
       RA(NROW)  = B%LOCATION
       R%LOCATION=NVAR
   END FUNCTION CLADD
```

Assignment "=" can also be overloaded to set a dependent variable (a function value, of type CDLLHS) to an expression, as in the following Fortran 90 code.

```
!  The following routine overloads f(i) = <expression>
   SUBROUTINE CLLHS(F,EXPRESSION)
       TYPE(CDLLHS), INTENT (OUT)::  F
       TYPE(CDLVAR), INTENT(IN)::  EXPRESSION
       NROW = NROW+1
       NEQ  = NEQ + 1
       OPA(NROW) = 18
       PA(NROW)  = EXPRESSION%LOCATION
       QA(NROW)  = 0
       RA(NROW)  = 0
       LHSLOC(NEQ) = NROW
       F%LOCATION = NEQ
   END SUBROUTINE CLLHS
```

The routines CLPOWN, CLADD, and CLLHS can then be connected to the two operations in Table 1.1 within a Fortran 90 module OVERLOAD, containing the following statements:

```
INTERFACE OPERATOR(**)
   MODULE PROCEDURE CLPOW1
END INTERFACE
INTERFACE OPERATOR(+)
   MODULE PROCEDURE CLADD
END INTERFACE
INTERFACE ASSIGNMENT(=)
   MODULE PROCEDURE CLLHS
END INTERFACE
```

Suppose that the module OVERLOAD also contains a routine

INITIALIZE_CODELIST

to set NROW and NVAR to zero and to set various array dimensions such as the number of independent variables; and a routine FINISH_CODELIST to write the arrays[29] OPA, PA, QA, and RA to a table. Then the following Fortran 90 program would make sense.

```
PROGRAM EXAMPLE_OF_CODELIST_GENERATION
  USE OVERLOAD
      TYPE(CDLVAR), DIMENSION(1) ::  X
      TYPE(CDLLHS) ::  PHI
      CALL INITIALIZE_CODELIST(X)
      PHI =  X(1)**4 + X(1)**3 + X(1)
      CALL FINISH_CODELIST
END PROGRAM EXAMPLE_OF_CODELIST_GENERATION
```
(1.36)

Execution of this program would produce the code list in Table 1.2. Table 1.2 is a numerical description of precisely the sequence of operations given in System (1.30) in Example 1.3 on page 37.

1.4.5 Generic Functions

Once generated, a code list such as that in Table 1.2 (next page) contains, in principle, all information necessary to evaluate the corresponding function and

[29] declared in overload and thus available to any routine that uses OVERLOAD

OP	p	q	r
19	2	1	4
19	3	1	3
20	4	2	3
20	5	4	1
18	5	0	0

Table 1.2 Tabular representation of the output to the program (1.36)

its derivatives, with any data type. In fact, a single *interpreter* routine may be supplied to compute e.g. an interval enclosure for the range of *any* function representable as a code list. The principle is simple, and can be implemented by an if-then-else cascade or by a case statement. For example, suppose that `NI`, a variable available in module `OVERLOAD`, denotes the number of intermediate quantities computed during evaluation of the function. (`NI` = 4 in the code list in Table 1.2.) Then the subroutine in Figure 1.5 will return the floating-point value at X of a function represented by the code list with arrays `OPA(I)`, `PA(I)`, `QA(I)` and `RA(I)`, `I` = 1,...`NI`, provided the function can be evaluated with only additions and integer powers. Observe that such programs are easily extensible.

1.4.6 Exercises

1. Apply forward substitution (or use the results displayed in Example 1.3) to compute x_j, $j = 2, 3, 4, 5$ from the expanded System (1.31). Then use the reverse mode for the System (1.32) to compute an interval enclosure for the range of f' over $x = [.99, 1.01]$. Compare the result to the enclosure computed in the forward mode in Example 1.3.

2. Apply both the forward mode and the reverse mode of automatic differentiation to
$$\phi(x) = \left[\sin\left(\pi x^3\right)\right]^2$$
at $x = 1$.

3. Apply the reverse mode of automatic differentiation to compute $\partial\phi/\partial x_1(1,2)$ and $\partial\phi/\partial x_2(1,2)$, where $\phi(x_1, x_2) = x_1^2 + x_2 + x_1 x_2$. Observe that both partial derivatives may be obtained with only one pass through the initial part of the elimination process.

```
SUBROUTINE POINT_EVALUATION(X,FVAL)
   USE CODELIST_VARIABLES
   DOUBLE PRECISION, INTENT(IN) :: X
   DOUBLE PRECISION, INTENT(OUT):: FVAL
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE:: XX
      INTEGER J
   ALLOCATE(XX(NI+1))
      XX(1) = X
      DO J = 1, NI
         SELECT CASE(OPA(J))
            CASE(19)
               XX(PA(J)) = XX(QA(J))**RA(J)
            CASE(20)
               XX(PA(J)) = XX(QA(J)) + XX(RA(J))
            CASE DEFAULT
               WRITE(6,*)  'UNIMPLEMENTED OPERATION'
               STOP
         END SELECT
      END DO
      FVAL = XX(PA(NI+1))
   DEALLOCATE(XX)
END SUBROUTINE POINT_EVALUATION
```

Figure 1.5 Sample program to compute a floating point function value from a code list

4. Use the chain rule to show that the solution to systems of the form of System (1.32) gives $x_5 = f'(x)$ (or, more generally, $x_N = \partial\phi(x_1, \ldots, x_n)/\partial x_k$).

   Hint: You may want to use induction on the number of operations required to compute the function.

5. Assuming each operation is either a unary or binary operation (where standard function evaluations are counted as an operation), count the number of additions, subtractions, multiplications, and divisions to perform the reverse mode on a general system set up as System (1.32). Let n be the number of variables and N the number of operations to evaluate $\phi(x_1, x_2, \ldots, x_n)$. Write the operation count in terms of N and n. Does this differ from the common perception of the amount of work required to compute gradients?

6. If $\phi(x) = x^5 + x^4 + x^3 + x$, then write a code list for ϕ in the form of Table 1.2, using the operation codes of Table 1.1. Are such code lists unique? If not, then also write down an alternative code list.

   Note: Different code lists are obtained by doing the operations in a different order. However, functions such as $\tilde{\phi}(x) = x(1+x^2(1+x(1+x)))$, although equivalent in real arithmetic, are considered to be different functions. You may interpret the problem as asking if there are two different code lists that both obey the restrictions on the order of operations defined in the Fortran 90 standard.

7. For $f(x) = x^4 + x^3 + x$ with code list as in Table 1.2 and with entry value `X` $= 2$, manually trace execution of the subroutine `POINT_EVALUATION` in Figure 1.5. (That is, make a list of each statement passed in the program, with values of the variables after they are assigned.)

   The value `FVAL` *upon return should be* $f(2)$.

1.5 INTERVAL NEWTON METHODS AND INTERVAL FIXED POINT THEORY

The books [77], [175] and [191] are good references for interval Newton methods and interval fixed point theory. The reference [77] emphasizes concepts and practical aspects of implementation. An especially good reference for theory and the computational existence and uniqueness tests in §1.5.2 is [175], supplemented with the more recent results on uniqueness with slope matrices that appear in [208]. The reference [7] gives a good survey of interval Newton

methods based on interval Gaussian elimination, and also surveys the Krawczyk method. One relatively early use of classical fixed point theory in the analysis of interval methods is in [205].

In this section, univariate interval Newton methods will first be discussed. After that, multivariate interval Newton methods, based upon methods of bounding the solution sets to interval linear systems in §1.2.2 will be introduced. Next, computational existence and uniqueness theory, based on classical fixed-point theorems, will be presented.

1.5.1 Interval Newton Methods

Except for the choice of interval extensions or slopes, there is essentially a single univariate interval Newton method. In contrast, multivariate interval Newton methods vary according to solution method for the linear system, preconditioner, etc. We will discuss the simpler case first.

Univariate Interval Newton Methods

Suppose $f : \boldsymbol{x} = [\underline{x}, \overline{x}] \to \mathbb{R}$ has a continuous first derivative on $\boldsymbol{x}$, suppose that there exists $x^* \in \boldsymbol{x}$ such that $f(x^*) = 0$, and suppose that $\check{x} \in \boldsymbol{x}$. Then, since the mean value theorem implies

$$\begin{aligned} 0 &= f(x^*) = f(\check{x}) + f'(\xi)(x^* - \check{x}), \\ \text{we have } x^* &= \check{x} - f(\check{x})/f'(\xi) \end{aligned}$$

for some $\xi \in \boldsymbol{x}$. If $\boldsymbol{f}'(\boldsymbol{x})$ is any interval extension of the derivative of f over $\boldsymbol{x}$, then

$$x^* \in \check{x} - f(\check{x})/\boldsymbol{f}'(\boldsymbol{x}) \quad \text{for any } \check{x} \in \boldsymbol{x}.$$

Similarly, if $\mathbf{S}(f, \boldsymbol{x}, \check{x})$ is a slope set for f centered at $\check{x}$ (as in Definition 1.12 on page 27), then, also

$$x^* \in \check{x} - f(\check{x})/\mathbf{S}(f, \boldsymbol{x}, \check{x}) \quad \text{for any } \check{x} \in \boldsymbol{x}.$$

This leads to

Definition 1.15 (Univariate interval Newton methods) *The operators*

$$\boldsymbol{N}(f; \boldsymbol{x}, \check{x}) = \check{x} - f(\check{x})/\boldsymbol{f}'(\boldsymbol{x}) \tag{1.37}$$

or

$$N(f;x,\check{x}) = \check{x} - f(\check{x})/S(f,x,\check{x}) \tag{1.38}$$

are[30] *termed the* univariate interval Newton methods.

Recapitulating, if $x^* \in x$ has $f(x^*) = 0$, then $x^* \in N(f;x,\check{x})$. This fact underlies the use of interval Newton iteration to sharpen bounds on solutions and global optimizers. Another important fact making interval Newton methods efficient for global optimization algorithms is local quadratic convergence. An example of the type of theorem that can be proven[31] is

Theorem 1.14 *Suppose*

(a) $\check{x}^{(k)} = (\underline{x}^{(k)} + \overline{x}^{(k)})/2$ *for all* $x^{(k)}$,

(b) f *has a continuous derivative on an initial interval* $x^{(0)}$,

(c) $|f(x)| \leq M_f$ *for* $x \in x^{(0)}$,

(d) $N(f;x^{(0)},\check{x}^{(0)})$ *is defined with Equation (1.37),*

(e) f' *is an inclusion monotonic interval extension of at least first order in the sense of Definition 1.4 on page 14,*

(f) $\min\{|y|, y \in f'(x^{(0)})\} = m_{f'} > 0$, *and*

(g) $KM_f/m_{f'}^2 = \beta < 1$.

Then

1. $\mathrm{w}\big(N(f;x^{(0)},\check{x}^{(0)})\big) \leq \beta \mathrm{w}\big(x^{(0)}\big)$.

2. *Furthermore, if*

$$x^{(k+1)} = x^{(k)} \cap N(f;x^{(k)},\check{x}^{(k)}), \tag{1.39}$$

then $\mathrm{w}\big(x^{(k+1)}\big) = \mathcal{O}\left(\mathrm{w}\big(x^{(k)}\big)\right)^2$.

[30] Although methods differ according to the interval extension of the derivative and whether a derivative or slope is used, all such variants will be viewed as a single method here.

[31] This theorem is just meant to be an example that is easy to understand, not the sharpest or most general result in the literature.

Proof: Property 1 will be proved first. Only the case $f(\check{x}^{(k)}) > 0$ and $\boldsymbol{f}'(\boldsymbol{x}^{(0)}) = [a, b] > 0$ will be considered, since the other cases are similar. In that case,

$$\begin{aligned} \mathrm{w}\Big(\boldsymbol{N}(f; \boldsymbol{x}^{(0)}, \check{x}^{(0)})\Big) &= \mathrm{w}\left(\frac{f(\check{x}^{(0)})}{[a,b]}\right) \\ &= \mathrm{w}\left(\left[\frac{f(\check{x}^{(0)})}{b}, \frac{f(\check{x}^{(0)})}{a}\right]\right) = f(\check{x}^{(0)})\frac{b-a}{ab} \\ &\leq M_f \mathrm{w}\Big(\boldsymbol{f}'(\boldsymbol{x}^{(0)})\Big)/m_{f'}^2 \\ &\leq K M_f \mathrm{w}\Big(\boldsymbol{x}^{(0)}\Big)/m_{f'}^2 = \beta \mathrm{w}\Big(\boldsymbol{x}^{(0)}\Big). \end{aligned}$$

To prove property 2, we proceed as with property 1, assuming the case $f(\check{x}^{(k)}) > 0$ and $\boldsymbol{f}'(\boldsymbol{x}^{(k)}) = [a_k, b_k] > 0$: $\mathrm{w}\big(\boldsymbol{x}^{(k+1)}\big) \leq \mathrm{w}\big(\boldsymbol{N}(f; \boldsymbol{x}^{(k)}, \check{x}^{(k)})\big) = f(\check{x}^{(k)})\frac{b_k - a_k}{a_k b_k}$. However, $\check{x}^{(k)} = x^* + (\check{x}^{(k)} - x^*)$, so $f(\check{x}^{(k)}) = f'(\eta)(\check{x}^{(k)} - x^*) \leq b_k \mathrm{w}\big(\boldsymbol{x}^{(k)}\big)/2$ since $x^* \in \boldsymbol{x}^{(k)}$ and $\check{x}^{(k)}$ is the midpoint of $\boldsymbol{x}^{(k)}$. Therefore, $\{f(\check{x}^{(k)})\}\left\{\frac{b_k - a_k}{a_k b_k}\right\} \leq \{b_k \mathrm{w}\big(\boldsymbol{x}^{(k)}\big)/2\}\{K M_f \mathrm{w}\big(\boldsymbol{x}^{(k)}\big)/(m_{f'} b_k)\} = \frac{K M_f}{2 m_{f'}} \mathrm{w}\big(\boldsymbol{x}^{(k)}\big)^2$, thus proving property 2 for this case. The other cases are similar. □

Equation (1.39) may be used directly to reduce the size of the initial interval. In fact, this equation is applicable even when $0 \in \boldsymbol{f}'(\boldsymbol{x}^{(0)})$, because the set inclusion properties of the Kahan arithmetic of §1.1.4 are the same as those of ordinary interval arithmetic:

Example 1.4 (Extended interval arithmetic in an interval Newton method) *Let $f(x) = x^2 - 4$, and let $\boldsymbol{x}^{(0)} = [-2, 2]$, and $\check{x}^{(0)} = 0$, so the range of f' is $\boldsymbol{f}'(\boldsymbol{x}^{(0)}) = [-4, 4]$. The first iteration of the interval Newton method thus becomes*

$$\begin{aligned} \boldsymbol{N}(f; \boldsymbol{x}^{(0)}, \check{x}^{(0)}) &= 0 - \frac{-4}{[-4,4]} = 0 - ([-\infty, -1] \cup [1, \infty]) \\ &= ([-\infty, -1] \cup [1, \infty]). \end{aligned}$$

Thus,

$$\boldsymbol{N}(f; \boldsymbol{x}^{(0)}, \check{x}^{(0)}) \cap \boldsymbol{x}^{(0)} = [-2, -1] \cup [1, 2] = \boldsymbol{x}^{(1,1)} \cup \boldsymbol{x}^{(1,2)}.$$

One of $\boldsymbol{x}^{(1,1)} = [-2, -1]$ and $\boldsymbol{x}^{(1,2)} = [1, 2]$, say $\boldsymbol{x}^{(1,1)}$, is now put on a stack for later processing, and the interval Newton method is iterated with $\boldsymbol{x}^{(1,2)}$. The latter, implemented in the software system of §2.2.1 below, gives the iterates[32]

[32]The last digits in $\boldsymbol{x}^{(k)}$ and $\check{x}^{(k)}$ have been rounded out for rigorous inclusions.

k	$\boldsymbol{x}^{(k)}$	$\check{x}^{(k)}$	$\mathrm{w}\left(\boldsymbol{x}^{(k)}\right)$	rat.
1	[0.9999999999999997, 2.0000000000000005]	1.5000000000000000	1.00	—
2	[1.9374999999999990, 2.3750000000000019]	2.1562500000000004	4.38E-01	4E-1
3	[1.9886592741935471, 2.0195312500000010]	2.0040952620967740	3.09E-02	2E-1
4	[1.9999724292486506, 2.0000354537727550]	2.0000039415107027	6.30E-05	7E-2
5	[1.9999999999417791, 2.0000000000659869]	2.0000000000038831	1.24E-10	3E-2
6	[1.9999999999999988, 2.0000000000000010]	2.0000000000000000	2.00E-15	1E+5

Table 1.3 Illustration of quadratic convergence of the univariate interval Newton method for $f(x) = x^2 - 4$

in Table 1.3. There, the last column, labelled "rat.", represents the ratios $\mathrm{w}\left(\boldsymbol{x}^{(k)}\right)/\mathrm{w}\left(\boldsymbol{x}^{(k-1)}\right)^2$.

Provided the case $f(\check{x}) = 0$ with $0 \in \boldsymbol{f}'(\boldsymbol{x})$ does not occur, the univariate interval Newton method combined with Kahan–Novoa–Ratz arithmetic and deferred processing of additional boxes often converges to small boxes containing all of the roots of f. Such a univariate interval Newton method algorithm can be summarized in

Algorithm 5 (Univariate interval Newton search, similar to [77, §7.4])

INPUT: f, $\boldsymbol{x}$, and a tolerance ϵ.

OUTPUT: A list $\mathcal{C}$ of narrow intervals of length at most ϵ that possibly contain roots.

1. *Initialize:* $\mathcal{U} \leftarrow \{\boldsymbol{x}\}$.
2. DO *while* $\mathcal{U} \neq \emptyset$.
   (a) *Remove an interval* $\boldsymbol{x}$ *from* $\mathcal{U}$.
   (b) DO *while* $\mathrm{w}(\boldsymbol{x}) > \epsilon$.
      i. *Compute* $\boldsymbol{N}(f; \boldsymbol{x}, \check{x}) \cap \boldsymbol{x}$ *for some* $\check{x} \in \boldsymbol{x}$.
      ii. IF $\boldsymbol{N}(f; \boldsymbol{x}, \check{x}) \cap \boldsymbol{x}$ *consists of a single interval*
         THEN
            $\boldsymbol{x} \leftarrow \boldsymbol{N}(f; \boldsymbol{x}, \check{x}) \cap \boldsymbol{x}$.
         ELSE
            Put one of the intervals into $\mathcal{U}$.
            Set the other interval to $\boldsymbol{x}$.

END IF
END DO
(c) Store x in C.
END DO

End Algorithm 5

A relative width $w(x)/\max\{|\underline{x}|, |\overline{x}|\}$ may be used in Step 2b. For rigor, all of the computations to obtain $N(f; x, \check{x})$, including evaluation of $f(\check{x})$, should be done with outwardly rounded interval arithmetic.

For a summary of the theory underlying Algorithm 5, as well as for extensive additional commentary, see [77, Chapter 7]. Algorithm 5 is simplified, and does not include steps to verify existence or uniqueness within the boxes in U. Such techniques are explained in the general, multidimensional context in §1.5.2.

Algorithm 5 should function satisfactorily for a variety of f, although more sophisticated techniques are necessary for problems such as the Gritton problem studied in [114], in which there is a combination of serious overestimation in the derivative values and cancellation error.

Multivariate Interval Newton Methods

Although analogous to the univariate interval Newton methods above, multivariate interval Newton methods have more variety. Different univariate interval Newton methods of formulas (1.37) and (1.38) vary in the choice of base point $\check{x}$ and interval extension of the derivative or slope. In multivariate interval Newton methods, not only can the base point $\check{X}$ and interval extension $\boldsymbol{F}'(\boldsymbol{X})$ or $\mathbf{S}(\boldsymbol{F}, \boldsymbol{X}, \check{\boldsymbol{X}})$ be varied, but there is a wider variety of ways of bounding the solution sets of the resulting linear systems than in the univariate case; cf. §1.2.2. Furthermore, there are additional issues in dealing with interval slopes [120, §3.3] and [210]. Finally, there are questions concerning preconditioning the related interval linear systems. Differences in how interval Newton methods are used in computational existence and uniqueness tests are discussed later in this chapter in §1.5.2. Extensive new results on theory and practice of preconditioning systems are presented later in Chapter 3. Further discussion (besides that in §1.2.2) of the differences between the Krawczyk method, interval Gaussian elimination, and the interval Gauss–Seidel method also occurs below and in Chapter 3.

Multivariate interval Newton methods are closely tied to the methods for linear systems discussed in §1.2. In particular, multivariate interval Newton methods involve iterative execution of algorithms to bound the solution sets of linear systems, except that the linear systems are of the special form

$$\boldsymbol{A}(X - \check{X}) = -F(\check{X}),$$

where $\check{X}$ is an approximation to a root of F (and hence $\|F(\check{X})\|$ is small), and $\boldsymbol{A}$ is either $\boldsymbol{F}'(\boldsymbol{X})$ or $\mathbf{S}(\boldsymbol{F}, \boldsymbol{X}, \check{\boldsymbol{X}})$. Thus, interval Newton methods involve bounding the solution sets to approximately homogeneous systems of equations, and there is more structure and symmetry than in the general systems of §1.2.

Widely studied, the Krawczyk method was introduced in [140]. The power of the Krawczyk method as a computational fixed point theorem was first presented and analyzed in [164]. The Krawczyk method is based on considering the classical multivariate Newton method as a fixed point iteration. In particular, suppose $F : \mathbb{R}^n \to \mathbb{R}^n$, and solutions to $F(X) = 0$ within the interval vector $\boldsymbol{X}$ are sought. Let $\check{X} \in \boldsymbol{X}$, $A = F'(\check{X})$ denote the Jacobi matrix of F at $\check{X}$, and let Y denote some sort of approximation to A^{-1}. Then the chord method based on iteration matrix Y can be viewed as a fixed point iteration with iteration function

$$P(X) := X - YF(X). \tag{1.40}$$

Form a mean value extension for P over $\boldsymbol{X}$ as in Definition 1.6:

$$\boldsymbol{P}(\boldsymbol{X}) = P(\check{X}) + \boldsymbol{P}'(\boldsymbol{X})(\boldsymbol{X} - \check{X}). \tag{1.41}$$

However, $P'(X) = I - YF'(X)$, where I is the $n \times n$ identity matrix. Thus, a natural interval extension $\boldsymbol{P}'(\boldsymbol{X})$ of P' over $\boldsymbol{X}$ is $I - Y\boldsymbol{F}'(\boldsymbol{X})$, where $\boldsymbol{F}'(\boldsymbol{X})$ is an interval extension of F over $\boldsymbol{X}$. Combining this with equations (1.40) and (1.41) gives

$$\boldsymbol{P}(\boldsymbol{X}) := \mathbf{K}(\boldsymbol{X}, \check{X}) = \check{X} - YF(\check{X}) + (I - Y\boldsymbol{F}'(\boldsymbol{X}))(\boldsymbol{X} - \check{X}). \tag{1.42}$$

Definition 1.16 *The operator* $\mathbf{K}(\boldsymbol{X}, \check{X}) : \mathbb{IR}^n \to \mathbb{IR}^n$ *is termed the* Krawczyk operator. *The iteration* $\boldsymbol{X} \leftarrow \mathbf{K}(\boldsymbol{X}, \check{X}) \cap \boldsymbol{X}$ *is termed the* Krawczyk method.

Now suppose that $X^* \in \boldsymbol{X}$ is a point for which $F(X^*) = 0$. Then $P(X^*) = X^*$, so $X^* \in \boldsymbol{P}(\boldsymbol{X}) = \mathbf{K}(\boldsymbol{X}, \check{X})$. In other words *any roots of* F *in* $\boldsymbol{X}$ *are also in* $\mathbf{K}(\boldsymbol{X}, \check{X})$. Furthermore, under certain conditions, such as when $\check{X}$ is taken

to be the midpoint vector of $\boldsymbol{X}$, Y is appropriate, the interval extensions are sufficiently sharp, and the widths of the components of $\boldsymbol{X}$ are sufficiently small, iteration of $\boldsymbol{X} \leftarrow \mathbf{K}(\boldsymbol{X}, \check{X}) \cap \boldsymbol{X}$ converges quadratically in the same sense as Theorem 1.14.

Example 1.5 *Take*

$$\begin{aligned} f_1(x_1, x_2) &= x_1^2 - 4x_2, \\ f_2(x_1, x_2) &= x_2^2 - 2x_1 + 4x_2, \end{aligned}$$

and

$$\boldsymbol{X} = (x_1, x_2)^T = ([-0.1, 0.1], [-0.1, 0.3])^T.$$

There is a unique root $X^* = (0,0)^T$ *of* $F = (f_1, f_2)^T$ *within* $\boldsymbol{X}$.

In Example 1.5, if $\check{X} = (0, .1)^T$, then $F(\check{X}) = (-.4, .41)^T$, and the matrix

$$\boldsymbol{F}'(\boldsymbol{X}) = \begin{pmatrix} 2x_1 & -4 \\ -2 & 2x_2 + 4 \end{pmatrix} = \begin{pmatrix} [-.2, .2] & -4 \\ -2 & [3.8, 4.6] \end{pmatrix}$$

has components consisting of the ranges of corresponding components of $F'(X)$ over $\boldsymbol{X}$. If Y is taken to be the inverse of the midpoint matrix of $\boldsymbol{F}'(\boldsymbol{X})$, then

$$Y = \{\mathrm{m}(\boldsymbol{F}'(\boldsymbol{X}))\}^{-1} = \begin{pmatrix} -.525 & -.5 \\ -.25 & 0 \end{pmatrix},$$

and the first iteration of the Krawczyk method becomes

$$\begin{aligned} \mathbf{K}(\boldsymbol{X}, \check{X}) &= \begin{pmatrix} 0.0 \\ 0.1 \end{pmatrix} - \begin{pmatrix} -.525 & -0.5 \\ -.25 & 0.0 \end{pmatrix} \begin{pmatrix} -.4 \\ .41 \end{pmatrix} \\ &\quad + \left\{ I - \begin{pmatrix} -.525 & -0.5 \\ -.25 & 0.0 \end{pmatrix} \begin{pmatrix} [-.2, .2] & -4 \\ -2 & [3.8, 4.6] \end{pmatrix} \right\} \left\{ \begin{pmatrix} [-.1, .1] \\ [-.1, .3] \end{pmatrix} - \begin{pmatrix} 0.0 \\ 0.1 \end{pmatrix} \right\} \\ &= \begin{pmatrix} -.005 \\ 0 \end{pmatrix} + \begin{pmatrix} [-.105, .105] & [-.2, .2] \\ [-.05, .05] & 0 \end{pmatrix} \begin{pmatrix} [-.1, .1] \\ [-.2, .2] \end{pmatrix} \\ &= \begin{pmatrix} -.005 \\ 0 \end{pmatrix} + \begin{pmatrix} [-.0505, .0505] \\ [-.005, .005] \end{pmatrix} \\ &= \begin{pmatrix} [-.0555, .0555] \\ [-.005, .005] \end{pmatrix} \subset \boldsymbol{X}, \\ \boldsymbol{X}^{(1)} &= \mathbf{K}(\boldsymbol{X}, \check{X}) \cap \boldsymbol{X} = \mathbf{K}(\boldsymbol{X}, \check{X}). \end{aligned}$$

If the iteration is continued, taking $\check{X}^{(k)}$ to be the midpoint of $\boldsymbol{X}^{(k)}$ and $Y^{(k)}$ to be the inverse of the matrix of midpoints of the components of $\boldsymbol{F}'(\boldsymbol{X}^{(k)})$,

k	$x_1^{(k)}$	$x_2^{(k)}$	$\|\mathrm{w}\left(X^{(k)}\right)\|$	rat.
0	[-0.11E-00, 0.11E-00]	[-0.11E-00, 0.31E-00]	0.40E-00	—
1	[-0.57E-01, 0.47E-01]	[-0.51E-02, 0.51E-02]	0.10E-00	0.63
2	[-0.27E-02, 0.27E-02]	[-0.14E-02, 0.14E-02]	0.51E-02	0.50
3	[-0.83E-05, 0.83E-05]	[-0.34E-05, 0.34E-05]	0.16E-04	0.62
4	[-0.79E-10, 0.79E-10]	[-0.34E-10, 0.34E-10]	0.16E-09	0.58
5	[-0.72E-20, 0.72E-20]	[-0.31E-20, 0.31E-20]	0.14E-19	0.59

Table 1.4 Illustration of quadratic convergence of the Krawczyk method for F as in Example 1.5

then Table 1.4 is obtained[33]. Analogously to Table 1.3 for the univariate example, the column labelled "rat." contains the ratios $\|\mathrm{w}\left(\boldsymbol{X}^{(k)}\right)\|/\|\mathrm{w}\left(\boldsymbol{X}^{(k-1)}\right)\|^2$. Since these ratios are nearly constant, the Krawczyk method is evidently quadratically convergent in practice.

For a given preconditioning matrix Y, the Krawczyk method is not necessarily as sharp as alternate interval Newton methods; cf. Theorem 1.8 on page 21. However, the Krawczyk method has various theoretical advantages, such as is outlined in [208]. Also, as mentioned on page 24 [208], the Krawczyk method can be used to obtain inner estimates.

The interval Gauss–Seidel method and interval Gaussian elimination are more directly related to the classical Newton method. In particular, derivation can proceed as in the univariate case on page 51: Suppose $F : \boldsymbol{X} \subset \mathbb{R}^n \to \mathbb{R}^n$, $X^* \in \boldsymbol{X}$ is such that $F(X^*) = 0$, and $\check{X} \in \boldsymbol{X}$. Then the mean value theorem implies

$$0 = F(X^*) = F(\check{X}) + A(X^* - \check{X})$$

for some $A \in \mathbf{S}(F, \boldsymbol{X}, \check{X})$, where $\mathbf{S}(F, \boldsymbol{X}, \check{X})$ is any set satisfying Definition 1.12. Thus, $X^* - \check{X}$ must be in the solution set to

$$\boldsymbol{A}(X - \check{X}) = -F(\check{X}), \tag{1.43}$$

for $\boldsymbol{A} = \mathbf{S}(F, \boldsymbol{X}, \check{X})$ or $\boldsymbol{A} = \boldsymbol{F}'(\boldsymbol{X})$, for some $\mathbf{S}(F, \boldsymbol{X}, \check{X})$ or $\boldsymbol{F}'(\boldsymbol{X})$. That is, any root X^* of F within $\boldsymbol{X}$ must be within $\check{X} + \tilde{\boldsymbol{X}}$, where $\tilde{\boldsymbol{X}}$ is any bounding interval for the solution set of Equation (1.43). Bounding this solution set with the interval Gauss–Seidel algorithm as on page 22 results in the interval Gauss–Seidel method. The vector $\boldsymbol{B}$ for Algorithm 2 is simply replaced by $-F(\check{X})$,

[33]The coordinates of $x_1^{(k)}$ and $x_2^{(k)}$ have been rounded out at two decimal digits, although rigorous bounds have been obtained to many more digits with the systems described in §2.1. and §2.2

although the iteration formula in Step 2 of Algorithm 2 can be replaced by

$$\tilde{\boldsymbol{x}}_i = \check{x}_i - \left\{ Y_i F(\check{X}) \;+\; \sum_{j=1}^{i-1} Y_i \boldsymbol{A}_j (\tilde{\boldsymbol{x}}_j - \check{x}_j) \right. \\ \left. +\; \sum_{j=i+1}^{n} Y_i \boldsymbol{A}_j (\boldsymbol{x}_j - \check{x}_j) \right\} \Big/ (Y_i \boldsymbol{A}_i), \qquad (1.44)$$

so that $\tilde{\boldsymbol{X}}$ may be computed directly.

Interval Gaussian elimination as on page 21 may also be used to bound the solution set.

Most interval Newton methods, such as those above, can be put into the general form

$$\tilde{\boldsymbol{X}} = \boldsymbol{N}(F; \boldsymbol{X}, \check{X}) = \check{X} + \boldsymbol{V}, \qquad (1.45)$$

where

$$\Sigma(\boldsymbol{A}, -F(\check{X})) \subset \boldsymbol{V}.$$

Generally, $\boldsymbol{V}$ is obtained from a method such as those in §1.2.

1.5.2 Derivative and Slope-Based Existence and Uniqueness

Here, the problems

> Given $F : \boldsymbol{X} \to \mathbb{R}^n$ and $\boldsymbol{X} \in \mathbb{IR}^n$, *rigorously* verify one of the following:
>
> - there exists a unique $X^* \in \boldsymbol{X}$ such that $F(X^*) = 0$,
> - there exists an $X^* \in \boldsymbol{X}$ such that $F(X^*) = 0$,
> - there is no $X^* \in \boldsymbol{X}$ such that $F(X^*) = 0$.

(1.46)

are considered. Computer arithmetic can be used to verify the assertions in Problem (1.46), with the aid of interval extensions or *computational fixed point theorems*. This theory is the subject of this section.

The quadratic convergence properties of interval Newton methods, such as Theorem 1.14, as well as Exercise 3 on page 65 below, and the existence / uniqueness

theory outlined in this section are analogous to the Kantorovich theorem for the classical Newton method, as stated in [48, p. 92]. In fact, comparisons, such as in [178], [186], and [220] compare interval Newton methods favorably with the Kantorovich theorem when the latter is used in computations.

The existence theory for interval Newton methods can be derived largely from the Brouwer fixed point theorem and the related Miranda's theorem, while uniqueness follows largely from non-singularity considerations.

Theorem 1.15 (Brouwer fixed point theorem, [25]) *Let* $\mathbf{D}$ *be homeomorphic to the closed unit ball in* $\mathbb{R}^n$*, and suppose* P *is a continuous mapping such that the range* $\boldsymbol{P}^u(\mathbf{D}) \subset \mathbf{D}$*. Then* P *has a fixed point, i.e. there is an* $X \in \mathbf{D}$ *such that* $P(X) = X$*.*

In particular, if P maps $\boldsymbol{X}$ strictly into $\boldsymbol{X}$, where $\boldsymbol{X} \in \mathbb{IR}^n$, then there is an $X \in \boldsymbol{X}$ such that $P(X) = X$.

Theorem 1.16 (Miranda's theorem, [161]) *Suppose* $\boldsymbol{X} \in \mathbb{IR}^n$*, and let the faces of* $\boldsymbol{X}$ *be denoted by*

$$\begin{aligned} \boldsymbol{X}_{\underline{i}} &= (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \underline{x}_i, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_n)^T \\ \boldsymbol{X}_{\overline{i}} &= (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{i-1}, \overline{x}_i, \boldsymbol{x}_{i+1}, \ldots, \boldsymbol{x}_n)^T . \end{aligned}$$

Let $F = (f_1, \ldots, f_n)^T$ *be a continuous function defined on* $\boldsymbol{X}$*. If*

$$\boldsymbol{f}_i{}^u(\boldsymbol{X}_{\underline{i}}) \boldsymbol{f}_i{}^u(\boldsymbol{X}_{\overline{i}}) \leq 0 \tag{1.47}$$

for each i *between* 1 *and* n*, then there is an* $X \in \boldsymbol{X}$ *such that* $F(X) = 0$*.*

In particular, the exact ranges $\boldsymbol{f}_i{}^u$ may be replaced by any interval extensions of the f_i in Condition 1.47.

Existence Theory

The Krawczyk method, derived as a fixed-point method, has a clear relationship to the Brouwer fixed point theorem:

Theorem 1.17 *Suppose* $\mathbf{K}(\boldsymbol{X}, \check{X})$ *is as in Equation (1.42), where* $F : \boldsymbol{X} \to \mathbb{R}^n$ *and* $\boldsymbol{F}'(\boldsymbol{X})$ *is either a Lipschitz matrix as in Definition 1.10 or a slope*

matrix as in Definition 1.12 (on page 27), and where Y is non-singular. If $\mathbf{K}(\boldsymbol{X}, \check{X}) \subset \boldsymbol{X}$, then there exists a point $X^* \in \boldsymbol{X}$ such that $F(X^*) = 0$.

Proof: Let $P(X) = X - YF(X)$ be as in Equation (1.40). Since $\mathbf{K}(\boldsymbol{X}, \check{X})$ is a mean value extension for P over $\boldsymbol{X}$, $\boldsymbol{P}^u(\boldsymbol{X}) = \{P(X) | X \in \boldsymbol{X}\} \subseteq \mathbf{K}(\boldsymbol{X}, \check{X})$. Therefore, the Brouwer fixed point theorem implies there is an $X^* \in \boldsymbol{X}$ such that $P(X^*) = X^*$. However, since Y is non-singular, $P(X) = X$ if and only if $F(X) = 0$. Therefore, $F(X^*) = 0$. □

Similarly, Miranda's theorem can be used to prove an analogous existence verification property of the interval Gauss–Seidel method.

Theorem 1.18 *Suppose $\tilde{\boldsymbol{x}}_i$ is defined by Formula (1.44) for i between 1 and n, where $F : \boldsymbol{X} \to \mathbb{R}^n$, where $\boldsymbol{A}$ is either a Lipschitz matrix or a slope matrix for F over $\boldsymbol{X}$, and where Y is non-singular. If $\tilde{\boldsymbol{x}}_i \subseteq \boldsymbol{x}_i$ for i between 1 and n, where the set inclusion is tested before $\tilde{\boldsymbol{x}}_i$ is replaced by $\boldsymbol{x}_i \cap \tilde{\boldsymbol{x}}_i$, then there exists an $X^* \in \boldsymbol{X}$ such that $F(X^*) = 0$.*

Proof: Notation for the preconditioned function $YF(X)$ will first be introduced. Define $G(X) = (g_1(X), \ldots, g_n(X))^T = YF(X)$, and define $\boldsymbol{M} = Y\boldsymbol{A} = (\boldsymbol{m}_{i,j})_{i,j=1}^n$. Also define

$$\gamma_i(x_i) = g_i(\check{X}) + \left\{ \sum_{\substack{j=1 \\ j \neq i}}^{n} \boldsymbol{m}_{i,j}(\boldsymbol{x}_j - \check{x}_j) \right\} + \boldsymbol{m}_{i,i}(x_i - \check{x}_i).$$

With the notation of Theorem 1.16, observe that $\boldsymbol{g}_i{}^u(\boldsymbol{X}_{\underline{i}}) \subseteq \gamma_i(\underline{x}_i)$ and $\boldsymbol{g}_i{}^u(\boldsymbol{X}_{\overline{i}}) \subseteq \gamma_i(\overline{x}_i)$.

Following this notation and Formula (1.44), the condition $\tilde{\boldsymbol{x}}_1 \subseteq \boldsymbol{x}_1$ can be stated as

$$-\frac{g_1(\check{X}) + \sum_{\substack{j=1 \\ j \neq i}}^{n} \boldsymbol{m}_{1,j}(\boldsymbol{x}_j - \check{x}_j)}{\boldsymbol{m}_{1,1}} \subseteq [\underline{x}_1 - \check{x}_1, \overline{x}_1 - \check{x}_1]. \tag{1.48}$$

With just the left endpoint $\underline{x}_1 - \check{x}_1$, the inclusion (1.48) implies

$$\frac{g_1(\check{X}) + \sum_{\substack{j=1 \\ j \neq i}}^{n} \boldsymbol{m}_{1,j}(\boldsymbol{x}_j - \check{x}_j)}{\boldsymbol{m}_{1.1}} + \underline{x}_1 - \check{x}_1 \leq 0. \tag{1.49}$$

However, it is implicit in the inclusion (1.48) that $0 \notin \boldsymbol{m}_{1,1}$. First assume that $\boldsymbol{m}_{1,1} > 0$. Then, multiplying by $\boldsymbol{m}_{1,1}$, the inequality (1.49) implies

$$g_1(\check{X}) + \left\{ \sum_{\substack{j=1 \\ j \neq i}}^{n} \boldsymbol{m}_{1,j}(\boldsymbol{x}_j - \check{x}_j) \right\} + \boldsymbol{m}_{1,1}(\underline{x}_1 - \check{x}_1) = \gamma_1(\underline{x}_1) \leq 0.$$

Similarly, with the right endpoint, the inclusion (1.48) and $\boldsymbol{m}_{1,1} > 0$ imply $\gamma_1(\overline{x}_1) \geq 0$. A similar argument for the case $\boldsymbol{m}_{1,1} < 0$ gives $\gamma_1(\underline{x}_1) \geq 0$ and $\gamma_1(\overline{x}_1) \leq 0$. Thus, $\tilde{\boldsymbol{x}}_1 \subseteq \boldsymbol{x}_1$ implies

$$\boldsymbol{g}_1{}^u(\boldsymbol{X}_{\underline{1}})\boldsymbol{g}_1{}^u(\boldsymbol{X}_{\overline{1}}) \leq 0.$$

To show $\boldsymbol{g}_2{}^u(\boldsymbol{X}_{\underline{2}})\boldsymbol{g}_2{}^u(\boldsymbol{X}_{\overline{2}}) \leq 0$, replace $\boldsymbol{x}_1$ by $\tilde{\boldsymbol{x}}_1$. Then, with an argument identical to that above, but replacing $[\underline{x}_i - \check{x}_i, \overline{x}_i - \check{x}_i]$ by $[\underline{\tilde{x}}_i - \check{x}_i, \overline{\tilde{x}}_i - \check{x}_i]$, $\boldsymbol{g}_1{}^u(\boldsymbol{X}_{\underline{1}})\boldsymbol{g}_1{}^u(\boldsymbol{X}_{\overline{1}}) \leq 0$ still holds for this new $\boldsymbol{X}$. Furthermore, the same argument with g_2 replacing g_1 and x_2 replacing x_1 then also implies

$$\boldsymbol{g}_2{}^u(\boldsymbol{X}_{\underline{2}})\boldsymbol{g}_2{}^u(\boldsymbol{X}_{\overline{2}}) \leq 0$$

for this new $\boldsymbol{X}$, provided $\tilde{\boldsymbol{x}}_2 \subseteq \boldsymbol{x}_2$.

The entire argument above can then be repeated for $i = 3$ to n to give

$$\boldsymbol{g}_i{}^u(\tilde{\boldsymbol{X}}_{\underline{i}})\boldsymbol{g}_i{}^u(\tilde{\boldsymbol{X}}_{\overline{i}}) \leq 0 \qquad 1 \leq i \leq n,$$

where $\tilde{\boldsymbol{X}}$ is the image of $\boldsymbol{X}$ under a Gauss–Seidel sweep given by Formula (1.44) and Algorithm 2. Thus, by Miranda's theorem, there is an $X^* \in \tilde{\boldsymbol{X}} \subseteq \boldsymbol{X}$ such that $G(X^*) = 0$. However, since $G = YF$ and Y is non-singular, this then implies $F(X^*) = 0$. □

The literature is replete with variations on such computational existence theorems. One such fairly general theorem is

Theorem 1.19 *([175, Theorem 5.1.7] and earlier) Suppose $F : \boldsymbol{X} \to \mathbb{R}^n$, suppose $\boldsymbol{A}$ is a regular Lipschitz matrix (e.g. an interval Jacobi matrix $\boldsymbol{F}'(\boldsymbol{X})$) for F over $\boldsymbol{X}$, suppose $\check{X} \in \operatorname{int}(\boldsymbol{X})$, and suppose*

$$\tilde{\boldsymbol{X}} = \check{X} + \boldsymbol{V},$$

where $\boldsymbol{V} \in \mathbb{IR}^n$ is any interval vector such that

$$\Sigma(\boldsymbol{A}, -F(\check{X})) \subset \boldsymbol{V}.$$

Then, if $\tilde{\boldsymbol{X}} \subseteq \boldsymbol{X}$, *it follows that there exists a unique solution of* $F(X) = 0$ *within* $\boldsymbol{X}$.

Theorem 1.19 encompasses the interval Gauss-Seidel method, interval Newton methods based on interval Gaussian elimination, and interval Newton methods based on any other technique for bounding the solution set $\Sigma(\boldsymbol{A}, -F(\check{X}))$. Note that it is an existence and uniqueness theorem, and it is assumed that $\boldsymbol{A}$ is a Lipschitz matrix. A similar existence theorem is possible if $\boldsymbol{A}$ is merely a slope matrix, but uniqueness requires additional consideration in that case. An alternate version of Theorem 1.19 is proven as Theorem 1.22 below.

Uniqueness

Uniqueness can proceed from verification of regularity. To this end, we have:

Lemma 1.20 *Suppose* $\tilde{\boldsymbol{X}} = \check{X} + \boldsymbol{V}$ *is the image under the interval Newton method (Formula (1.45)), where* $\boldsymbol{V}$ *is computed by any method that bounds the solution set* $\Sigma(\boldsymbol{A}, -F(\check{X}))$ *and* $\tilde{\boldsymbol{X}} \subseteq \boldsymbol{X}$. *Then* $\boldsymbol{A}$ *is regular.*

Proof: If $\boldsymbol{A}$ were not regular, then $\Sigma(\boldsymbol{A}, -F(\check{X}))$ and hence $\boldsymbol{V}$ and $\tilde{\boldsymbol{X}}$ would be unbounded. □

When the interval Newton image obeys $\boldsymbol{N}(F; \boldsymbol{X}, \check{X}) \subseteq \boldsymbol{X}$, uniqueness can be inferred when a Lipschitz matrix is used.

Lemma 1.21 *Suppose* $F : \boldsymbol{X} \to \mathbb{R}^n$ *and* $\boldsymbol{A}$ *is a Lipschitz matrix, such as* $\boldsymbol{F}'(\boldsymbol{X})$. *If* $\boldsymbol{A}$ *is regular, then any root of* F *in* $\boldsymbol{X}$ *is unique.*

Proof: Suppose $X^* \in \boldsymbol{X}$ and $X \in \boldsymbol{X}$ have $F(X^*) = 0$ and $F(X) = 0$. If $\boldsymbol{A}$ is a Lipschitz set, then there is an $A \in \boldsymbol{A}$ such that

$$F(X^*) - F(X) = 0 = A(X^*) - A(X) = A(X^* - X).$$

If $X^* \neq X$, then A would have a null vector, contradicting the regularity of $\boldsymbol{A}$. □

Combining Lemma 1.21 and Lemma 1.20 gives

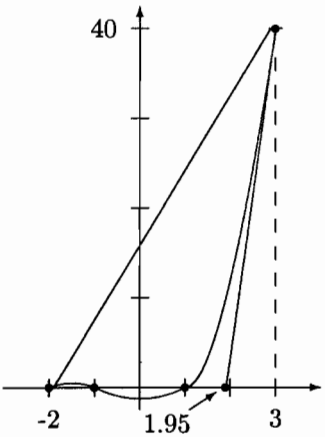


Figure 1.6 Non-uniqueness with slopes with $f(x) = (x^2 - 1)(x + 2)$, $\boldsymbol{x} = [-2, 3]$ and $\check{x} = 3$

Theorem 1.22 *Suppose $F : \boldsymbol{X} \subset \mathbb{R}^n \to \mathbb{R}^n$ and $\boldsymbol{A}$ is a Lipschitz matrix such as $\boldsymbol{F}'(\boldsymbol{X})$. If $\tilde{\boldsymbol{X}}$ is the image under an interval Newton method as in Formula (1.45) and $\tilde{\boldsymbol{X}} \subseteq \boldsymbol{X}$, then there is a unique $X^* \in \boldsymbol{X}$ with $F(X^*) = 0$.*

On the other hand, if $\boldsymbol{A}$ is merely a slope set $\mathbf{S}(\boldsymbol{F}, \boldsymbol{X}, \check{\boldsymbol{X}})$, uniqueness cannot be inferred in this way. As an example, take $f(x) = (x^2 - 1)(x + 2)$, $\boldsymbol{x} = [-2, 3]$ and $\check{x} = 3$, as in Figure 1.6. Then the exact slope set is $\mathbf{S}^\sharp(f, [-2, 3], 3) = [8, 38]$, and $0 \notin [8, 38]$. Furthermore, an interval Newton step gives

$$\tilde{\boldsymbol{x}} = \check{x} - f(\check{x})/\mathbf{S}^\sharp(f, \boldsymbol{x}, \check{x}) = 3 - 40/[8, 38] \subseteq 3 - [1.05, 5] = [-2, 1.95] \subset \boldsymbol{x},$$

but there are nonetheless three roots of f in $\boldsymbol{x}$. The slopes of the oblique lines in Figure 1.6 are the bounds on $\mathbf{S}^\sharp(f, \boldsymbol{x}, \check{x})$; those lines' intersection with the x-axis represents the image $\tilde{\boldsymbol{x}}$.

As Rump pointed out [208], uniqueness verification is still possible in certain contexts, such as with the ϵ-inflation technique explained in §4.2 below, when $\boldsymbol{A}$ represents a slope matrix $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$. The technique is based on:

Theorem 1.23 *Let $F : \boldsymbol{X} \subset \mathbb{R}^n \to \mathbb{R}^n$, and let $\check{\boldsymbol{X}} \subseteq \boldsymbol{X}$ be such that there exists an $X^* \in \check{\boldsymbol{X}}$ with $F(X^*) = 0$. Let $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ be a slope matrix for F over $\boldsymbol{X}$ at $\check{\boldsymbol{X}}$. If $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ is regular (such as when $\boldsymbol{N}(F; \boldsymbol{X}, \check{X}) \subseteq \boldsymbol{X}$ for $\check{X} \in \check{\boldsymbol{X}}$), then X^* is unique within $\boldsymbol{X}$.*

Proof: If $\widehat{X} \in \boldsymbol{X}$ is such that $F(\widehat{X}) = 0$, then there would be an $A \in \mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ such that $F(\widehat{X}) = F(X^*) = A(\widehat{X} - X^*) = 0$, since $X^* \in \check{\boldsymbol{X}}$. Therefore, $A \in \mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ could not be regular. □

Theorem 1.23 allows verification of uniqueness within larger boxes than when an interval derivative $\boldsymbol{F}'(\boldsymbol{X})$ is used, provided the box $\check{\boldsymbol{X}}$ can be found. This is because slope bounds $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ are generally narrower than interval derivatives $\boldsymbol{F}'(\boldsymbol{X})$.

1.5.3 Exercises

1. (*This exercise is best done with a software system or programming language, such as the system described in §2.2.1 below, that has an interval data type.*) Program interval Gaussian elimination (Algorithm 1 on page 21) to perform interval Newton iteration on the function in Example 1.5. Produce a table analogous to Table 1.4. Do you observe quadratic convergence? Does the convergence (if there is convergence) appear to be faster, slower, or about the same as that in Table 1.4?

2. Repeat Exercise 1, but with interval Gauss–Seidel iteration and Formula (1.44) instead of interval Gaussian Elimination. (*Note that knowledge of §2.2.1 below will help for this problem, too.*)

3. *(This exercise is involved; use of matrix norms and the triangle inequality, as is found in e.g. [61] would be useful.)* This exercise consists of construction of a multidimensional analogue of Theorem 1.14. Suppose $\boldsymbol{X}^{(0)} \in \mathbb{IR}^n$, suppose $F : \boldsymbol{X}^{(0)} \to \mathbb{R}^n$, and suppose that $\boldsymbol{N}(F; \boldsymbol{X}^{(k)}, \check{X}^{(k)})$ is defined by

$$\boldsymbol{N}(F; \boldsymbol{X}^{(k)}, \check{X}^{(k)}) = \check{X}^{(k)} + \boldsymbol{V},$$

where $\boldsymbol{V}$ represents a set of bounds on the solution set of

$$\boldsymbol{F}'(\boldsymbol{X}^{(k)})(X - \check{X}^{(k)}) = -F(\check{X}^{(k)}),$$

where $\check{X}^{(k)}$ is the midpoint vector of $\boldsymbol{X}^{(k)}$, and where the components of $\boldsymbol{F}'(\boldsymbol{X}^{(k)})$ are inclusion monotonic interval extensions of order at least 1 of corresponding components of the real Jacobi matrix $F'(X)$, in the sense of Definition 1.4 on page 14. Assume that $\boldsymbol{V}$ is obtained by a method such as those in §1.2, with

$$\left\| \mathrm{w}\left(\Sigma\left(\boldsymbol{F}'(\boldsymbol{X}^{(k)}), -F(\check{X}^{(k)})\right)\right) - \mathrm{w}(\boldsymbol{V}) \right\| \leq L \left\| \mathrm{w}\left(\Sigma\left(\boldsymbol{F}'(\boldsymbol{X}^{(k)}), -F(\check{X}^{(k)})\right)\right) \right\|$$

for some fixed $L > 0$. (For example, see [175, p. 127].) Also assume that

$$\max\left\{\|A\|_\infty : A \in \boldsymbol{F}'(\boldsymbol{X}^{(0)})\right\} = M_{F'} < \infty$$

and

$$\max\left\{\|A^{-1}\|_\infty : A \in \boldsymbol{F}'(\boldsymbol{X}^{(0)})\right\} = \tilde{M}_{F'} < \infty.$$

Then find a β analogous to that of Theorem 1.14 such that, if $\beta < 1$, then

(a) $\mathrm{w}\left(\boldsymbol{N}(F; \boldsymbol{X}^{(0)}, \check{X}^{(0)})\right) \leq \beta \mathrm{w}\left(\boldsymbol{X}^{(0)}\right)$;

(b) if

$$\boldsymbol{X}^{(k+1)} = \boldsymbol{X}^{(k)} \cap \boldsymbol{N}(F; \boldsymbol{X}^{(k)}, \check{X}^{(k)}),$$

then $\left\|\mathrm{w}\left(\boldsymbol{X}^{(k+1)}\right)\right\| = \mathcal{O}\left(\left\|\mathrm{w}\left(\boldsymbol{X}^{(k)}\right)\right\|\right)^2$.

Hint: This problem is, essentially, solved on page 219 ff.

1.6 THE TOPOLOGICAL DEGREE

Interval Newton methods allow computational existence and uniqueness, useful in verified global optimization, in cases in which the Jacobi matrix is nonsingular in a neighborhood of the critical points or solution of the nonlinear system. However, when the Jacobi matrix at a solution is singular, such methods cannot satisfy the hypotheses of the theorems in §1.5.2. This is because the image $\tilde{\boldsymbol{X}}$ under an interval Newton method like the Krawczyk method or of the form (1.45) must contain the solution set to $\boldsymbol{A}(X - \check{X}) = -F(\check{X})$, where $\boldsymbol{A}$ is either a Lipschitz matrix over $\boldsymbol{X}$ or a slope matrix; since a Lipschitz matrix must contain all Jacobi matrices at points $\boldsymbol{X}$, this solution set would be unbounded if such a Jacobi matrix were singular. Similarly, for uniqueness verification, a slope matrix must be based at a small box $\check{\boldsymbol{X}}$ containing a solution X^* of $F(X) = 0$; such a slope matrix[34] would necessarily contain the Jacobi matrix at X^*.

The topological degree can be of use in verifying critical point structure in such cases, with ill-conditioning or singularity at critical points, or with an unusual critical point structure, such as non-trivial manifolds of critical points. The following illustrative example exhibits the phenomenon, which occurs in practice.

[34] However, existence can possibly still be proven in such cases by basing the slope matrix at some point other than the solution.

Example 1.6 *Define*

$$\phi(x_1, x_2) = \begin{cases} x_1^2 + x_2^2 - 1 & \text{for } x_1^2 + x_2^2 \geq 1 \\ 0 & \text{otherwise,} \end{cases}$$

and suppose that global minimizers within the box

$$\boldsymbol{X} = \left([-2, 2], [-2, 2]\right)^T$$

are sought. The function ϕ is continuous within $\boldsymbol{X}$, the global minimum within $\boldsymbol{X}$ is 0, and each point within the closed unit disk $x_1^2 + x_2^2 \leq 1$ is a global minimizer. However,

$$\nabla\phi(X) = F(X) = (2x_1, 2x_2)^T = (0, 0)^T$$

at every *point within this disk, so an interval Newton method based on $\nabla\phi(X) = 0$ would necessarily fail to exhibit uniqueness, and interval Newton iteration would probably fail to narrow the widths of $\boldsymbol{X}$.*

In such instances, the topological degree allows computation of properties of the solution set within $\boldsymbol{X}$ by just considering the function on the boundary $\partial\boldsymbol{X}$. The topological degree, or Brouwer degree, is a generalization of the concept of winding number in complex analysis. It can be defined in terms of the Kronecker interval [9, pp. 415–437], the Heinz integral [81, 200], or by simplicial approximations [9]; various characterizations can be shown to be equivalent. The conceptually most informative definition begins by assuming that the function $F : \mathbf{D} \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is differentiable, with non-singular Jacobi matrix at its solutions, and generalizes it to continuous functions.

Definition 1.17 *Suppose that $F : \mathbf{D} \to \mathbb{R}^n$ is such that, if $F(X^*) = 0$, then $X^* \notin \partial\mathbf{D}$, where $\mathbf{D}$ is a compact subset of $\mathbb{R}^n$. Also suppose that the Jacobi matrix $F'(X^*)$ is non-singular at every $X^* \in \text{int}(\mathbf{D})$ at which $F(X^*) = 0$. Then the topological degree $\text{d}(F, \mathbf{D}, 0)$ is equal to the number of solutions of $F(X) = 0$ in $\text{int}(\mathbf{D})$ at which the determinant of F' is positive, minus the number of such solutions at which the determinant is negative.*

For example, if F represents a polynomial in the complex plane, with real and imaginary parts represented as separate components, then, from the fundamental theorem of algebra, $\text{d}(F, \mathbf{D}, 0)$ is equal to the algebraic degree of the polynomial when $\mathbf{D}$ is a sufficiently large box in $\mathbb{R}^2$ centered at the origin.

The power of the topological degree for problems in which the critical points or solutions are at singularities of F' lies in two properties, stated roughly here:

1. continuity with respect to deformations in the function F, and
2. dependence only on the function values on $\partial\mathbf{D}$.

Formally, $\mathrm{d}(F, \mathbf{D}, 0)$ is defined for continuous but non-differentiable F by considering approximations to F that obey the conditions in Definition 1.17. Formal presentations appear in [9], [200, Ch. 6], [153], [81], or [43]. Assuming the topological degree has been defined for continuous, but not necessarily differentiable F, the following theorem is relevant to global optimization. It is not stated in its most general form (see [105, p. 18]), but in a form useful in the interval computations context.

Theorem 1.24 *Assume $F : \boldsymbol{X} \to \mathbb{R}^n$ is continuous on the box $\boldsymbol{X} \subset \mathbb{IR}^n$, assume $F(X) \neq 0$ if $X \in \partial\mathbf{D}$, let $s, t \in \{-1, +1\}$, and pick any $i \in \{1, 2, \ldots, n\}$. Then define $X_{\sim j,t} = (x_1, \ldots, x_{j-1}, x_{j,t}, x_{j+1}, \ldots, x_n)^T$, where $x_k \in \boldsymbol{x}_k$ for $k \neq j$, and*

$$x_{j,t} = \begin{cases} \underline{x}_j & \text{if } t = -1, \\ \overline{x}_j & \text{if } t = +1. \end{cases}$$

Also define

$$F_{\sim i}(X_{\sim j,t}) = (f_1(X_{\sim j,t}), \ldots, f_{i-1}(X_{\sim j,t}), f_{i+1}(X_{\sim j,t}), \ldots, f_n(X_{\sim j,t}),$$

and view $X_{\sim j,t}$ as a point in $\mathbb{R}^{n-1}$ by ignoring its j-th coordinate. Then $\mathrm{d}(F, \boldsymbol{X}, 0)$ is equal to the number of solutions to $F_{\sim i}(X_{\sim j,t}) = 0$ at which $\mathrm{sgn}(f_i) = s$ and such that $st(-1)^{i+j} F'_{\sim i}$ is positive, minus the number of such solutions at which $st(-1)^{i+j} F'_{\sim i}$ is negative, as j runs over all $j \in \{1, 2, \ldots, n\}$, and t takes on the values -1 and 1, where $F'_{\sim i}$, the $(n-1) \times (n-1)$ Jacobi matrix of $F_{\sim i}$ with respect to $(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)^T$, is assumed to be non-singular at such solutions.

In cases where F is merely continuous, we have

Theorem 1.25 *If $F : \boldsymbol{X} \to \mathbb{R}^n$ is continuous and $\mathrm{d}(F, \boldsymbol{X}, 0) \neq 0$, then there exists a solution of $F(X) = 0$ in $\boldsymbol{X}$.*

Even when F is such as Example 1.6, the degree can still be intuitively viewed, in terms of Definition 1.17, as the algebraic number of solutions, since there is a differentiable function $\tilde{F}$ with non-singular Jacobi matrix at the solutions X^* with $\tilde{F}(X^*) = 0$, such that $\mathrm{d}(\tilde{F}, \boldsymbol{X}, 0) = \mathrm{d}(F, \boldsymbol{X}, 0)$.

In Example 1.6, if $i = 2$ and $s = -1$, then $\mathrm{d}(\nabla\phi, \boldsymbol{X}, 0)$ can be computed by finding roots of $f_1 = 2x_1$ for which $f_2 = 2x_2 < 0$ on $\partial\boldsymbol{X}$. There is only one such root, occurring at the point $X_{\sim 2,-1} = (0, -2)$, at which $F'_{\sim i} = \frac{\partial f_1}{\partial x_1} = 2$. Thus, Theorem 1.24 states that $\mathrm{d}(F, \boldsymbol{X}, 0) = (-1)(-1)(-1)^{2+2} = 1$. Note that $\mathrm{d}(F, \boldsymbol{X}, 0)$ is the same as the degree of the gradient of the strictly convex function $\tilde{\phi}(X) = x_1^2 + x_2^2 - 1$.

The following theorem has motivated generalized bisection methods based on the topological degree.

Theorem 1.26 *Suppose F is continuous on $\mathbf{D}_1 \cup \mathbf{D}_2 \subset \mathbb{R}^n$, suppose $\mathbf{D}_1 \cap \mathbf{D}_2 \subset \partial\mathbf{D}_1 \cup \partial\mathbf{D}_2$, and suppose $F(X) \neq 0$ for $X \in \partial\mathbf{D}_1 \cup \partial\mathbf{D}_2$. Then* $\mathrm{d}(F, \mathbf{D}_1 \cup \mathbf{D}_2, 0) = \mathrm{d}(F, \mathbf{D}_1, 0) + \mathrm{d}(F, \mathbf{D}_2, 0)$.

Early machine computation of the topological degree proceeded from formulas for determinants [226], or from formulas involving sign changes of the components of F on $\partial\mathbf{D}$ [228, 105, 106]; corresponding algorithms contained heuristics that did not rigorously guarantee the correct degree. Generalized bisection methods were devised based on Theorem 1.26 and on successively dividing $\mathbf{D}$ into $\mathbf{D}_1$ and $\mathbf{D}_2$. More recently, Aberth [2] has proposed interval Newton methods to compute $\mathrm{d}(F, \boldsymbol{X}, 0)$, as indicated in Theorem 1.24; such methods must *rigorously* compute the topological degree.

The topological degree undoubtedly has its place in root-finding and global optimization algorithms such as those explained in Chapters 4 and 5 below, when there is singularity or degeneracy at solutions or critical points. However, computation of the degree requires searching all of the $2n$ faces of dimension $(n-1)$ of the box $\boldsymbol{X} \in \mathbb{IR}^n$, in contrast to a single search of $\boldsymbol{X}$ itself, when there are no singularities. Thus, a direct search of the n-dimensional space is probably preferable to computing the degree, in cases where there are no singularities.

2

SOFTWARE ENVIRONMENTS

This chapter deals with implementations of the interval arithmetic concepts described in Chapter 1. Such software is, in turn, used in later chapters to construct higher-level software systems to solve nonlinear systems of equations and for global optimization. A FORTRAN-77 library and a set of Fortran 90 modules built upon this library are described here. Although some information here is specific, underlying principles and philosophy are emphasized, and the packages are readily available, free of cost, to readers.

Alternate packages are also outlined.

2.1 INTLIB

The FORTRAN-77 package INTLIB is *ACM Transactions on Mathematical Software* Algorithm 737 [123], and is thus available over NETLIB [50, 68], either via FTP (ibid.) or over the World Wide Web[1]. INTLIB consists of standard FORTRAN-77 subroutines and functions for the elementary interval operations, some of the Fortran standard functions, and certain logical and utility operations. INTLIB achieves portability with simulated directed rounding, although minor modifications to INTLIB allow true directed rounding, where available.

Some simplification of INTLIB would be possible if it were rewritten with Fortran 90 constructs. However, INTLIB is presently directly accessible through Fortran 90, so its ease of use would not be affected.

[1]The address at the time of writing is `http://www.netlib.org/index.html`

Details of the construction, installation, and use of INTLIB that do not appear below can be found in [124], [123], and in the instructions provided with the FORTRAN-77 source.

2.1.1 Philosophy and Design Goals

The goals and design of INTLIB are consistent with the philosophy stated with the original Level-1 BLAS routines[2] [151]. That is to say, the routines are meant to be well-documented, generally available, production quality, and totally portable. Moreover, the routine names and argument lists are meant to provide a pattern upon which more efficient machine-specific versions can be based. With this scheme, users can transfer application programs from one machine to another, and the program, without modification, will execute efficiently on a variety of machines. An attempt was made in INTLIB to conform to a proposal of Corliss [37].

In addition, the routines in INTLIB are meant to be user-readable, thus providing instruction in the underlying methods and, possibly, allowing enough understanding for the user to improve them. The contents were selected to provide a core set of clearly needed routines, without large numbers of superfluous functions. When there was ambiguity of mathematical interpretation, such as branch points (leading to disconnected ranges, etc.), a routine was not included.

2.1.2 Simulated Directed Rounding

Conceptually simple, simulated directed rounding is a device with which rigorous enclosures to floating point results may be obtained in FORTRAN-77 and other languages that do not have access to true directed rounding, or on machines without hardware-supported directed rounding. The only requirement is that arithmetic parameters, such as the accuracy of an elementary operation, are known. For example, suppose $z = x + y$ is to be computed. Also suppose that the largest relative spacing ϵ_m between numbers[3], as well as the number

[2] although INTLIB is more like a "Level-0" BLAS, since INTLIB deals with scalar interval operations, rather than vector operations, as the Level-1 BLAS, or matrix-vector operations, as the Level-2 BLAS.

[3] For example, in a hypothetical decimal machine with numbers of the form $0.xxx \times 10^x$, the largest relative spacing between numbers is $0.101 \times 10^1 - 0.100 \times 10^1 = 10^{-2}$.

of units M in the last place by which the result of $+$, $-$, $\times$ or $\div$ can be in error, are known. Then the following procedure can be applied.

Algorithm 6 (Simplified example: addition with simulated directed rounding. See INTLIB routines `ADD` and `RNDOUT`.)

INPUT: $\boldsymbol{x}$ with $\underline{x} > 0$ and $\boldsymbol{y}$ with $\underline{y} > 0$. (The other cases are analogous; see the listing of `RNDOUT`.)

OUTPUT: $\boldsymbol{z}$, a machine interval containing the exact interval sum $\boldsymbol{x} + \boldsymbol{y}$.

1. *Compute* $\underline{z} = \underline{x} + \underline{y}$ *and* $\overline{z} = \overline{x} + \overline{y}$ *with the computer's floating point arithmetic.*
2. IF $\underline{z} \neq 0$ *and the following computations would neither underflow nor overflow*

   THEN

   (a) $\underline{z} \leftarrow (1 - \mathrm{M}\epsilon_m)\underline{z}$

   (b) $\overline{z} \leftarrow (1 + \mathrm{M}\epsilon_m)\overline{z}$

   ELSE

   Do a special computation for rigor.

   END IF

End Algorithm 6

Different formulas must be used when part or all of $\boldsymbol{x}$ or $\boldsymbol{y}$ is less than 0, and special consideration must be given for rigor when underflow or overflow would occur. In practice, Step 2 is performed after each elementary operation (i.e. after performing Step 1 with op replacing $+$, for op $\in \{+, -, \times, \div\}$). The INTLIB routine `RNDOUT` performs Step 2.

For details and clarification, the `FORTRAN-77` source for `RNDOUT` should be consulted directly. This routine is meant to be comprehensible by the user.

Within INTLIB, calls to `RNDOUT` may be replaced by similar calls to an assembler language routine, such as that described in [135]. This results in slightly

Name	function
ADD	interval addition: $z \leftarrow x + y$
CANCEL	cancellation subtraction as explained on page 5
IDIV	ordinary interval division $z \leftarrow x/y$, $0 \notin y$
MULT	interval multiplication $z \leftarrow xy$
RNDOUT	simulated directed rounding, as in §2.1.2
SCLADD	addition of an interval and a real number
SCLMLT	multiplication of an interval and a real number
SUB	interval subtraction $z \leftarrow x - y$

Table 2.1 Elementary arithmetic routines in INTLIB

narrower result intervals and slightly faster execution. In fact, this was done for Sun SPARC computers (six low-level routines in INTLIB were changed), with approximately a factor of two increase in speed[4].

2.1.3 Contents of INTLIB

INTLIB is organized into subsets of elementary interval arithmetic subroutines, interval standard function subroutines, utility routines, an error-printing routine, a routine to set mathematical constants and machine-dependent constants, and testing programs. Table 2.1 lists the arithmetic routines in INTLIB, while Table 2.2 lists the INTLIB implementations of standard functions.

INTLIB also contains utility functions, whose mathematical functions are elementary, but whose existence is justified by the error-checking they provide and their usefulness in operator overloading procedures. Table 2.3 lists these. Within this table, the arguments to a function are x and y for binary interval functions, x for unary interval functions, x and x for binary real / interval functions, and x and y for binary real / real functions. The results are z for interval-valued functions, `.TRUE.` / `.FALSE.` for logical-valued functions, and r for floating-point-valued functions. All intervals are considered to be closed.

Finally, INTLIB contains error-printing routines, a routine to initialize global constants and parameters, a suite of routines for testing, and sample test output. For details, see [123].

[4]Details are available from the author, or consult his home page on the World Wide Web at `ftp://interval.usl.edu/pub/interval_math/www/kearfott.html`.

Name	function
IACOS	interval arc cosine
IACOT	single argument arc cotangent
IASIN	interval arc sine
IATAN	single argument arc tangent
ICOS	interval cosine
IEXP	interval e^x
IIPOWR	nonnegative interval to an interval power
ILOG	interval natural logarithm
ISIN	interval sine
ISINH	interval hyperbolic sine
ISQRT	interval square root
POWER	integer power of an interval

Table 2.2 Standard function routines in INTLIB

Name	function
ICAP	$z \leftarrow x \cap y$ (Error signalled if empty)
IDISJ	.TRUE. if $x \cap y = \emptyset$
IHULL	$z \leftarrow [\min\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}]$
IILEI	.TRUE. if $x \subseteq y$
IILTI	.TRUE. if $\underline{x} > \underline{y}$ and $\overline{x} < \overline{y}$
IINF	$r \leftarrow \underline{x}$
IMID	$r \leftarrow (\underline{x} + \overline{x})/2$ (computation in floating point)
IMIG	"mignitude:" $r \leftarrow \min\{\lvert\underline{x}\rvert, \lvert\overline{x}\rvert\}$ if $0 \notin x$, and $r \leftarrow 0$ otherwise
INEG	negation with round out: $z \leftarrow -x$
INTABS	"magnitude:" $r \leftarrow \max\{\lvert\underline{x}\rvert, \lvert\overline{x}\rvert\}$
IRLEI	.TRUE. if $x \in x$
IRLTI	.TRUE. if $x > \underline{x}$ and $x < \overline{x}$
ISUP	$r \leftarrow \overline{x}$
IVL1	construct an interval from a point: $\underline{z} \leftarrow x$, $\overline{z} \leftarrow x$
IVL2	construct an interval from two points: $\underline{z} \leftarrow x$, $\overline{z} \leftarrow y$
IVLABS	range of $\lVert \circ \rVert$ over an interval: $z \leftarrow \{\lvert x\rvert, x \in x\}$
IVLI	assign one interval to another: $z \leftarrow x$
IWID	upwardly rounded width: $r \leftarrow \overline{x} - \underline{x}$

Table 2.3 Utility functions in INTLIB

2.1.4 An Example

Figure 2.1 contains a FORTRAN-77 routine that computes and prints rigorous bounds on the range[5] over $[1, 2]$ of the function $f(x) = x^4 + x^3 + x$ of Example 1.3 on page 37. The operations in this program are done in the order implied by Table 1.2; TMP2 and TMP3 represent the intermediate variables x_2 and x_3 in the first two computations. Note, however, that separate storage locations are not needed for x_4 and x_5, if all that is desired[6] is an interval enclosure $\boldsymbol{f}(\boldsymbol{x})$.

Figure 2.1 illustrates various aspects of INTLIB. Nonetheless, it is **not** recommended that INTLIB generally be used in this way, if a Fortran 90 compiler is available. Programming with an interval data type, as in §2.2.1 below, is, with a few exceptions, a more desirable way to program interval operations.

2.1.5 Exercises

1. Obtain and install INTLIB. Use INTLIB to evaluate the range (to within roundout error) of

$$\phi(x) = \left[\sin\left(\pi x^3\right)\right]^2$$

over the interval $x = [0.99, 1.01]$.

Recall that INTLIB may be obtained either through NETLIB at

`http://www.netlib.org/index.html`

or via anonymous FTP to

`interval.usl.edu`,

in the directory

`pub/interval_math/intlib`.

Hint: It is always necessary to call SIMINI *first. After* SIMINI *has been called, an interval enclosure for* π *is available as variable* PI *in the common block* MTHCNS.

2. Study the program SIMINI.

   (a) What assumptions are made here on the decimal to binary conversions of constants? How could you modify SIMINI to work with machine / compiler combinations for which these assumptions are not valid?

[5] Over $[1, 2]$ and with this particular function, the bounds are the actual range, to within roundout error.

[6] However, see §7 below.

```
C  This standard FORTRAN-77 routine uses INTLIB directly.
      PROGRAM TEST_INTLIB

C  Intervals are represented as double precision arrays
C  with two elements --
      DOUBLE PRECISION X(2), F(2)
      DOUBLE PRECISION TMP2(2), TMP3(2)

C Initialize machine constants and interval constants used in
C the standard functions --
      CALL SIMINI

C The range over [1,2] will be computed --
      X(1) = 1D0
      X(2) = 2D0

C Round out in case the decimal-to-binary conversion is
C not exact --
      CALL RNDOUT(X,.TRUE.,.TRUE.)

C Compute X**4 + X**3 + X --
      CALL POWER(X,4,TMP2)
      CALL POWER(X,3,TMP3)
      CALL ADD(TMP2,TMP3,TMP2)
      CALL ADD(TMP2,X,F)

      WRITE(6,*) F(1), F(2)

      END
```

Figure 2.1 A FORTRAN-77 program using INTLIB

(b) What assumptions are made on the word length of the machines upon which INTLIB is to be installed? What could you do to provide rigorous computation with INTLIB on machines with larger word lengths?

(c) What would you do to install INTLIB rigorously on Cray machines? *Hint: See the comments on Cray installation in [123].*

2.2 FORTRAN 90 INTERVAL AND CODE LIST SUPPORT

Throughout this section, it is assumed that the reader is familiar with Fortran 90. An excellent introduction, among others, is [22].

Several tools are available for conveniently accessing the routines in INTLIB through Fortran 90. These include an interval data type and a set of modules and functions for generating and interpreting code lists, as introduced in §1.4.5. Available freely, these will be called "`INTLIB_90`." A set of additional, higher-level tools to actually solve nonlinear equations and optimization problems, explained in §4.4 and §5.3 below, will be called "`INTOPT_90`."

2.2.1 An Interval Data Type

The interval data type is defined in a Fortran 90 module `INTERVAL_ARITHMETIC`. The module takes its support routines from INTLIB [123].

For example, `INTERVAL_ARITHMETIC` may be used for the same computation as in Figure 2.1; corresponding Fortran 90 code appears in Figure 2.2. The output to the program in Figure 2.2 should be similar to the output[7] to the program in Figure 2.1. However, the order in which the operations are done depends on how the particular compiler parses the statement

```
F = X**4 + X**3 + X,
```

and may not be the same as that in Table 1.2 or Figure 2.1.

The module `INTERVAL_ARITHMETIC` defines the four elementary operations (+, −, *, and /), as well as negation on interval data types (i.e. unary minus).

[7] namely, [2.9999999999999982, 26.0000000000000391] or something approximately equal, depending on the compiler and the machine

```
PROGRAM EVALUATE_EXAMPLE

!  This standard Fortran-90 routine
!  evaluates X**4 + X**3 + X over [1,2].

USE INTERVAL_ARITHMETIC
      TYPE(INTERVAL) X, F
      CALL SIMINI

      X = INTERVAL(1,2)
      F = X**4 + X**3 + X
      WRITE(6,*) F

END PROGRAM EVALUATE_EXAMPLE
```

Figure 2.2 Illustration of use of the interval data type

Mixed-mode operations are allowed only between intervals and double precision (or equivalent type), or between intervals and integer numbers[8]. Exponentiation ** is also defined for interval-to-integer, interval-to-interval, double precision-to-interval, and interval-to-double precision.

The module defines the generic names

ACOS, ACOT, ASIN, ATAN, COS, COT, EXP, LOG, SINH, SIN, SQRT, and TAN, (2.1)

each of which returns bounds on the range[9], to within roundout error, of the corresponding point-valued function. In each case, the corresponding INTLIB routine from Table 2.2 is used, with TAN and COT making use of the INTLIB routines ISIN and ICOS.

Additionally, the special interval functions in Table 2.4 are defined[10]. The definitions of ABS and MAG vary slightly from those in ACRITH-XSC [237] and

[8]This is because intervals are stored and rounded out as double precision numbers. Arithmetic between a real and interval would first involve converting the real to double precision, then rounding according to the double precision machine epsilon. However, the single-precision value may be only accurate to single precision, so the rounded interval would not contain the theoretical value. Rigor would thus be sacrificed.

[9]These bounds are not tight in the sense of being the smallest machine intervals that enclose the actual range. This is because, for portability, only double precision arithmetic is used in INTLIB. However, the bounds are rigorous inclusions of the range, and are reasonably tight.

[10]In Table 2.4, the Fortran 90 interval variables X, Y and Z are identified with intervals $x = [\underline{x}, \overline{x}]$, $y = [\underline{y}, \overline{y}]$ and $z = [\underline{z}, \overline{z}]$, while r is identified with the double precision variable R.

Syntax	Corresponding INTLIB routine	function
R = MAG(X)	INTABS	$r \leftarrow \max\{\lvert\underline{x}\rvert, \lvert\overline{x}\rvert\}$
R = WID(X)	IWID	$r \leftarrow \overline{x} - \underline{x}$
R = MID(X)	IMID	$r \leftarrow (\overline{x} - \underline{x})/2$
R = MIG(X)	IMIG	"mignitude:" $r \leftarrow \min\{\lvert\underline{x}\rvert, \lvert\overline{x}\rvert\}$ if $0 \notin \boldsymbol{x}$, and $r \leftarrow 0$ otherwise
Z = ABS(X)	IVLABS	$z \leftarrow \{\lvert x\rvert, x \in \boldsymbol{x}\}$
Z = MAX(X,Y)	–	$z \leftarrow [\max\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}]$ (cf. Chapter 6)

Table 2.4 Special interval functions in module INTERVAL_ARITHMETIC

other languages with interval data types: in [237], the function ABS corresponds to the INTERVAL_ARITHMETIC function MAG, and is consistent with use of $|\circ|$ throughout the literature on interval computations. However, when coding objective functions for interval branch and bound algorithms, it is more natural for ABS to return the range of $|\circ|$, as ABS does if INTERVAL_ARITHMETIC is used.

Finally, INTERVAL_ARITHMETIC defines the binary interval and logical-valued operators exhibited in Table 2.5. The binary operations in Table 2.5 that correspond to Fortran intrinsic operators on default data types (i.e. .LT., .GT., .LE., .GE., and .NE.) admit mixed mode operations between intervals and double precision or integers, while either or both arguments of .CH. may be double precision or integer.

With the exception of ABS and MAG, the operators and functions in the list (2.1) and Tables 2.4 and 2.5 act identically to those in ACRITH-XSC [237].

Assignment of interval values can be done using the default Fortran 90 assignment to structures, e.g. X = INTERVAL(.3D0,.3D0). However, this scheme is not recommended, since the values may not be properly rounded when converted from character strings to floating point numbers. The function IVL, which accepts either one or two double precision arguments, causes the internally-stored result to be rigorously rounded. Also, assignment (=) is overloaded in the module INTERVAL_ARITHMETIC, so that the result is properly rounded when an integer or double precision number is assigned to an interval. For example, X = IVL(0.3D0), X = IVL(0.3D0,0.3D0), and X=0.3D0 each[11]

[11] IVL also accepts integer arguments, such as X=IVL(3), X=IVL(3,3D0), or X=3, where X is an interval variable. However, many machines can store small integers exactly in floating

Syntax	Corresponding INTLIB routine	function
`Z = X.IS.Y`	ICAP	$z \leftarrow x \cap y$
`Z = X.CH.Y`	IHULL	$z \leftarrow [\min\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}]$
`X.SB.Y`	IILEI	`.TRUE.` if $x \subseteq y$
`X.SP.Y`	IILEI	`.TRUE.` if $x \supseteq y$
`X.DJ.Y`	IDISJ	`.TRUE.` if $x \cap y = \emptyset$
`R.IN.X`	IRLEI	`.TRUE.` if $r \in x$
`Y.LT.X`	–	`.TRUE.` if $\overline{y} < \underline{x}$
`Y.GT.X`	–	`.TRUE.` if $\underline{y} > \overline{x}$
`Y.LE.X`	–	`.TRUE.` if $\overline{y} \leq \underline{x}$
`Y.GE.X`	–	`.TRUE.` if $\underline{y} \geq \overline{x}$
`Y.NE.X`	–	`.TRUE.` if $y \neq x$ (set inequality)

Table 2.5 Logical and interval operators in module `INTERVAL_ARITHMETIC`

cause a properly rounded inclusion to the number 0.3 to be stored in the interval variable `X`.

The left and right endpoints of an interval `X` are double precision numbers accessed as `X%LOWER` and `X%UPPER`, respectively. These expressions can occur on either side of an assignment statement[12]. Additionally, `INF(X)` will return the lower bound on an interval `X`, and `SUP(X)` will return the upper bound[13] on an interval `X`. Implicit conversion from interval is not allowed: the only conversion from interval to floating point is through the functions `MID`, `INF`, and `SUP`.

An additional example, the program used to generate Table 1.3, appears in Figure 2.3.

point formats; for such machines, outward rounding is not necessary, and `X=INTERVAL(3,3)` would be more logical.

[12]The interval `X` is just a Fortran 90 sequenced structure with double precision components `X%LOWER` and `X%UPPER`. This is compatible with INTLIB, which stores intervals in `FORTRAN-77` as arrays with two elements.

[13]The routines `INF` and `SUP` were included since it is slightly easier with them than with `X%LOWER` and `X%UPPER` to make programs language-independent.

```
PROGRAM INTERVAL_NEWTON_ITERATION_1_D
   USE INTERVAL_ARITHMETIC
   IMPLICIT NONE

   DOUBLE PRECISION XP
   TYPE(INTERVAL) :: X, X_IMAGE
   INTEGER K
   DOUBLE PRECISION WIDTH_OF_X, OLD_WIDTH

   CALL SIMINI
      WIDTH_OF_X = 4
      X = IVL(1,2)

      DO K = 1,10000
         OLD_WIDTH = WIDTH_OF_X
         WIDTH_OF_X = IWID(X)
         XP = DBLE(X)
         WRITE(6,*) K, X, XP, WIDTH_OF_X, &
                    WIDTH_OF_X/OLD_WIDTH**2
         X_IMAGE = XP -  ( IVL(XP)**2 - IVL(4) ) / (2*X)
         IF(WIDTH_OF_X.LT.1D-11) EXIT
         X = X_IMAGE
      END DO

END PROGRAM INTERVAL_NEWTON_ITERATION_1_D
```

Figure 2.3 A univariate interval Newton method program, used to generate the data in Table 1.3

2.2.2 Generation of Code lists

Some details of an implementation of the operator overloading ideas sketched in §1.4.4 appear in this section.

A module `OVERLOAD` containing functions such as `CLADD` and `CLPOWN` of §1.4.4 defines the `CDLVAR` and `CDLLHS` data types described on page 45. This module produces the operation codes listed in Table 2.6 and Table 2.7 (next page). The set of operations in Table 2.6, excluding operation numbers 18 and 28, which have special meanings, and excluding operation numbers 31 and 32, is closed under repeated differentiation[14]. Operation 31 corresponds to user-defined univariate functions, while operation 32 corresponds to the derivatives of these functions. Similarly, each member of this same set of operations, and possibly also operations 31 and 32, can be formally inverted as either another operation in the set or as a short sequence of operations in the set. This is important for the techniques of Chapter 7.

The module `OVERLOAD` enables a user-supplied program to produce a code list from a representation of the function in Fortran 90 syntax. Figure 2.4 contains such a program that causes generation of a code list of the function $f(x) = x^4 + x^3 + x$ of Example 1.3 on page 37. This code list consists of a header line giving dimensions, followed by a sequence of lines specifying the individual operations. Although the exact sequence of operations depends on the particular compiler and version, a code list actually produced by the program in Figure 2.4 (page 85) appears[15] in Figure 2.5. Comparing the lines in Figure 2.5 to the meanings in Tables 2.6 and 1.1, it can be seen that the list in Figure 2.5 corresponds exactly to the sequence of operations in equations (1.30) on page 37.

The name and format of the file in which the code list is stored can be controlled. The default output file name, `CODELIST.OUT`, can be changed by including a statement of the form `OUTPUT_FILE_NAME='NAME.CDL'` before the statement `CALL INITIALIZE_CODELIST(X)`. This would cause the information as in Figure 2.5 to be written to the file `NAME.CDL` instead of `CODELIST.OUT`. The string `NAME` can be replaced by anything, but the suffix should be `CDL`, since programs, such as those in §2.2.3 and §2.2.4 assume file names of this form.

The numerical information corresponding to Figure 2.5 can be stored either in a binary or readable ASCII format. Control of this is through a configuration file

[14] Differentiation of the non-differentiable functions χ, $|\cdot|$, and max is explained in Chapter 6.

[15] The actual code list is formatted using Fortran output editing.

OP no.	operation	Comments
4.	$x_p = x_q x_r$	
5.	$x_p = x_q^2$	
6.	$x_p = x_q / x_r$	
7.	$x_p = x_q^{x_r}$	
8.	$x_p = \cos(x_q)$	
9.	$x_p = e^{x_q}$	
10.	$x_p = \log(x_q)$	
11.	$x_p = \sin(x_q)$	
12.	$x_p = \sqrt{x_q}$	
13.	$x_p = \arccos(x_q)$	
14.	$x_p = \arcsin(x_q)$	
16.	$x_p = a^{x_q}$	
17.	$x_p = x_q^a$	
18.	x_p is dependent	(objective or equation)
19.	$x_p = x_q^n$	(n stored in r)
20.	$x_p = x_q + x_r$	
21.	$x_p = x_q - x_r$	
22.	$x_p = a x_q$	
23.	$x_p = x_q + b$	
24.	$x_p = -x_q + b$	
25.	$x_p = a / x_q$	
26.	$x_p = a$	
27.	$x_p = \chi(x_s, x_q, x_r)$	Conditional branch
28.	x_p is dependent	(equality constraint)
29.	$x_p = -x_q$	
30.	$x_p = 0$	
31.	$x_p = \mathtt{U\_}(r, x_q, a, b)$	(user-defined; index stored in r)
32.	(derivative of U_)	(user-defined; index stored in r)
37.	$x_p = \lvert x_q \rvert$	
38.	$x_p = \max\{x_q, x_r\}$	
39.	$x_p = \min\{x_q, x_r\}$	

Table 2.6 Operations in module `OVERLOAD` and program `MAKE_GRADIENT`

OP no.	operation
33.	arctan
34.	cosh
35.	sinh
36.	tan

Table 2.7 Additional operations supported in module `OVERLOAD`

```
PROGRAM CODELIST_GENERATION
  USE OVERLOAD
    TYPE(CDLVAR), DIMENSION(1) ::  X
    TYPE(CDLLHS) ::  F
    CALL INITIALIZE_CODELIST(X)
    F = X(1)**4 + X(1)**3 + X(1)
    CALL FINISH_CODELIST
END PROGRAM CODELIST_GENERATION
```

Figure 2.4 Program that generates a code list for $f(x) = x^4 + x^3 + x$

OP no.	p	q	r
19	2	1	4
19	3	1	3
20	4	2	3
20	5	4	1
18	5	0	0

Figure 2.5 Operation lines of the code list produced by the program of Figure 2.4

```
NEQMAX, NROWMAX, NCONSTMX, BRANCHMX, BINARY_CODELIST
100     10000    2000      10        F
```

Figure 2.6 Sample `OVERLOAD.CFG` file

`OVERLOAD.CFG`, as in Figure 2.6. The first four numbers in this file are bounds that define maximum number of equations, maximum total number of rows, maximum number of constants that can be stored, and maximum number of conditional branches. The last entry specifies whether the code list is to be stored in an ASCII or binary format; if it is "F", then an ASCII code list is formed, and a binary code list is formed if it is "T".

Binary code lists are generally preferable except in example problems, for various reasons. The binary code lists can be retrieved more rapidly, and they take less space. Also, information of the constants a and b that are found in operations 16, 17, 22–26, and 31–32 is stored in indexed lines at the end of the file; conversion errors occur when such constants are stored and retrieved in an ASCII format[16], but do not occur when the constants are stored and retrieved in a binary format. Finally, the ASCII output formats are not wide enough for large code lists.

[16]The software takes account of these conversion errors by rounding out, assuming the errors are no larger than an error in an elementary operation. This, however, is not true with all machines and compilers.

```
PROGRAM PRODUCT
  USE OVERLOAD
  PARAMETER(NFACTORS=2000)
    TYPE(CDLVAR), DIMENSION(NFACTORS):: X
    TYPE(CDLLHS), DIMENSION(1):: F
    TYPE(CDLVAR) :: TMP
    OUTPUT_FILE_NAME='PRODUCT.CDL'
    CALL INITIALIZE_CODELIST(X)
    TMP = X(1)
    DO I = 2,NFACTORS
      TMP = TMP*X(I)
    END DO
    F(1) = TMP
    CALL FINISH_CODELIST
END PROGRAM PRODUCT
```

Figure 2.7 Accumulating a product for a dependent variable

Since operator overloading is used, the functions as illustrated in Figure 2.4 must be programmed with certain special syntax rules, as follows:

1. The input data variables and variables depending on these should be declared to be type CDLVAR ("CoDe List VARiable"). These variables may occur on both sides of assignment statements, and in arithmetic expressions obeying the rules of Fortran 90, containing "+", "−", "*", "/", "**", COS, EXP, LOG, SIN, SQRT, ACOS, ASIN, ATAN, COSH, SINH, TAN, ABS, MAX and U_. Mixed-mode operations with intervals, double precision, and integers are acceptable.

2. For multivariate systems of equations, variables defining the function components must be in a single vector F(1), ..., F(N), and these variables must be declared to be type CDLLHS. Each of them may only occur once in the program, on the left side of an assignment statement, while single objective functions may be handled as in Figure 2.4. If a dependent variable is a result of an accumulated sum, a temporary variable of type CDLVAR may be defined for the accumulation; see Figure 2.7.

3. For optimization problems, equality constraints of the form $g_i(X) = 0$ may be expressed by declaring an array G to be of type CDLINEQ, then assigning the expressions for the g_i to the elements of G, exactly as with variables of type CDLLHS.

```
PROGRAM REASSIGN_INDEPENDENT
  USE OVERLOAD
    TYPE(CDLVAR), DIMENSION(1):: X
    TYPE(CDLLHS), DIMENSION(1):: F
    OUTPUT_FILE_NAME='GUIDREAS.CDL'
    CALL INITIALIZE_CODELIST(X)
    DO I = 1,10
      X(1) = X(1)**2
    END DO
  F(1) = X(1)
    CALL FINISH_CODELIST
END PROGRAM REASSIGN_INDEPENDENT
```

Figure 2.8 Reassignment of a dependent variable

OP no.	p	q	r
5	2	1	0
5	3	2	0
5	4	3	0
5	5	4	0
5	6	5	0
5	7	6	0
5	8	7	0
5	9	8	0
5	10	9	0
5	11	10	0
18	11	0	0

Figure 2.9 The code list operation lines for the program in Figure 2.8

4. Independent variables must be in a single array[17] of type CDLVAR, and must be "initialized" by passing them as an argument to the subroutine INITIALIZE_CODELIST at the beginning of the program. The dimension of this array must equal exactly the number of independent variables. Examples occur in Figure 2.4 and Figure 2.7. The elements of this array may occur on either side of the assignment statement, but each such assignment results in a new intermediate variable; see the example in Figure 2.8, with output in Figure 2.9. (The program in Figure 2.8 defines $f_1 = x_1^{2^{10}}$.) Variables of type CDLVAR, other than the independent variables, may be defined and used.

5. Conditionally executed constructs (e.g. IF-THEN-ELSE or CASE statements) may not be used[18], unless the conditions tested are constant. However, most non-conditional loops and other non-conditional constructs following Fortran 90 syntax are allowable. Conditional branches may be programmed using the branch function χ described in Chapter 6 below.

6. Any variables declared as integer, double precision, or interval are treated as constants when forming the code list. Integers and double precision are rounded out to intervals. For rigor, such variables should be interval, or conversion should be used carefully. Single precision variables is not recommended, due to possible lack of rigor, while any variable that depends on the independent variables (as specified with INITIALIZE_CODELIST) should be declared to be of type CDLVAR.

[17] even if there is only one independent variable
[18] For an explanation of why, see [119].

7. The function may be defined through a hierarchy of subroutines, consistent with standard Fortran practice, as long as each subroutine contains an appropriate `USE OVERLOAD` statement at the beginning.

8. The last program statement must be a call to `FINISH_CODELIST`. This subroutine places the dependent variable identifications (operation code 18) at the end of the code list, then writes the code list to a file.

User-Defined Functions

Operation code 31 is reserved for user-defined unary operations. Such operations are useful both to reduce the size of the code list and to reduce overestimation in the ranges. For example, each component function f_i may be separable, i.e. f_i may be written in the form

$$f_i(X) = E_i(\phi_1(x_1), \ldots, \phi_n(x_n)), \tag{2.2}$$

where the exact range of each ϕ_j and its derivatives can be computed. In such an instance, the overestimation in f_i can be reduced by computing the exact range of each ϕ_j, rather than programming E_i using naive interval arithmetic. In an ideal case, $E_i(u_1, \ldots . u_n) = \sum_{j=1}^{n} u_j$ and the exact range of f_i and its gradient can be computed[19]. Furthermore, the code list for f_i can then be represented with a single operation for each entity ϕ_j.

Alternately, certain elementary or special functions not available within the basic system may be required. In such cases, it would be both more efficient and more effective (in computing sharp bounds on ranges) to provide a specific operation in the code list, rather than to program the function in terms of existing operations.

The system allows an arbitrary number of such unary operations, each associated with a unique index k. The set of operations is defined in module `USER_FUNCTIONS`, containing functions `USR`, `USRD`, `USRDD`, `USRP`, `USRPD`, `USRPDD`, `USRSL`, and `USRDSL`, with meanings as follows:

[19]Functions of this form are termed *separable*. Functions of the form $\sum_{j=1}^{n} u_j$ where the u_j depend on more than one, but not many of the variables x_i are termed *partially separable*; see [67].. Such partial separability has been exploited in the LANCELOT approximate optimization package [35].

`USR(K,INVERSE,X,A,B)` provides the range of the `K`-th function if `INVERSE=.FALSE.`, or the range of the inverse of the `K`-th function if `INVERSE=.TRUE.` over the interval `X` and with parameters `A` and `B`.

`USRD(K,X,A,B)` provides a range for the derivative of the `K`-th function over the interval `X` and with parameters `A` and `B`.

`USRDD(K,X,A,B)` provides a range for the second derivative of the `K`-th function over the interval `X` and with parameters `A` and `B`.

`USRP(K,RX,A,B)` provides a floating point value of the `K`-th function at `RX` and with parameters `A` and `V`.

`USRPD(K,RX,A,B)` provides a floating point value of the derivative of the `K`-th function at `RX` and with parameters `A` and `B`.

`USRPDD(K,RX,A,B)` provides a floating point value of the second derivative of the `K`-th function at `RX` and with parameters `A` and `B`.

`USRSL(K,XC,XR,A,B)` provides an interval enclosure for the slope of the `K`-th function over `XR` and centered at `XC`, with parameters `A` and `B`.

`USRDSL(K,XC,XR,A,B)` provides an interval enclosure for the slope of the derivative of the `K`-th function over `XR` and centered at `XC`, with parameters `A` and `B`.

New functions are defined by creating additional branches in `CASE` statements in each of the above routines. Not all functions are required for all applications. For example, the second derivative functions and `USRDSL` are used in second-order optimization methods, with code lists that have been symbolically differentiated with the techniques described below; these functions may not be necessary for root-finding algorithms. Similarly, depending on the application, it may not be necessary to provide a branch for the inverse of a function.

Example 2.1 *Suppose that a single code list operation is to be defined for the general quadratic* $f(x) = x^2 + ax + b$. *This function corresponds to the branch* `K=1` *in figures 2.10a and 2.10b; routines for* `USRDD`, `USRP`, `USRPD`, `USRPDD`, `USRSL`, *and* `USRDSL` *are similar, and are available with* `INTLIB_90`[20]. *The function* $f_1(x) = x^2 + 3x$ *can then be described with a single code list operation, generated by the program in Figure 2.11.*

[20]Similarly, the routines `QUADRATIC_RANGE`, `QUADRATIC_INVERSE`, and `QUADRATIC_SLOPE` are slightly lengthier, but are supplied in the template module `USER_FUNCTIONS` in `INTLIB_90`.

```
FUNCTION USR(K,INVERSE,X,A,B) RESULT(R)
   USE INTERVAL_ARITHMETIC
   INTEGER, INTENT(IN) :: K, INVERSE
   TYPE(INTERVAL), INTENT(IN) :: X, A, B
   TYPE(INTERVAL) :: R
   SELECT CASE(K)
      CASE(1)
         IF(INVERSE.EQ.0) THEN ! Compute exact range of x^2 + ax + b
            R = QUADRATIC_RANGE(1D0,DBLE(A),DBLE(B),X)
         ELSE ! Compute the inverse -- not required in all cases
            R = QUADRATIC_INVERSE(1D0,DBLE(A),DBLE(B),X)
         END IF
      CASE DEFAULT
         WRITE(*,*) 'ERROR IN FUNCTION "USR.",& '
                    'FUNCTION NUMBER K =', K
         WRITE(*,*) 'IS UNIMPLEMENTED.'
         STOP
   END SELECT
END FUNCTION USR
```

Figure 2.10a Function USR for Example 2.1

```
FUNCTION USRD(K,X,A,B) RESULT(R)
   USE INTERVAL_ARITHMETIC
   INTEGER, INTENT(IN) :: K
   TYPE(INTERVAL), INTENT(IN) :: X
   TYPE(INTERVAL), INTENT(IN) :: A, B
   TYPE(INTERVAL) :: R
   SELECT CASE(K)
      CASE(1)
       ! Compute the exact derivative range of x^2 + ax + b.
         R = 2D0*X + A
      CASE DEFAULT
         WRITE(*,*) 'ERROR IN FUNCTION "USRD.", &
                    'FUNCTION NUMBER K =', K
         WRITE(*,*) 'IS UNIMPLEMENTED.'
         STOP

   END SELECT
END FUNCTION USRD
```

Figure 2.10b Function USRD for Example 2.1

```
!  This tests the user-defined function by computing x**2 + 3x +2
PROGRAM USER_FUNCTION_TEST
  USE OVERLOAD
      TYPE(CDLVAR), DIMENSION(1) :: X
      TYPE(CDLLHS), DIMENSION(1) :: F_1
      OUTPUT_FILE_NAME='USRTST.CDL'
      CALL INITIALIZE_CODELIST(X)
      F_1(1) = U_(1,X(1),IVL(3),IVL(2)) - 2
      CALL FINISH_CODELIST
END PROGRAM USER_FUNCTION_TEST
```

Figure 2.11 A program to generate a code list for $f_1(x) = f(x) - 2$, where $f(x)$ is as in Example 2.1

2.2.3 Differentiation of Code Lists

A code list generated according to the principles and procedures of §1.4.4 and §2.2 is a *symbolic* representation of a particular sequence of operations for evaluating the function. Not only can such code lists be interpreted, as described in §1.4.5 and §2.2.4, but they can also be symbolically manipulated. In fact, as first pointed out in [187, p. 14], code lists can be symbolically differentiated in a loop similar to that of Figure 1.5: To process each intermediate variable of the code list, an additional code list line (or lines), corresponding to the derivative of that intermediate variable is created. In such a scheme, a *derivative code list* with exactly the same format and set of operations as the original code list can be created[21]. For example, a line corresponding to $x_p = x_q x_r$ generates lines corresponding to the formula $x'_p = x_q x'_r + x'_q x_r$. Here x'_p, x'_q and x'_r are new intermediate variables if x_p, x_q and x_r were intermediate variables[22]; generally, x'_q and x'_r have been generated prior to encountering the code list line corresponding to x_p, so x'_p can be defined.

Although the actual implementation of such a differentiation scheme is somewhat lengthy, it is based on the above simple principle. It is embodied in module GRADIENT_CODELIST of INTLIB_90. For example, suppose the code list for $f(x) = x^4 + x^3 + x$, as in Figure 2.5 as been generated and placed in the file EX1.CDL; then the program COMPUTE_GRADIENT of Figure 2.12 produces the code list for the derivative of $f(x)$, as in Figure 2.13. Comparing the derivative

[21] The idea of differentiating code lists appeared, perhaps first, in [187, p. 14 ff].

[22] x'_q and x'_r are either 1 or 0 if x_q and x_r were independent variables.

```
PROGRAM COMPUTE_GRADIENT
   USE GRADIENT_CODELIST
      INPUT_FILE_NAME='EX1.CDL'
      CALL INPUT_FUNCTION_CODELIST
      CALL OBJECTIVE_TO_GRADIENT
      CALL FINISH_GRADIENT
END PROGRAM COMPUTE_GRADIENT
```

Figure 2.12 Generation of a derivative code list corresponding to the code list in file `EX1.CDL`

OP no.	p	q	r	a	b
19	2	1	4	–	–
19	3	1	3	–	–
20	4	2	3	–	–
20	5	4	1	–	–
19	6	1	3	–	–
22	7	6	0	4	–
19	8	1	2	–	–
22	9	8	0	3	–
20	10	7	9	–	–
23	11	10	0	–	1
18	5	0	0	–	–
18	11	0	0	–	–

Figure 2.13 Operation lines of the code list for the derivative of $f(x) = x^4 + x^3 + x$

$$
\begin{array}{rcl}
x_2 & = & x_1^4 \\
x_3 & = & x_1^3 \\
x_4 & = & x_2 + x_3 \\
x_5 & = & x_4 + x_1 \\
x_6 & = & x_1^3 \\
x_7 & = & 4x_6 \\
x_8 & = & x_1^2 \\
x_9 & = & 3x_8 \\
x_{10} & = & x_7 + x_9 \\
x_{11} & = & x_{10} + 1 \\
\hline
f_1 & = & x_5 \\
f_2 & = & x_{11}
\end{array}
$$

Figure 2.14 Algebraic interpretation of the code list in Figure 2.13

code list in Figure 2.13 and the operation set in Table 2.6 gives the algebraic interpretation in Figure 2.14.

Produced by a program in `INTLIB_90`, such derivative code lists have exactly the same format as the original code list, and can be interpreted or further differentiated as such. However, the meaning attached to dependent variables (corresponding to operation code 18, at the end of the code list) differs, depending on the way the code list is viewed, as follows.

Example 2.2 *Let* $F : \mathbb{R}^2 \to \mathbb{R}^2$ *be defined*[23] *by*

$$
\begin{aligned}
f_1(x_1, x_2) &= 4x_1^3 - 3x_1 - x_2 \\
f_2(x_1, x_2) &= x_1^2 - x_2.
\end{aligned}
$$

[23]This function appeared as a test problem in [171], then in [108] and subsequent work.

OP no.	p	q	r	a	b
19	3	1	3	–	–
22	4	3	0	4	–
22	5	1	0	3	–
21	6	4	5	–	–
21	7	6	2	–	–
5	8	1	0	–	–
21	9	8	2	–	–
18	7	0	0	–	–
18	9	0	0	–	–

Figure 2.15 A code list for the function in Example 2.2

OP no.	p	q	r	a	b
19	3	1	3	–	–
22	4	3	0	4	–
22	5	1	0	3	–
21	6	4	5	–	–
21	7	6	2	–	–
5	8	1	0	–	–
21	9	8	2	–	–
19	10	1	2	–	–
22	11	10	0	3	–
22	12	11	0	4	–
26	13	0	0	3	–
21	14	12	13	–	–
22	15	1	0	2	–
26	16	0	0	−1	–
26	17	0	0	−1	–
18	7	0	0	–	–
18	14	0	0	–	–
18	16	0	0	–	–
18	9	0	0	–	–
18	15	0	0	–	–
18	17	0	0	–	–

Figure 2.16 The derivative code list corresponding to Figure 2.15

The code list produced by INTLIB_90 *is as in Figure 2.15, while the corresponding derivative code list appears in Figure 2.16.*

Observe that there are two dependent variables in Figure 2.15 and six dependent variables in Figure 2.16. The two dependent variables in Figure 2.15 are viewed as f_1 and f_2. If the code list in Figure 2.16 were viewed as an original (i.e. non-differentiated) code list, then there would be six dependent variables $\tilde{f}_1$, $\tilde{f}_2$, $\tilde{f}_3$, $\tilde{f}_4$, $\tilde{f}_5$ and $\tilde{f}_6$. Viewed as a first derivative code list, the six dependent variables are given the meanings f_1, $\frac{\partial f_1}{\partial x_1}$, $\frac{\partial f_1}{\partial x_2}$, f_2, $\frac{\partial f_2}{\partial x_1}$, and $\frac{\partial f_2}{\partial x_2}$, in that order. If the code list of Figure 2.16, viewed as an original code list, is differentiated again, then a code list for the second derivative tensor results. In this second derivative code list, there would be 18 dependent variables, corresponding to

$$f_1, \quad \frac{\partial f_1}{\partial x_1}, \quad \frac{\partial f_1}{\partial x_2}, \quad \frac{\partial f_1}{\partial x_1}, \quad \frac{\partial^2 f_1}{\partial {x_1}^2}, \quad \frac{\partial^2 f_1}{\partial x_2 \partial x_1}, \quad \frac{\partial f_1}{\partial x_2}, \quad \frac{\partial^2 f_1}{\partial x_2 \partial x_1}, \quad \frac{\partial^2 f_1}{\partial {x_2}^2},$$
$$f_2, \quad \frac{\partial f_2}{\partial x_1}, \quad \frac{\partial f_2}{\partial x_2}, \quad \frac{\partial f_2}{\partial x_1}, \quad \frac{\partial^2 f_2}{\partial {x_1}^2}, \quad \frac{\partial^2 f_2}{\partial x_2 \partial x_1}, \quad \frac{\partial f_2}{\partial x_2}, \quad \frac{\partial^2 f_2}{\partial x_2 \partial x_1}, \quad \frac{\partial^2 f_2}{\partial {x_2}^2},$$

in that order. (Note the redundancy, assuming continuity of the partial derivatives.)

There are alternatives to symbolic differentiation of code lists. In particular, Frechét arithmetic [65, Rall, pp. 17–24], similar to the forward mode of automatic differentiation as in §1.4.1, can be used for derivative tensors of any order. Also, a first derivative tensor can be obtained from the code list for the original function with implementations of the reverse mode of §1.4.2. In the nonlinear systems solver in INTOPT_90 (Chapter 4 below), only the original code lists are used: Jacobi matrices are obtained with reverse automatic differentiation. Similarly, in the optimization code in INTOPT_90 (Chapter 5 below), a first derivative, or gradient code list is created, and Hessian matrices are obtained with the reverse mode. In optimization problems, the original code list has only one dependent variable, the objective function, not counting dependent variables corresponding to constraints (associated with operation number 28 of Table 2.6).

The redundancy in the dependent variables in code lists for higher-order derivative tensors can be eliminated [179], at the expense of generality of the code list format and some simplicity of implementation and use. Also, it has been reported[24] that use of second derivative code lists in the general format can be efficient relative to first derivative code lists and automatic differentiation for the Hessian matrices.

Also notice some redundancy in the operations corresponding to intermediate variables in Figure 2.16. Care is taken to avoid some of this redundancy during generation of the individual code list lines corresponding to derivatives[25], while other redundancy could be reduced with post-processing. The types of post-processing that may be necessary are possibly similar to techniques appropriate for eliminating redundant arithmetical computations in optimizing programming language compilers. It is known that at most five times the storage for the original code list should be required for the code list for the gradient, regardless of the number of independent variables in the original function [92]. It is unknown how close the implementation of GRADIENT_CODELIST is to this complexity bound, but the derivative code list, not counting lines di-

[24] in a private conversation with Claire Adjiman
[25] See the actual module GRADIENT_CODELIST.

rectly related to the dependent variables, is within this bound on many specific problems[26].

Although the process of producing a derivative code list can be viewed as symbolic, it is not symbolic differentiation in the sense commonly associated with symbolic manipulation packages such as Maple [229, 80], Mathematica [244, 80], MACSYMA [189, 80] or Reduce [156, 80]. In such packages, alternate schemes are used to process arithmetic expressions.

2.2.4 Interpretation of Code Lists – Generic Functions

Routines within `INTLIB_90` that use code lists to compute floating point and interval function values and derivatives, according to the automatic differentiation principles of §1.4 (page 36) and the generic function ideas of §1.4.5 (page 47), are outlined here.

Once the code list is created[27], input routines supplied with the package enter it into global storage space in the module `CODELIST_VARIABLES`. The code list can then be used by a number of package-supplied generic functions. Some of these operate on generic code lists, while others deal with gradient code lists that are assumed to have been produced by differentiating a code list corresponding to a single objective function.

Generic Code List Routines

The following routines operate on any code list, although interpretation of the results differs, depending on whether the code list is the result of differentiation of another code list. Interval routines are:

`SUBROUTINE F(X,FVAL)` returns a set of bounds on the ranges of the function components over the box represented in the interval vector `X(:)` in the interval vector `FVAL`. For example, if the code list presently in memory corresponded to Figure 2.15, i.e. to Example 2.2, then, upon return, `FVAL(1)` would contain the bounds on the range of $4x_1^3 - 3x_1 - x_2$ over the box represented by the array `(X(1),X(2))` and `FVAL(2)` would con-

[26] even though, at the time of writing, no post-processing is done in `INTLIB_90`.

[27] e.g. by running a program such as that in Figure 2.4

tain the range of $x_1^2 - x_2$ over that box. The natural interval extension corresponding to the code list is used.

SUBROUTINE F_2(X,FVAL) returns a set of bounds on the ranges F(:) over the box X(:), as SUBROUTINE F(X,FVAL), except that a mean value extension is used.

SUBROUTINE DENSE_JACOBI_MATRIX(X,FP) returns a bound on the range of $\frac{\partial f_i}{\partial x_j}$ over the box X(:) in the interval array component FP(i,j), for each i and j. These bounds are computed using the reverse mode of automatic differentiation of §1.4.2.

SUBROUTINE DENSE_SLOPE_MATRIX(XC,XR,S) returns a slope bound $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ in the two-dimensional interval array S, where $\boldsymbol{X}$ corresponds to the interval vector XR and $\check{\boldsymbol{X}}$ corresponds to the interval vector XC. Hansen's technique of §1.3.2, the slope arithmetic of §1.4.3, and a modified[28] differencing technique as in Theorem 1.13 are used.

Floating point routines are:

SUBROUTINE F_POINT(RX,RFVAL) returns a floating point approximation to the function components at the point represented in the floating point vector RX in the floating point vector RFVAL.

SUBROUTINE POINT_JACOBI_MATRIX(RX,RFP) returns a floating point approximation to $\frac{\partial f_i}{\partial x_j}$ at the point represented in the floating point vector RX(:) in the interval array component RFP(i,j), for each i and j, computed using the reverse mode of automatic differentiation of §1.4.2.

For example, suppose the code list of Figure 2.15, corresponding to the function in Example 2.2, is stored in the file EX2.CDL. Then the program:

```
PROGRAM EXAMPLE_INTERVAL_EVALUATION
   USE CODELIST_FUNCTIONS
   IMPLICIT NONE
      TYPE (INTERVAL), DIMENSION(:), ALLOCATABLE :: X, FVAL
      CODELIST_FILE_NAME = 'EX2.CDL'
      CALL GET_CODELIST
   ALLOCATE ( X(NSMALL),  FVAL(NEQ) )
```

[28] modified to avoid instances of cancellation error, as in Formula 1.35

```
      X(1) = IVL(1,2); X(2) = IVL(3,4)
      WRITE(6,*) 'Initial box:'
      WRITE(6,'(2(1X,2(1X,D12.4)))') X
      WRITE(6,*)
      WRITE(6,*) 'INTERVAL VALUE F(X) USING THE NATURAL ', &
                 'INTERVAL EXTENSION:'
      CALL F(X,FVAL)
      WRITE(6,'(2(1X,2(1X,D12.4)))') FVAL
   DEALLOCATE ( X, FVAL )
END PROGRAM EXAMPLE_INTERVAL_EVALUATION
```

produces the output:

```
 Initial box:
    0.1000D+01    0.2000D+01     0.3000D+01    0.4000D+01

 INTERVAL VALUE F(X) USING THE NATURAL INTERVAL EXTENSION:
   -0.6000D+01    0.2600D+02    -0.3000D+01    0.1000D+01
```

Indeed, $4[1,2]^3-3[1,2]-[3,4] = [-6,26]$, and $[1,2]^2-[3,4] = [-3,1]$. Additional details of use are distributed with the package itself.

The asymptotic computational cost of these routines is summarized in Table 2.8. There, n is the number of independent variables, N_{EQ} is the number of dependent variables, and N_{OPS} is the total number of operations required to evaluate the function, i.e. the number of operation lines in the code list file. For example, in code list of Figure 2.15, corresponding to Example 2.2, $n = 2$, $N_{\mathrm{EQ}} = 2$, and $N_{\mathrm{OPS}} = 7$.

The actual cost of computing floating point and interval function values with these generic routines depends on various factors. In a particular comparison, on a Sun SPARC 20 with a floating point accelerator and the NAG Fortran 90 compiler, floating point evaluation of the Shekel function [49], F_POINT was 8 times slower than programming that specific function in floating point, and F was roughly 16 times slower [114]. The overhead in the floating point computations lies in storage and retrieval of intermediate quantities and the looping and case statements in F_POINT. However, that overhead is not significant for the interval computations, since programming the function directly with the interval data type of §2.2.1 is not significantly faster. The advantage of the generic functions is that only *one* function need be programmed, as in Figure 2.4, thus

Procedure name or task	Number of operations
Form the code list	$\mathcal{O}(N_{\text{OPS}})$
Form a derivative code list	$\mathcal{O}(n(N_{\text{EQ}} + N_{\text{OPS}}))$
F	$\mathcal{O}(N_{\text{OPS}}) + \mathcal{O}(N_{\text{EQ}})$
F_2	$\mathcal{O}(N_{\text{OPS}}) + \mathcal{O}(N_{\text{EQ}}) + \mathcal{O}(N_{\text{EQ}}N_{\text{OPS}})$
DENSE_JACOBI_MATRIX	$\mathcal{O}(N_{\text{OPS}} + N_{\text{EQ}}) + \mathcal{O}(N_{\text{EQ}}N_{\text{OPS}})$
DENSE_SLOPE_MATRIX	$\mathcal{O}(nN_{\text{OPS}}) + \mathcal{O}(N_{\text{OPS}})$
F_POINT	$\mathcal{O}(n + N_{\text{EQ}} + N_{\text{OPS}})$
POINT_JACOBI_MATRIX	$\mathcal{O}(N_{\text{OPS}} + N_{\text{EQ}}) + \mathcal{O}(N_{\text{EQ}}N_{\text{OPS}})$

Table 2.8 Operational complexity – generic routines

minimizing programming effort. This advantage makes the overhead acceptable for many computations.

All of these routines can be applied to code lists that have been produced from the differentiation process of §2.2.3, but have a different interpretation. For example, the output of the program EXAMPLE_INTERVAL_EVALUATION, when the file EX2.CDL contains the derivative code list of Figure 2.16 is:

```
Initial box:
   0.1000D+01   0.2000D+01    0.3000D+01   0.4000D+01

INTERVAL VALUE F(X) USING THE NATURAL INTERVAL EXTENSION:
  -0.6000D+01   0.2600D+02    0.9000D+01   0.4500D+02
  -0.1000D+01  -0.1000D+01   -0.3000D+01   0.1000D+01
   0.2000D+01   0.4000D+01   -0.1000D+01  -0.1000D+01
```

Here, the meaning of F(X) is as explained on page 93. That is, upon output, the array FVAL contains $\boldsymbol{f}_1([1,2],[3,4]) = [-6,26]$, $\frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}_1}([1,2],[3,4]) = [9,45]$, $\frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}_2}([1,2],[3,4]) = -1$, $\boldsymbol{f}_2([1,2],[3,4]) = [-3,1]$, $\frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{x}_1}([1,2],[3,4]) = [2,4]$, $\frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{x}_2}([1,2],[3,4]) = -1$, in that order. Similarly, if EX2.CDL contained the code list corresponding to the second derivative tensor, then the array FVAL would contain components of the function, Jacobi matrix, and second derivative tensor in the order explained on page 93.

Gradient Code List Routines

These routines make it easier to extract particular information, such as the objective function values only or the gradient component values only, from first derivative code lists. Some of them are:

`SUBROUTINE F_GRADIENT_D(X,FVAL)` is similar to `SUBROUTINE F`, but returns only bounds on the objective function, even though it works with the gradient code list corresponding to that objective function.

`SUBROUTINE F_2G(X,FVAL)` as `SUBROUTINE F_2`, returns only bounds on the objective function computed a second-order extension, but works with a gradient code list.

`SUBROUTINE GRAD(X,GVAL)` is similar to `F`, but works with a gradient code list, and returns only the components of the gradient.

`SUBROUTINE F_POINT_GRADIENT(RX,RFVAL)` is similar to `F_POINT`, but returns a floating point approximation to only the function value while working with the code list for the gradient of the function.

`SUBROUTINE POINT_HESSIAN_G(RX,RFP)` is similar to `POINT_JACOBI_MATRIX`, but returns a floating point approximation to the Hessian matrix of the original function while working with the code list for the gradient of the function.

`SUBROUTINE CONSTRAINTS_D(X,CVAL)` returns, in the interval array `CVAL`, interval bounds for the constraint residuals over the box represented by the interval array `X`; see Example 2.3.

`SUBROUTINE CONSTRAINT_GRADIENTS_D(X,GMAT)` returns interval bounds on the components of the Jacobi matrix of the constraints in the two-dimensional interval array `GMAT`.

`SUBROUTINE POINT_CONSTRAINTS(RX,RCVAL)` returns floating point approximations to the constraint residuals at the floating point vector `RX`, in the array `RCVAL`.

`POINT_CONSTRAINT_GRADIENTS(RX,CGMAT)` returns floating point approximations to the Jacobi matrix of the constraints at the floating point vector `RX`, in the two-dimensional floating point array `CGMAT`.

```
PROGRAM CNSEX
  USE OVERLOAD
      TYPE(CDLVAR), DIMENSION(2) :: X
      TYPE(CDLLHS), DIMENSION(1) :: PHI
      TYPE(CDLINEQ), DIMENSION(1) :: G
      OUTPUT_FILE_NAME='CNSEX.CDL'
      CALL INITIALIZE_CODELIST(X)
  PHI(1) = X(1)**2 + X(2)**2
    G(1) = X(1) + X(2) - 1
      CALL FINISH_CODELIST
END PROGRAM CNSEX
```

Figure 2.17 A program to generate a code list for $\phi(x_1, x_2) = x_1^2 + x_2^2$, subject to $x_1 + x_2 - 1 = 0$

```
PROGRAM EXAMPLE_CONSTRAINT_EVALUATION
   USE CODELIST_FUNCTIONS; IMPLICIT NONE
      TYPE (INTERVAL), DIMENSION(:), ALLOCATABLE :: X, CVAL
      CODELIST_FILE_NAME = 'CNSEXG.CDL'
      CALL GET_CODELIST
   ALLOCATE ( X(NSMALL),  CVAL(NINEQ/(NSMALL+1)) )
      X(1) = IVL(1,2); X(2) = IVL(3,4)
      WRITE(6,*) 'CONSTRAINT RESIDUALS:'
      CALL CONSTRAINTS_D(X,CVAL)
      WRITE(6,'(2(1X,2(1X,D12.4)))') CVAL
   DEALLOCATE ( X, CVAL)
END PROGRAM EXAMPLE_CONSTRAINT_EVALUATION
```

Figure 2.18 Evaluation of the constraints in Example 2.3

Example 2.3 *Consider optimization of* $\phi(x_1, x_2) = x_1^2 + x_2^2$ *subject to the constraint* $x_1 + x_2 - 1 = 0$. *The program in Figure 2.17 will place a code list for the function and constraint in the file* CNSEX.CDL. *Suppose that this code list is differentiated and placed in* CNSEXG.CDL. *Then the program in Figure 2.18 will produce the output:*

```
CONSTRAINT RESIDUALS:
   0.3000D+01    0.5000D+01
```

In addition to the above routines, there are various routines to support computation of the Fritz John equations, etc., for the computations described in Chapter 5.

Additional Package Capabilities

Additional, more specialized routines are also part of the package. Some of these routines are lower-level routines that nonetheless can be useful in making specific computations more efficient. For example, a function and the components of its gradient may share many common subexpressions in a gradient code list. Thus, it is more efficient to evaluate the intermediate variables in the gradient code list only once, to obtain both functions and gradients. A routine `FORWARD_SUBSTITUTION` evaluates all of the intermediate and dependent variables, then specialized routines quickly extract specific information (objective function values or gradient components) by simply looking up values that have already been computed. Details are distributed with `INTLIB_90`.

Other routines, available with the package but not documented here, set up the function, residuals, and Jacobi matrix for the expanded system of equations in which each intermediate variable is considered to be a dependent variable, as described in Chapter 7. These also include routines for the automatic and user-defined substitution-iteration process of Chapter 7. Routines are also included to compute the reduced gradient and reduced Hessian matrix described in §5.2.3.

Finally, availability of the code list allows easy translation to any format. For example, relatively simple prototypical routines that output the function as a TEX representation or as a `FORTRAN-77` program work with `INTLIB_90`. However, considerable effort is required to polish such routines into professional quality software.

2.2.5 Exercises

1. Program Algorithm 5 on page 54 using the interval data type and the supplied functions for code list interpretation. Use either fixed-size arrays for the lists (actually stacks) $\mathcal{L}$ and $\mathcal{C}$, or the `INTERVAL_LIST` data type available with `INTLIB_90`. Use your program to attempt to find all solutions to the system $f(x) = \sin(x) - .125x = 0$ within the interval $[-3\pi, 3\pi]$.

2. Use the program from Problem 1 to attempt to find all eighteen real solutions within the interval $[-12, 8]$ of the polynomial system

$$f(x) = \sum_{k=0}^{18} a_k x^k = 0,$$

where:

$$
\begin{array}{llll}
a_0 = -0.3719362500 \times 10^3 & a_1 = -0.7912465656 \times 10^3 \\
a_2 = 0.4044944143 \times 10^4 & a_3 = 0.9781375167 \times 10^3 \\
a_4 = -0.1654789280 \times 10^5 & a_5 = 0.2214072827 \times 10^5 \\
a_6 = -0.9326549359 \times 10^4 & a_7 = -0.3518536872 \times 10^4 \\
a_8 = 0.4782532296 \times 10^4 & a_9 = -0.1281479440 \times 10^4 \\
a_{10} = -0.2834435875 \times 10^3 & a_{11} = 0.2026270915 \times 10^3 \\
a_{12} = -0.1617913459 \times 10^2 & a_{13} = -0.8883039020 \times 10^1 \\
a_{14} = 0.1575580173 \times 10^1 & a_{15} = 0.1245990848 \times 10^0 \\
a_{16} = -0.3589148622 \times 10^{-1} & a_{17} = -0.1951095576 \times 10^{-3} \\
a_{18} = 0.2274682229 \times 10^{-3}
\end{array}
$$

What behavior do you observe? Is the behavior affected by the choice of ϵ? What explanation do you have for the behavior? (Hint: This problem is "Gritton's second problem," first introduced to the author by the authors of [14], and discussed in [114].)

2.3 OTHER SOFTWARE ENVIRONMENTS

The main goals of INTLIB_90 are to provide a reasonably efficient, portable access to its capabilities, to provide a unifying framework and, perhaps, standardization of terminology, names, and features, for interval arithmetic support, and to supply the most common needs in global optimization. Various other programming languages and packages excel generally and for specific purposes. Here, some of these other packages are listed, along with perceived strengths and weaknesses and availability. The list is not comprehensive; in particular, many researchers have created packages for their own use, but have not widely publicized them.

Interval arithmetic packages can generally be classified as follows.

Early packages: Most were Fortran or Algol-based, and were constructed on the precompiler principle.

SC-languages: Including ACRITH-XSC (also known as FORTRAN-SC) and Pascal-SC, these extensions of common languages feature an accurate dot product and a large and useful set of operators and functions.

XSC-languages: Created on the "SC" model, these are portable languages based mainly on the operator overloading principle.

Other modern packages: These include INTLIB_90 and various packages in C++, and other languages that support operator overloading.

Packages for symbolic manipulators: These include libraries for interval arithmetic in Mathematica and Maple.

Capabilities in spreadsheets: These include an extensions to Excel and the product UniCalc.

Logic programming languages: Prolog, with its ability to define relationships without specifying dependent and independent variables, is suited for some types of interval computations.

Miscellaneous packages: These are certain specialized packages, and many packages written for the authors' own work.

Interval packages contain attributes other than these. For example, the package can support variable precision arithmetic, it can have a maximally accurate dot product, etc.

2.3.1 Early Packages

A set of Algol-60 procedures for the standard functions was presented by Herzberger in 1970 [82]. Later, Triplex-Algol appeared [11, 33], with a data type consisting of an interval combined with a floating point approximation.

A widely available package was Yohe's set of "reasonably portable" FORTRAN-66 interval arithmetic routines [247], combined with Augment precompiler [42]. Ahead of its time, Augment featured operator overloading and other modern constructs that it translated to FORTRAN-66. Yohe's routines were designed to be transportable by building on of several low-level machine-specific routines that contained the directed rounding; assembler versions of these routines were provided for several machines, such as IBM 360 mainframes and Univac machines. This package may still be available, although it is unsupported, and modern alternatives exist.

Jeter and Shriver described the portable preprocessor SEPAFOR (target language FORTRAN IV), for a variable precision interval arithmetic [100]. Such variable precision is useful e.g. in solving systems of differential equations, where the precision can be adjusted to take account of sensitivity to initial conditions, etc. The precompiler was based on a modification of Brent's multiple

precision package [24] and the preprocessor RATFOR. ACM Algorithm 524, Brent's package is still available. Also, see [13], a modern Fortran 90 multiple precision package. A convenient Fortran 90 portable multiple precision interval arithmetic package could be created by extending the package [13].

The University of Minnesota's M77 compiler, a FORTRAN-77 compiler, had an extensive interval arithmetic library[29] by W. R. Walster, and efficiently supported an interval data type intrinsically within the compiler[30] [230]. Appendix Z of [230] contains a description of the capabilities of Walster's package. The M77 compiler ran on CDC mainframes. Hansen and Walster used it successfully in much of their research, e.g. [235].

Bendzulla presents a two-pass precompiler for an interval data type in FORTRAN IV [18]. S. Markov and his co-workers also created a Fortran package. Early software at the Mathematics Research Center of the University of Wisconsin is mentioned in [187, Ch. 5]. An example interval arithmetic package in PL/I is given in [15].

2.3.2 The "SC" Languages

FORTRAN-SC, known commercially as ACRITH-XSC [236, 160, 237], and Pascal-SC [188] made interval capabilities and user-defined data types widely accessible. Available first for CP/M-based microcomputers, and then for microcomputers running DOS, Pascal-SC is cited heavily in the interval computations literature; numerous packages, such as implementation of the hierarchy of higher operations[31] [145, 147] and packages for standard numerical tasks, such as linear system solvers, have been written in Pascal-SC.

Based on FORTRAN-77, ACRITH-XSC is a compiler for IBM 370 mainframes that makes use of the ACRITH [206, 95] subroutine package, with improvements.

Both ACRITH-XSC and Pascal-SC feature a maximally accurate dot product, intrinsic interval data types, dynamically allocated arrays, and user-defined data types, with operator overloading. The results of the elementary interval operations are the best possible, i.e. they are the smallest possible machine

[29] By coincidence, Walster's library happens to be called INTLIB; it does not have a direct relationship to the later ACM Algorithm 737.

[30] According to informal reports, interval arithmetic was only five times slower than floating point.

[31] e.g. arithmetic on vectors and matrices

Task	CPURAT
$\sum_{i=1}^{10^6}$ 1.0D0	39.7
$\prod_{i=1}^{10^6}$ 1.0D0	34.6
Compute sin(1) 10^6 times	17.7

Table 2.9 Ratio of interval to floating point CPU times CPURAT for ACRITH-XSC on an IBM 3090

Task	CPURAT
$\sum_{i=1}^{10^6}$ 1.0D0	19.7
$\prod_{i=1}^{10^6}$ 1.0D0	24.1
Compute sin(1) 10^6 times	104.9
Compute 1^2 10^6 times	9.6

Table 2.10 Ratio of interval to floating point CPU times CPURAT for INTERVAL_ARITHMETIC on a Sun SPARC 20 model 51

intervals that contain the mathematical result. Additionally, the standard function library in ACRITH is in IBM 360/370 assembler language, and has been crafted for both tightness of bounds and speed with techniques described in [23, 138].

Table 2.9 hints[32] that the efficiency of interval arithmetic in ACRITH-XSC is not excessive *vis à vis* floating point arithmetic, especially for the standard functions. A similar table for the module INTERVAL_ARITHMETIC from INTLIB_90 appears[33] in Table 2.10. Discounting differences in the machines, this is favorable to ACRITH-XSC, especially to the design of the standard function routines, since INTERVAL_ARITHMETIC gives fairly tight bounds, but not the smallest possible.

A disadvantage of Pascal-SC and ACRITH-XSC is that they are available on a limited variety of computers.

2.3.3 The "XSC" Languages

The advent of modern programming languages such as Ada, C++ and Fortran 90 facilitated portability. A second generation of languages, the XSC languages, modelled on Pascal-SC and ACRITH-XSC, is portable. Developed under the leadership of Prof. Dr. Kulisch at Karlsruhe University, these languages all feature a maximally accurate dot product and interval operations

[32]This table first appeared in [130].
[33]This data first appeared in [222].

that are nearly the best possible, in the sense described on page 104. A good discussion of these properties appears in [243].

Pascal-XSC [71] is essentially a portable compiler that adheres to the specifications of Pascal-SC. The compiler itself is written in C. A "numerical toolbox," that is, a well-documented package with routines[34] for evaluation of polynomials, automatic differentiation, roots of single equations, accurate, verified solution of linear systems of equations, nonlinear systems of equations and global optimization, etc. is available [70] in Pascal-XSC. Krämer [139] has written a set of modules for variable precision floating point and interval arithmetic in Pascal-XSC. The compiler itself is commercial[35]: licenses are for executable versions for specific machines and C++ compilers.

C-XSC [134] is an extension to the programming language C that consists of a set of C++ class libraries. In addition to explanation of the language itself, the book [134] contains short descriptions of programs for important numerical computations, similar to the computations in the Pascal-XSC toolbox [70]. A version of C-XSC for the Borland C++ compiler version 4 is available free of charge via FTP from the University of Karlsruhe[36]. C-XSC is one of a number of implementations of interval arithmetic based on C++; see §2.3.4 below.

Fortran-XSC [238] is still under development. As with `INTLIB_90`, it will be a set of Fortran 90 modules. However, emphasis in the design of Fortran-XSC is on an accurate dot product and on modules for accurate arithmetic, as well as interval vector and matrix operations. Separate sets of modules are planned for 2 ULP and 1 ULP accuracy, under the assumption that the machine floating point arithmetic is accurate to within one ULP.

2.3.4 Other Modern Packages

Several other interval packages, besides C-XSC, have appeared in C++, perhaps since C++ was one of the first widely available languages with user-defined data types and operator overloading. Jeff Ely [53] has such a variable precision interval arithmetic package that has been ported to several machines and vectorized on a Cray YMP. It is not publicly available at the time of writing.

[34] of varying sophistication and application

[35] from Numerik Software GmbH, P. O. Box 2232, D76492 Baden-Baden, Germany

[36] at `iamk4515.mathematik.uni-karlsruhe.de` in the directory `pub/cxsc/borland` at the time of writing

Anthony Leclerc has a C++ package [152] that he has ported to several types of workstations, and has used to solve difficult global optimization problems in parallel on a distributed network of workstations. It is also not publicly available at the time of writing.

Knüppel [135] developed a package in C / C++ following a philosophy somewhat similar to that of INTLIB / `INTERVAL_ARITHMETIC`, and has made it available via anonymous FTP[37]. The basic routines are termed BIAS, while the portion defining the higher-level functions is PROFIL. The package is portable, except for a single small assembler language routine that is called to change rounding modes; versions of this routine are available for Sun, IBM PC compatible (80x86-based) and HP workstations. On these machines, a single operation is sufficient to change rounding modes, for support of true directed rounding. Following the BLAS paradigm, Knüppel *et al.* devise efficient libraries for vector and matrix operations and numerical linear algebra. For example, in an interval vector addition, all of the sums of the lower end points of the components should be rounded down, and the sums of the upper end points all should be rounded up; thus only two changes of rounding modes are needed. With this pipelining of interval operations, certain interval matrix computations approach the theoretical maximum possible speed, taking roughly only twice the time of corresponding floating point computations. The author has created a version of INTLIB with a modification of Knüppel's assembler language routine for Sun SPARC machines; it resulted in a speed up of roughly 2 over the portable version of INTLIB, and in slightly tighter interval enclosures.

Jüllig has created a C++ package BIBINS [101] for interval computations that includes modules for vector and matrix arithmetic. Hyvönen and De Pascale [88] have created a C++ interval package InC++.

An Ada package `GENERIC_SCIENTIFIC_COMPUTATION` has been developed collaboratively with European Community support [136]; an emphasis in this package is on an accurate dot product, matrix, and vector operations, while an interval standard function library appears to be absent. George Corliss also has an Ada package for interval arithmetic and Taylor arithmetic [40] that is available from him.

Modula-SC [57], a Modula-2 extension, is a precompiler, written in Modula-2 and translating to Modula-2. Modula-SC provides an accurate dot product, as well as the operations in higher-order (vector and matrix) spaces of Kulisch *et al.* [145, 147].

[37] at the address `ti3sun.ti3.tu-harburg.de` at the time of writing

2.3.5 Packages for Symbolic Manipulators

Both interval and symbolic computations use computers to obtain mathematically rigorous statements. Also, since interval computations benefit from *a priori* rearrangement and simplification of expressions, and since algorithms such as interval Newton methods require evaluation of functions and derivatives with different data types, it is natural to consider interval computations within a symbolic computation environment.

Interval packages are available both within the Mathematica [132] and Maple [36] systems[38]. These packages are useful for various interactive explorations, and as interval calculators. For example, the following is a portion of a Mathematica session[39].

```
(Input):      f[x_] := x(x-1)
(Input):      fp[t_] = D[f[t],t]
(Output):                -1 + 2 t
(Input):      ix = Interval[0,1]
(Output):                Interval[0,1]
(Input):      f[ix]
(Output):                Interval[-1, 0]
(Input):      fp[ix]
(Output):                Interval[-1, 1]
(Input):      f[1]
(Output):                0
(Input):      fp[1]
(Output):                1
(Input):      Sin[ix]
(Output):                Interval[0, Sin[1]]
```

These packages may be useful for illustration and teaching of concepts in nonlinear systems and global optimization, such as in [5].

The Maple package [36] contains capabilities consistent with Corliss' BIAS proposal[40] [37]. An advantage of the Maple package is that the source code is available.

[38] A general reference for Mathematica is [244], while a general reference for Maple is [229].
[39] with format edited for presentation here
[40] Corliss' BIAS proposal has the same name as the Jansson / Knüppel package of [135], but is something different.

In their present state symbolic manipulation packages should be used with some care, since internal simplifications in the algebraic expressions may result in interval values that do not correspond to exhibited algebraic expressions. that is, the packages may rearrange expressions without telling the user, and such rearrangements give different interval extensions. Furthermore, *programs* that include loops, etc. written in these languages execute thousands of times more slowly than corresponding programs in languages such as Fortran 90.

2.3.6 Capabilities in Spreadsheets

Both spreadsheets and logic programming languages have been used in a technique termed *constraint satisfaction* in [89], *constraint logic programming* in [232], *subdefinite calculations* in [12], and *substitution-iteration* in [112], [222], and in Chapter 7 below. In a nutshell, such techniques consider equations such as $x_2 = x_1^3$ not as defining a *direction* of computation, but a *relation*; thus, if x_2 is known to lie in a certain interval, then it can be used to narrow the possible range of values for x_1, just as interval bounds for x_1 can narrow the range of values for x_2. This can be used for solving nonlinear systems of equations, proving infeasibility of inequality constraints, etc.

A spreadsheet enhancement, built upon Microsoft Excel, that supports such interval constraint propagation has been constructed by Hyvönen *et al.* [89] as an add-on function library, based on their C++ package InC++ [88].

A group at the Russian Institute of Artificial Intelligence has developed the notepad-like program UniCalc [12], in which objective functions, constraints of various types, etc. are entered into a screen area, and calculation proceeds in an order similar to that of an interval constraint propagation spreadsheet. A difference is that, in UniCalc, the expressions are completely parsed into component operations, as described with the expanded system on page 39 and in Chapter 7. The package, with a nice user interface, runs on DOS systems[41], and is available commercially. An advantage is that the user can stop the calculations in mid-stream and change certain values, for interactive explorations. The computations in UniCalc are rapid compared to other implementations of such techniques, but a disadvantage of an early version of UniCalc was lack of rigor in the arithmetic[42].

[41] and possibly on other systems now

[42] That is, directed roundings were not used, so that it was possible for a mathematically incorrect (though approximately correct) result to be given.

2.3.7 Logic Programming Languages

Logic programming languages such as Prolog are well suited for constraint processing [231, 19], as mentioned above. In [30], these techniques are briefly reviewed, and are used to advantage in a modification of the Moore–Skelboe algorithm (see §5.1 below) for unconstrained optimization.

Van Hentenryck *et al.* [232] re-examine the general theory of constraint logic programming, incorporating their considerations in an implementation of a general nonlinear system solver, then solving sizable problems arising from various applications.

The package `INTLIB_90` contains *ad-hoc* Fortran 90 routines for constraint propagation; see Chapter 7. Logic programming languages offer more flexibility, but basic operations may execute more slowly. Conclusions such as those in [30] depend on the relative speeds with which certain operations are executed, and on other aspects of the overall algorithms.

2.3.8 Miscellaneous Packages

Here, two packages with special capabilities, and a lesser-known `FORTRAN-77` package are described.

Range Arithmetic and Precision Basic

Aberth [1]has researched *range arithmetic* for some years. In this variable precision version of interval arithmetic, intervals are represented with a single mantissa that contains a variable number of words, combined with a *range*. For example, suppose that a word consisted of two decimal digits. Then the interval $[3.14150, 3.14160]$ would be represented as:

$$+\boxed{3.1}\boxed{41}\boxed{55} \quad \pm \quad \boxed{.05} \times 10^{-3}.$$

Operational definitions of this arithmetic, analogous to formulas (1.5)–(1.9) on page 3, are available for such representations [1, pp. 14–15]. Such arithmetic, implemented well, can make efficient use of both storage and time in problems requiring high precision.

Aberth developed the language PRECISION BASIC to support range arithmetic, and presented a complete numerical analysis text[43] [1], accessible to undergraduates, based on range arithmetic. A diskette with PRECISION BASIC, for IBM PC compatible computers, is distributed with the text.

Aberth and Schaefer also provide C++ modules[44] for range arithmetic [3]. Using these modules, Aberth has developed an algorithm for numerical computation of the Brouwer degree, as well as an associated algorithm for solving nonlinear systems of equations [2].

It is the author's opinion that multiple precision and variable precision arithmetic, except in certain cases, are more crucial in the verified solution of differential equations than in solution of nonlinear systems and global optimization. However, efficient environments that provide variable precision are advantageous for general scientific computation.

An Interval Calculator; An Optimization Input Format

Arnold Neumaier[45] and Friedhelm Heizmann have developed a set of `FORTRAN-77` routines[46] for interval arithmetic, along with an interactive interpreter INTCOM that parses expressions and evaluates them with either floating point or interval arithmetic. The interval arithmetic uses only the machine's floating point arithmetic, without outward rounding. The philosophy behind this is the following view: In global optimization and certain other computations, speed is useful, and approximate bounds on ranges are useful even if they are not rigorous[47].

[43] with chapters on the arithmetic, solvable and singular problems, a single nonlinear equation, rational arithmetic, zeros of polynomials, numerical linear algebra, numerical differentiation and integration, ordinary differential equations, and a PRECISION BASIC manual

[44] available via anonymous FTP at `math.tamu.edu` at the time of writing

[45] private communication

[46] The set of `FORTRAN-77` routines is called BIAS, Basic Interval Arithmetic Subroutines, the same as Knüppel's package [135] and as George Corliss' general proposal [37].

[47] The author of the present work experimented some time ago with non-rigorous interval arithmetic [108]. Misleading results were sometimes obtained, especially when transporting from one machine to another. These results were harder to trace and interpret than when rigorous arithmetic was used. The author's present opinion is that rigorous computations are usually worth the moderate extra cost. However, it is possible, through introduction of tolerances, to take into account in global optimization algorithms bounds that are only approximate.

Neumaier has also proposed a format for representing global optimization problems [176]. This format permits description of underlying structure that interval methods can use.

Another FORTRAN-77 *Package*

Krischchuk *et al.* present a preprocessor [143] for their INFFOR language, both written in FORTRAN-77 and translating to FORTRAN-77, along with an associated support library. The system is an outgrowth of an earlier system that had a preprocessor in PL/I and a support library in assembler language. This software has supported the authors' work in modelling manufacturing processes in radio-electronics. The package has been compiled with Microsoft Fortran version 5.0.

3

ON PRECONDITIONING

As mentioned in §1.2, it is usually necessary to precondition an interval linear system to obtain meaningful bounds on the solution set. This is clear for the interval Gauss–Seidel method, since it reduces to the classical Gauss–Seidel method when the coefficients of the system of equations are thin, and since the classical Gauss–Seidel method in general does not converge without preconditioning. However, preconditioning is also necessary and useful for the Krawczyk method and interval Gaussian elimination. For example, if interval Gaussian elimination, i.e. Algorithm 1 is applied to the system $\boldsymbol{AX} = \boldsymbol{B}$ of (1.19) on page 20, the result is

$$\left(\begin{array}{c|c|c} [\ 2,4] & [-2,1] & [-2,2] \\ \hline [-1,2] & [\ 2,4] & [-2,2] \end{array}\right)$$
$$\sim \left(\begin{array}{c|c|c} [\ 2,4] & [-2,1] & [-2,2] \\ \hline 0 & [2,4] - \frac{[-2,1]}{[2,4]}[2,4] & [-2,2] - \frac{[-2,2]}{[2,4]}[-2,2] \end{array}\right)$$
$$= \left(\begin{array}{c|c|c} [\ 2,4] & [-2,1] & [-2,2] \\ \hline [-1,2] & [-4,4] & [-6,6] \end{array}\right),$$

which gives only $\boldsymbol{x}_2 \in \frac{[-6,6]}{[-4,4]} = \mathbb{R}$ and $\boldsymbol{x}_1 \in \mathbb{R}$, whereas, as is seen in Figure 1.1, the smallest possible interval enclosure of the solution set is $\Sigma(\boldsymbol{A}, \boldsymbol{B}) = ([-4,4],[-4,4])^T$. Similarly, applying the interval Gauss–Seidel method results in $\tilde{\boldsymbol{x}}_1 \leftarrow \frac{[-2,2]-[-2,1][-10,10]}{[2,4]} = [-11,11] \supset [-10,10]$, and $\tilde{\boldsymbol{x}}_2 \leftarrow \frac{[-2,2]-[-1,2][-10,10]}{[2,4]} = [-11,11] \supset [-10,10]$. Finally, the Krawczyk method only makes sense in the context of preconditioners.

In contrast, if the system is preconditioned by the inverse midpoint matrix as in exercises 1, 2 and 3 of §1.2.3, the resulting augmented system and interval

Gaussian elimination process, rounded out to four digits, becomes

$$\begin{pmatrix} 0.3243 & 0.0541 \\ -0.0541 & 0.3243 \end{pmatrix} \left(\begin{array}{c|c|c} [\ 2,4] & [-2,1] & [-2,2] \\ \hline [-1,2] & [\ 2,4] & [-2,2] \end{array}\right)$$
$$= \left(\begin{array}{c|c|c} [0.5945,1.4055] & [-0.5406,0.5406] & [-0.7568,0.7568] \\ \hline [-0.5406,0.5406] & [0.5945,1.4055] & [-0.5406,0.5406] \end{array}\right)$$
$$\sim \left(\begin{array}{c|c|c} [0.5945,1.4055] & [-0.5406,0.5406] & [-0.7568,0.7568] \\ \hline 0 & [0.1031,1.8969] & [-1.1659,1.1659] \end{array}\right),$$

giving the large but finite solution vector

$$\boldsymbol{X} = ([-11.5430, 11.5430], [-11.5430, 11.5430])^T.$$

For this problem, the interval Gauss–Seidel method does not reduce the coordinate widths with initial guess $([-10,10],[-10,10])$, when the above preconditioner is used.

As a second example, take the system

$$\begin{pmatrix} [1.8,2.2] & [3.9,4.1] \\ [3.8,4.2] & [4.9,5.1] \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} [5.1,6.9] \\ [7.8,10.2] \end{pmatrix}. \tag{3.1}$$

Interval Gaussian elimination with no preconditioner gives

$$\boldsymbol{X} = ([-8.7201, 5.7695], [-0.8501, 5.0723])^T,$$

whereas interval Gaussian elimination with the inverse midpoint preconditioner gives

$$\boldsymbol{X} = ([-1.4679, 4.3251], [-0.8501, 3.1833])^T.$$

With initial guess vector $\boldsymbol{X} = ([-12,12],[-12,12])^T$ and no preconditioner, the interval Gauss–Seidel method gives

$$\boldsymbol{X} = ([-12, 12], [-12, 8.6939])^T,$$

while the interval Gauss–Seidel method with the inverse midpoint preconditioner gives

$$\boldsymbol{X} = ([-6.2143, 3.3572], [-3.6032, 1.3810])^T.$$

Thus, despite the fact that the actual solution set of the preconditioned system may be larger than the solution set of the original system, preconditioning is effective and necessary.

The inverse midpoint preconditioner, described in §3.1, is most commonly seen in the literature and in practice. Under various conditions, particularly when the widths of some, but not all, of the coefficients of the system are large and when the system arises from an interval Newton method, alternate preconditioners are useful. Classification, implementation, and advice concerning these optimal preconditioners appears in §3.2.

3.1 THE INVERSE MIDPOINT PRECONDITIONER

Let $\boldsymbol{AX} = \boldsymbol{B}$ be an n by n interval linear system as in §1.2. The inverse midpoint preconditioner is denoted by

$$Y^{\text{mid}} = \text{m}(\boldsymbol{A})^{-1}.$$

In many cases, the linear system will have the structure of systems arising from interval Newton methods, as in §1.5.

3.1.1 Shortcuts with the Arithmetic

Much of the computation related to preconditioning and bounding the solution sets of the preconditioned systems can be simplified[1] when the preconditioner is Y^{mid}. In particular, a large portion of these computations can be done in floating point arithmetic without the need to change rounding modes, provided a "round down" or a "round up" mode is available[2]. These shortcuts depend upon properties of *symmetric intervals.*

Definition 3.1 *An interval $\boldsymbol{s} \in \mathbb{IR}$ is said to be symmetric about 0, or simply* symmetric, *provided $\boldsymbol{s} = [-s, s]$ for some $s \geq 0$. A vector or matrix will be called* symmetric about 0 *provided each of its components or coefficients is symmetric.*

The following is not hard to prove. (See Exercise 2.)

Lemma 3.1 *Suppose $\boldsymbol{s}_1 \in \mathbb{IR}$ and $\boldsymbol{s}_2 \in \mathbb{IR}$ are symmetric, and let $x, y \in \mathbb{R}$ and interval $\boldsymbol{x} \in \mathbb{IR}$ be arbitrary. Then*

$$\begin{array}{ll} \boldsymbol{s}_1 + \boldsymbol{s}_2 & \textit{is symmetric;} \\ \boldsymbol{s}_1 \boldsymbol{x} & \textit{is symmetric;} \\ \boldsymbol{s}_1 / \boldsymbol{x} & \textit{is symmetric.} \end{array}$$

(In the division, $\boldsymbol{s}_1 / \boldsymbol{x} = \mathbb{R}$ in Kahan–Novoa–Ratz arithmetic when $0 \in \boldsymbol{x}$.)

[1] This has been observed by Rump in [208], etc.

[2] Such modes are available in IEEE standard arithmetic.

Now write

$$\boldsymbol{A} = \mathrm{m}(\boldsymbol{A}) + \boldsymbol{\Delta}, \tag{3.2}$$

so that $\boldsymbol{\Delta} = [-\Delta, \Delta]$ is symmetric about 0. Taking Lemma 3.1 and properties of interval arithmetic into account, it follows that

$$Y^{\mathrm{mid}}\boldsymbol{A} = Y^{\mathrm{mid}}(\mathrm{m}(\boldsymbol{A}) + \boldsymbol{\Delta}) = I + \boldsymbol{S}, \tag{3.3}$$

where $I \in \mathbb{R}^{n\times n}$ is the identity matrix and $\boldsymbol{S}$ is symmetric about 0. (See Exercise 3.) Furthermore,

$$\boldsymbol{S} = Y^{\mathrm{mid}}\boldsymbol{\Delta} = [-|Y^{\mathrm{mid}}|\Delta, |Y^{\mathrm{mid}}|\Delta]. \tag{3.4}$$

Thus, computation of $Y^{\mathrm{mid}}\boldsymbol{\Delta}$ can be done by taking the absolute value of the components of Y, then computing $|Y|\Delta$ with floating point arithmetic and the rounding mode set to "round up." This is considerably faster than computing $Y^{\mathrm{mid}}\boldsymbol{\Delta}$ in most current implementations of interval arithmetic.

A Caution and Disclaimer

The arithmetic shortcuts presented here are practical only under special circumstances: Formulas (3.3) and (3.4) are valid only if an *exact* value of Y^{mid} is used; otherwise, $\boldsymbol{S}$ may not be exactly symmetric, and terms corresponding to roundoff error must be added for rigorous existence, non-existence or uniqueness verification, and for obtaining rigorous bounds on the solutions. In particular, suppose $Y = \tilde{A}^{-1}$. Then

$$\begin{aligned} Y\boldsymbol{A} &= \tilde{A}^{-1}\left(\tilde{A} + (\mathrm{m}(A) - \tilde{A}) + \boldsymbol{\Delta}\right) \\ &= I + \tilde{A}^{-1}(\mathrm{m}(A) - \tilde{A}) + \boldsymbol{S}, \end{aligned}$$

so $Y\boldsymbol{A}$ should first be expanded so that it is symmetric about the identity matrix. If such computations take less effort than the effort saved by assuming symmetry[3], then the formulas presented here can be used.

Shortcuts with the Krawczyk Operator

In addition to choosing $Y = Y^{\mathrm{mid}}$, the base point will be chosen to be $\check{X} = \mathrm{m}(\boldsymbol{X})$, so that $\boldsymbol{X} - \check{X} = [-|\boldsymbol{X} - \check{X}|, |\boldsymbol{X} - \check{X}|]$ is symmetric about 0. Written

[3]One example of a method to take account of the errors in Y^{mid} appears in [76, §7, p. 1501].

in terms of an interval Newton method, the Krawczyk method then becomes

$$\begin{aligned}\mathbf{K}(\boldsymbol{X},\check{X}) &= \check{X} - Y^{\mathrm{mid}}F(\check{X}) + (I - Y\boldsymbol{A}(\boldsymbol{X}))(\boldsymbol{X}-\check{X}) \\ &= \check{X} - Y^{\mathrm{mid}}F(\check{X}) + Y^{\mathrm{mid}}\boldsymbol{\Delta}[-|\boldsymbol{X}-\check{X}|,|\boldsymbol{X}-\check{X}|] \\ &= \check{X} - Y^{\mathrm{mid}}F(\check{X}) + |Y^{\mathrm{mid}}|\Delta\frac{\mathrm{w}(\boldsymbol{X})}{2}[-1,1]\end{aligned} \tag{3.5}$$

Thus, $\mathbf{K}(\boldsymbol{X},\check{X})$ may be computed mainly with floating point operations, using upward rounding to compute $|Y^{\mathrm{mid}}|\Delta\mathrm{w}(\boldsymbol{X})/2$. It should be noted, however, that, for systems arising from interval Newton methods, roundoff error in computing $F(\check{X})$ should be taken into account for rigor. The easiest way to do this is to evaluate $F(\check{X})$ using interval arithmetic and to use interval arithmetic to compute $Y^{\mathrm{mid}}F(\check{X})$.

These ideas are expounded in Rump [208].

Shortcuts with the Interval Gauss–Seidel Method

The interval Gauss–Seidel method is given by Equation (1.44) on page 59. The shortcuts are similar to those in the Krawczyk method. Again, the base point will be chosen as $\check{X} = \mathrm{m}(\boldsymbol{X})$. As in Equation (1.44), the symbol $Y_i^{\mathrm{mid}}\boldsymbol{A}_j$ will denote the dot product of the i-th row of Y^{mid} with the j-th column of $\boldsymbol{A}$ and let Δ_j denote the j-th column of Δ. Also set $Y^{\mathrm{mid}}\boldsymbol{\Delta} = \boldsymbol{S} = \{\boldsymbol{s}_{i,j}\}_{i,j=1}^{n}$. Then the sums in the numerator of Equation (1.44) become

$$\begin{aligned}&\textstyle\sum_{j=1}^{i-1} Y_i^{\mathrm{mid}}\boldsymbol{A}_j(\tilde{\boldsymbol{x}}_j - \check{x}_j) + \sum_{j=i+1}^{n} Y_i^{\mathrm{mid}}\boldsymbol{A}_j(\boldsymbol{x}_j - \check{x}_j) \\ &\quad= \textstyle\sum_{j=1}^{i-1} \boldsymbol{s}_{i,j}(\tilde{\boldsymbol{x}}_j - \check{x}_j) + \sum_{j=i+1}^{n} \boldsymbol{s}_{i,j}(\boldsymbol{x}_j - \check{x}_j) \\ &\quad= \textstyle\sum_{j=1}^{i-1} |Y_i^{\mathrm{mid}}|\Delta_j\frac{\mathrm{w}(\tilde{\boldsymbol{x}}_j)}{2}[-1,1] + \sum_{j=i+1}^{n} |Y_i^{\mathrm{mid}}|\Delta_j\frac{\mathrm{w}(\boldsymbol{x}_j)}{2}[-1,1]\end{aligned} \tag{3.6}$$

Similarly, the denominator in Equation (1.44) becomes

$$1 + |Y_i^{\mathrm{mid}}|\Delta_i[-1,1].$$

Thus, except for the interval division, the interval Gauss–Seidel step can be computed with only upwardly-rounded floating point arithmetic. Furthermore, when

$$|Y_i^{\mathrm{mid}}F(\check{\boldsymbol{X}})| \le \sum_{j=1}^{i-1} |Y_i^{\mathrm{mid}}|\Delta_j\frac{\mathrm{w}(\tilde{\boldsymbol{x}}_j)}{2} + \sum_{j=i+1}^{n} |Y_i^{\mathrm{mid}}|\Delta_j\frac{\mathrm{w}(\boldsymbol{x}_j)}{2},$$

such as when the box $\boldsymbol{X}$ is centered on an approximate root, then the numerator in Equation (1.44) contains zero, and the interval division can be carried out

with a downwardly rounded and an upwardly rounded floating point division, with the formulas

$$\underline{\tilde{x}}_i = \frac{\underline{Y_i^{\mathrm{mid}} F(\check{\boldsymbol{X}})} - \left\{ |Y_i^{\mathrm{mid}}| \Delta_j \frac{\mathrm{w}(\tilde{x}_j)}{2} + \sum_{j=i+1}^{n} |Y_i^{\mathrm{mid}}| \Delta_j \frac{\mathrm{w}(x_j)}{2} \right\}}{1 - |Y_i^{\mathrm{mid}}| \Delta_i},$$

$$\overline{\tilde{x}}_i = \frac{\overline{Y_i^{\mathrm{mid}} F(\check{\boldsymbol{X}})} + \left\{ |Y_i^{\mathrm{mid}}| \Delta_j \frac{\mathrm{w}(\tilde{x}_j)}{2} + \sum_{j=i+1}^{n} |Y_i^{\mathrm{mid}}| \Delta_j \frac{\mathrm{w}(x_j)}{2} \right\}}{1 - |Y_i^{\mathrm{mid}}| \Delta_i}.$$

Shortcuts with Interval Gaussian Elimination

Arithmetic associated with interval Gaussian elimination is similar. In particular, in Algorithm 1 on page 21, $Y^{\mathrm{mid}} \boldsymbol{A} = \boldsymbol{M} = I + \boldsymbol{S}$, and step 1b becomes:

$$\begin{aligned} & \boldsymbol{m}_{j,k} \leftarrow \boldsymbol{m}_{j,k} - (\boldsymbol{m}_{j,i}/\boldsymbol{m}_{i,i})\, \boldsymbol{m}_{i,k} \\ \Leftrightarrow \quad & \boldsymbol{s}_{j,k} \leftarrow \boldsymbol{s}_{j,k} - (\boldsymbol{s}_{j,i})/(1 + \boldsymbol{s}_{i,i})\, \boldsymbol{s}_{i,k}. \end{aligned} \tag{3.7}$$

Thus, all off-diagonal entries in the elimination process are symmetric intervals, and Equation (3.7) can be implemented with upwardly or downwardly rounded floating point arithmetic; if $\boldsymbol{s}_{j,k} = [-s_{j,k}, s_{j,k}]$, then Equation (3.7) becomes

$$s_{j,k} \leftarrow s_{j,k} + (s_{j,i})/(1 - s_{i,i})\, s_{i,k}, \tag{3.8}$$

where the results of all operations except the subtraction should be rounded upward, and the subtraction should be rounded downward. A corresponding simplification is, in general, not possible for the operations on the right-hand-side vector, although the summation in the back-substitution phase can be simplified somewhat (Exercise 4).

3.1.2 Optimality Properties

First introduced by Hansen [72] for interval Gaussian elimination[4] the inverse midpoint preconditioner has often been assumed in the literature to be the best overall choice of preconditioner. For example, Neumaier [175, p. 116] develops a theory based on the inverse midpoint preconditioner. Neumaier

[4] and also explained in [77, pp. 29–31]

defines an interval matrix $\boldsymbol{A}$ to be *strongly regular* provided $Y^{\text{mid}}\boldsymbol{A}$ is regular. He then gives various consequences of strong regularity and various conditions that imply strong regularity.

Besides the resulting symmetry and simplifications described in §3.1, certain optimality properties have been proven for the inverse midpoint preconditioner for the Krawczyk method, as defined in Equation (1.42) on page 56. We have

Theorem 3.2 *(Chen and Wang [31]) Let* $\mathbf{K}(\boldsymbol{X}, \check{X})$ *be as in Equation (1.42) on page 56. Then*

1. $\mathbf{K}(\boldsymbol{X}, \check{X}) \subseteq \boldsymbol{X}$ *only if the spectral radius* $\rho(|(I - Y\boldsymbol{F}'(\boldsymbol{X}))|)$ *is less than 1.*
2. $Y = Y^{\text{mid}}$ *minimizes* $\rho(|(I - Y\boldsymbol{F}'(\boldsymbol{X}))|)$.

Thus, Y^{mid} is optimal for the Krawczyk method. However, for a given Y, the interval Gauss–Seidel method gives a narrower box than the Krawczyk method, so this optimality result should be interpreted carefully.

In interval Newton methods, if $\boldsymbol{F}'(\boldsymbol{X})$ is an interval extension of the Jacobi matrix whose components are of order at least 1, then, as the width $\mathrm{w}(\boldsymbol{X}) \to 0$, $Y^{\text{mid}}\boldsymbol{F}'(\boldsymbol{X}) \to I$, where I is the identity matrix. This is advantageous for the Krawczyk and Gauss–Seidel methods, as well as for interval Gaussian elimination.

In practice, the inverse midpoint preconditioner is an excellent choice in ϵ-inflation procedures[5], in which a small box is centered at a good approximation to a root. However, alternate preconditioners are better in some circumstances.

3.1.3 Exercises

1. Perform the computations below Equation (3.1) to check their correctness. Show all details.
2. Prove Lemma 3.1 by explicitly writing down the results of the operations.

[5]See §4.2 on page 150 below.

3. Prove Equation (3.3).

   Hint: Write down a typical coefficient of the product matrix $Y^{\mathrm{mid}}(\mathrm{m}(\boldsymbol{A}) + \boldsymbol{\Delta})$ *and rearrange the terms using the fact that* $x(y+z) = xy+xz$ *for* $x \in \mathbb{R}$ *and* $y, z \in \mathbb{IR}$*; prove this fact by simply writing down the components. Then use Lemma 3.1.*

4. Using notation and techniques of §3.1.1, simplify step 2b of Algorithm 1 on page 22. (That is, rewrite the step so that most interval operations are replaced by floating point operations with the appropriate directed rounding.)

3.2 OPTIMAL LINEAR PROGRAMMING PRECONDITIONERS

In cases in which $\boldsymbol{F}'(\boldsymbol{X})$ or $\mathbf{S}(F, \boldsymbol{X}, \check{X})$ has some rows with much wider entries than others, preconditioners other than the inverse midpoint preconditioner can be more effective in interval Newton methods.

Example 3.1 (Brown's almost linear function)

$$f_i(X) = x_i + \left(\sum_{1 \le j \le n} x_j - n - 1 \right), \quad 1 \le i \le n-1, \text{ and}$$

$$f_n(X) = \left(\prod_{1 \le j \le n} x_j - 1 \right),$$

with $n = 5$. *This function has exactly three roots in the box*

$$([-10^3, 10^3], [-10^3, 10^3], [-10^3, 10^3], [-10^3, 10^3], [-10^3, 10^3])^T.$$

These roots are contained in the boxes

$$\begin{aligned}
\boldsymbol{X}^{(1)} &= \big([0.999, 1.01], [0.999, 1.01], [0.999, 1.01], [0.999, 1.01], [0.999, 1.01]\big)^T \\
\boldsymbol{X}^{(2)} &= \big([0.916, 0.917], [0.916, 0.917], [0.916, 0.917], [0.916, 0.917], [1.41, 1.42]\big)^T \\
\boldsymbol{X}^{(3)} &= \big([-0.580, -0.579], [-0.580, -0.579], [-0.580, -0.579], \\
&\qquad [-0.580, -0.579], [8.89, 8.90]\big)^T.
\end{aligned}$$

Suppose that an initial box

$$\boldsymbol{X} = ([-2.0, 2.1], [-2.1, 2.2], [-1.9, 2.0], [-2.0, 1.9], [-2.0, 2.05])^T \tag{3.9}$$

is given, and the goal is to apply an interval Newton method to either reduce the size of the box or eliminate the box. An interval Jacobi matrix of Brown's function over this box is

$$\boldsymbol{F}'(\boldsymbol{X}) = \begin{pmatrix} 2 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ [-18.1, 17.7] & [-17.3, 16.9] & [-19.0, 18.5] & [-18.5, 19.0] & [-18.5, 17.7] \end{pmatrix}. \tag{3.10}$$

Since $\boldsymbol{F}'(\boldsymbol{X})$ contains a singular matrix, interval Gaussian elimination cannot be used, but the interval Gauss–Seidel method can possibly reduce the size of some of the coordinates of the box (3.9). The inverse midpoint preconditioner corresponding to the matrix (3.10) is approximately[6]

$$Y^{\text{mid}} \approx \begin{pmatrix} 0.8802 & -0.1257 & -0.1132 & -0.3898 & 0.5988 \\ -0.1198 & 0.8743 & -0.1132 & -0.3898 & 0.5988 \\ -0.1198 & -0.1257 & 0.8868 & -0.3898 & 0.5988 \\ -0.1198 & -0.1257 & -0.1132 & 0.6102 & 0.5988 \\ -0.4012 & -0.3713 & -0.4341 & 0.9491 & -2.9940 \end{pmatrix}.$$

With this preconditioner, the interval Gauss–Seidel method as in Equation (1.44) does not make any progress, i.e. $\tilde{\boldsymbol{x}}_i \supseteq \boldsymbol{x}_i$ for $i = 1, \ldots, 5$. However, with the preconditioner

$$Y = \begin{pmatrix} 0.8 & -0.2 & -0.2 & -0.2 & 0.0 \\ -0.2 & 0.8 & -0.2 & -0.2 & 0.0 \\ -0.2 & -0.2 & 0.8 & -0.2 & 0.0 \\ -0.2 & -0.2 & -0.2 & 0.8 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}, \tag{3.11}$$

Algorithm 2 gives

$$\begin{array}{lcccccl} \tilde{\boldsymbol{x}}_1 \cap \boldsymbol{x}_1 & \subset & [0.789, 1.601] & \subset & [-2.0, 2.1] & = & \boldsymbol{x}_1, \\ \tilde{\boldsymbol{x}}_2 \cap \boldsymbol{x}_2 & \subset & [0.789, 1.601] & \subset & [-2.1, 2.2] & = & \boldsymbol{x}_2, \\ \tilde{\boldsymbol{x}}_3 \cap \boldsymbol{x}_3 & \subset & [0.789, 1.601] & \subset & [-1.9, 2.0] & = & \boldsymbol{x}_3, \\ \tilde{\boldsymbol{x}}_4 \cap \boldsymbol{x}_4 & \subset & [0.789, 1.601] & \subset & [-2.0, 1.9] & = & \boldsymbol{x}_4, \\ \tilde{\boldsymbol{x}}_5 \cap \boldsymbol{x}_5 & & & = & & & \boldsymbol{x}_5. \end{array}$$

Thus, with this preconditioner, the interval Gauss–Seidel method reduces the volume of the box in which all roots in $\boldsymbol{X}$ must lie from approximately 1086

[6] Precise enclosures of preconditioners are usually not necessary, except, as mentioned above on page 116, when implementing shortcuts in the arithmetic.

to 1.76, a factor of approximately 616. This is very efficient compared to generalized bisection[7].

For a second example, suppose

$$\boldsymbol{X} = ([0.5, 0.95], [0.5, 0.95], [0.5, 0.95], [0.5, 0.95], [0.5, 0.95])^T, \tag{3.12}$$

so that there are no roots of F in $\boldsymbol{X}$. The fifth row of an interval Jacobi matrix over this box is

$$\boldsymbol{F'}_{(5,*)}(\boldsymbol{X}) \subseteq \begin{pmatrix}[0.0624, 0.815], & [0.0624, 0.815], & [0.0624, 0.815], & [0.0624, 0.815], \\ [0.0624, 0.815]\end{pmatrix}, \tag{3.13}$$

The inverse midpoint preconditioner for this function is approximately

$$Y^{\text{mid}} \approx \begin{pmatrix} 1 & 0 & 0 & 0 & -2.280 \\ 0 & 1 & 0 & 0 & -2.280 \\ 0 & 0 & 1 & 0 & -2.280 \\ 0 & 0 & 0 & 1 & -2.280 \\ -1 & -1 & -1 & -1 & 11.40 \end{pmatrix}.$$

The factorization in interval Gaussian elimination fails in the second column with this preconditioner, and the interval Gauss–Seidel method does not make any progress, either. With the preconditioner (3.11), interval Gaussian elimination fails in the last column, but $\tilde{\boldsymbol{x}}_1 \cap \boldsymbol{x}_1 = \emptyset$ with the interval Gauss–Seidel method[8].

As a third example, take an initial box

$$\boldsymbol{X} = ([0.96, 1.04], [0.96, 1.04], [0.96, 1.04], [0.96, 1.04], [0.96, 1.04])^T \tag{3.14}$$

for Example 3.1, so that there is a unique root at the center of $\boldsymbol{X}$. In the methods, choose the slope matrix $\mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ with $\check{\boldsymbol{X}} = (0, 0, 0, 0, 0)^T$, whose first four rows are as in (3.10) and whose fifth row is contained in

$$([1, 1], [0.9599, 1.041], [0.9215, 1.083], [0.8846, 1.126], [0.8492, 1.171]).$$

The inverse midpoint preconditioner is approximately

$$Y^{\text{mid}} \approx \begin{pmatrix} 0.9908 & -0.0092 & -0.0077 & -0.0046 & -0.9600 \\ -0.0092 & 0.9908 & -0.0077 & -0.0046 & -0.9600 \\ -0.0092 & -0.0092 & 0.9923 & -0.0046 & -0.9600 \\ -0.0092 & -0.0092 & -0.0077 & 0.9954 & -0.9600 \\ -0.9539 & -0.9539 & -0.9616 & -0.9769 & 4.8000 \end{pmatrix}.$$

[7] See §4.3.2 below.

[8] Actually, in this case, a second-order interval extension based on this $\boldsymbol{F'}(\boldsymbol{X})$ would reveal 0 cannot be in the range $\boldsymbol{F''}(\boldsymbol{X})$, but this is not always the case when preconditioners other than the inverse midpoint preconditioner give better results.

With this preconditioner, interval Gaussian elimination fails in the fifth column, while the interval Gauss–Seidel method gives

$$\begin{array}{lcccccl} \tilde{x}_1 \cap x_1 & \subset & [0.984, 1.014] & \subset & [0.96, 1.04] & = & x_1, \\ \tilde{x}_2 \cap x_2 & \subset & [0.985, 1.015] & \subset & [0.96, 1.04] & = & x_2, \\ \tilde{x}_3 \cap x_3 & \subset & [0.987, 1.013] & \subset & [0.96, 1.04] & = & x_3, \\ \tilde{x}_4 \cap x_4 & \subset & [0.991, 1.009] & \subset & [0.96, 1.04] & = & x_4, \\ \tilde{x}_5 \cap x_5 & & & = & & & x_5. \end{array}$$

Thus, the interval–Newton-based existence test fails[9]. However, if a preconditioner that is approximately

$$Y = \begin{pmatrix} 0.8 & -0.2 & -0.2 & -0.2 & 0.0 \\ -0.2 & 0.8 & -0.2 & -0.2 & 0.0 \\ -0.2 & -0.2 & 0.8 & -0.2 & 0.0 \\ -0.01042 & -0.01042 & -0.00868 & 1.125 & -1.085 \\ -4.134 & -4.134 & -4.168 & -4.234 & 20.80 \end{pmatrix} \tag{3.15}$$

is used, the interval Gauss–Seidel method gives

$$\begin{array}{lcccccl} \tilde{x}_1 \cap x_1 & \subset & [0.991, 1.009] & \subset & [0.96, 1.04] & = & x_1, \\ \tilde{x}_2 \cap x_2 & \subset & [0.991, 1.009] & \subset & [0.96, 1.04] & = & x_2, \\ \tilde{x}_3 \cap x_3 & \subset & [0.991, 1.009] & \subset & [0.96, 1.04] & = & x_3, \\ \tilde{x}_4 \cap x_4 & \subset & [0.992, 1.008] & \subset & [0.96, 1.04] & = & x_4, \\ \tilde{x}_5 \cap x_5 & \subset & [0.96005, 1.03995] & \subset & [0.96, 1.04] & = & x_5. \end{array}$$

Thus, existence of a solution of Brown's nonlinear system $F(X) = 0$ within $\boldsymbol{X}$ cannot be proven with the inverse midpoint preconditioner, but can be proven with the preconditioner[10] (3.15).

The special preconditioners Y in (3.11), the second example, and in (3.15) are examples of *width-optimal preconditioners*, to be discussed in the next section.

3.2.1 General Properties and Classification of Optimal Preconditioners

The techniques here were introduced in [111]. Additional considerations were outlined in [126], and Manuel Novoa did much unpublished work on the sub-

[9] An existence test based on the Krawczyk method would also fail for this slope matrix, since $\mathbf{K}(\boldsymbol{X}, \tilde{\boldsymbol{X}}) \supseteq \mathbf{GS}(\boldsymbol{X})$ for a given preconditioner; see p. 24.

[10] The difference between the power of such preconditioners and the power of the inverse-midpoint preconditioner to prove existence or uniqueness with iteration matrix $\boldsymbol{A}$ is related to the distance between a strongly regular matrix and the nearest singular matrix. This distance is in turn related to the spectral radius of $\mathrm{m}(\boldsymbol{A})^{-1}(\boldsymbol{A} - \mathrm{m}(\boldsymbol{A}))$. Work of Rump has clarified this; see [209, Proposition 7.3 and Conjecture 7.5].

ject, some of which appears here. The works [85] and [222] contain additional ideas and explanation, while some experimental comparisons in the optimization context appear in [194].

Optimal LP preconditioners are based upon heuristics applied to optimize some aspect of $\tilde{x}_i$ in Formula 1.44 at each step[11] of the interval Gauss–Seidel method. Depending on the optimization criterion, different preconditioners are obtained. In turn, the heuristics enable such preconditioners to be computed as solutions to linear programming problems.

Formula 1.44 is restated here for convenience:

$$\tilde{x}_i = \check{x}_i - \left\{ Y_i F(\check{X}) + \sum_{j=1}^{i-1} Y_i \boldsymbol{A}_j (\tilde{x}_j - \check{x}_j) + \sum_{j=i+1}^{n} Y_i \boldsymbol{A}_j (x_j - \check{x}_j) \right\} \Big/ (Y_i \boldsymbol{A}_i).$$

Following [111], we rewrite the numerator as $\mathbf{n}_i(Y_i)$ and the denominator as $\mathbf{d}_i(Y_i)$, i.e.:

$$\begin{aligned} \tilde{x}_i &= \check{x}_i - \frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)} \\ &= \check{x}_i - \frac{[\underline{n}_i(Y_i), \overline{n}_i(Y_i)]}{[\underline{d}_i(Y_i), \overline{d}_i(Y_i)]}. \end{aligned} \tag{3.16}$$

Optimization criteria depend upon whether it is desired or possible that $0 \in Y_i \boldsymbol{A}_i = \mathbf{d}_i(Y_i)$.

Definition 3.2 *A preconditioning row Y_i is a* C-preconditioner *if and only if* $0 \notin \mathbf{d}_i(Y_i)$. *Furthermore, Y_i is a* normal C-preconditioner *provided* $\underline{d}_i(Y_i) = 1$.

Definition 3.3 (Novoa, unpublished) *A preconditioning row Y_i is an* E-preconditioner[12] *if and only if* $0 \in \mathbf{d}_i(Y_i)$ *and* $0 \notin \mathbf{n}_i(Y_i)$. *Furthermore, Y_i is a* normal E-preconditioner *if and only if* $\underline{n}_i(Y_i) = 1$.

A C-preconditioner is meant to *contract* x_i, while an E-preconditioner is meant to *split* x_i with Kahan–Novoa–Ratz arithmetic; see Figures 3.1 and 3.2.

[11] For this reason, such preconditioners are sometimes termed *rowwise preconditioners*.

[12] The original definition, discussed in [126] and [85], was an "S-preconditioner," defined as an "E-" or "extended" preconditioner, but without the condition $0 \notin \mathbf{n}_i(Y_i)$.

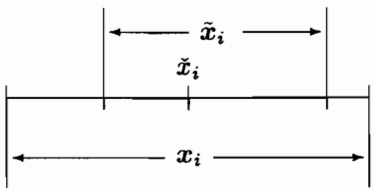


Figure 3.1 Action of a C-preconditioner

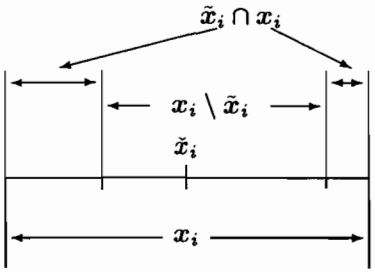


Figure 3.2 Action of an E-preconditioner

All useful preconditioning rows are either C- or E-preconditioners. Any other preconditioner would necessarily have $0 \in \mathbf{n}_i(Y_i)$ and $0 \in \mathbf{d}_i(Y_i)$, so that $\tilde{x}_i = \mathbb{R}$.

In general, it is not easy to determine whether an E-preconditioner exists. However, we have

Lemma 3.3 *(Hu, [85]) There exists a C-preconditioner Y_i if and only if at least one element of the i-th column of $\boldsymbol{A}$ does not contain* 0.

Optimality Criteria for C-preconditioners

We define the following optimality criteria.

Definition 3.4 *Let Y_i be a C-preconditioner.*

1. *Y_i is* width-optimal, *or a* C^{W}-preconditioner, *provided it minimizes the width of $\tilde{x}_i$ over all C-preconditioners.*
2. *Y_i is* left-optimal, *or a* C^{L}-preconditioner, *provided it maximizes the left endpoint of $\tilde{x}_i$ over all C-preconditioners.*
3. *Y_i is* right-optimal, *or a* C^{R}-preconditioner, *provided it minimizes the right endpoint of $\tilde{x}_i$ over all C-preconditioners.*
4. *A C-preconditioner Y_i is magnitude-optimal, or a* C^{M}-preconditioner, *provided it minimizes the magnitude of $\tilde{x}_i - \check{x}_i$ over all C-preconditioners.*

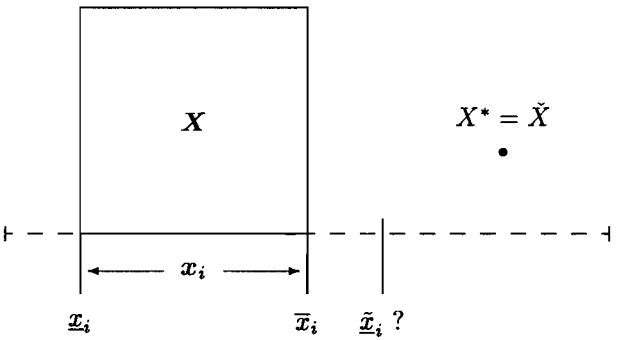


Figure 3.3 Appropriate situation for a left-optimal preconditioner

We now define several specific types of C-preconditioners and E-preconditioners.

Most practical experience has been with width-optimal preconditioners. This preconditioner tends to minimize the volume of the image under the interval Gauss–Seidel method. Ideally, $\mathrm{w}(\tilde{\boldsymbol{x}}_i \cap \boldsymbol{x}_i)$ would be minimized, but it is easier to optimize $\mathrm{w}(\tilde{\boldsymbol{x}}_i)$.

Although there has been less practical experience to date, left-optimal and right-optimal preconditioners can be useful to prove that boxes $\boldsymbol{X}$ do not contain roots. Such preconditioners can be effective if $\boldsymbol{A} = \mathbf{S}(F, \boldsymbol{X}, \check{\boldsymbol{X}})$ where $\check{\boldsymbol{X}}$ is small, if there is an $X^* \in \check{\boldsymbol{X}}$ with $F(X^*) = 0$, and if $\check{\boldsymbol{X}} \cap \boldsymbol{X} = \emptyset$. See Figure 3.3.

It has also been proposed to use the left-optimal and right-optimal preconditioners together to make $\tilde{\boldsymbol{x}}$ as narrow as possible. However, this requires double the amount of work of a single width-optimal preconditioner, and preliminary experiments have hinted that this is not advantageous.

The magnitude-optimal preconditioner would be appropriate in existence and uniqueness, since it maximizes the likelihood that $\tilde{\boldsymbol{x}}_i \subset \boldsymbol{x}_i$ when $\check{\boldsymbol{x}}_i$ is the midpoint of $\boldsymbol{x}_i$; see Figure 3.4.

Optimality Criteria for E-preconditioners

There is a sort of duality between C-preconditioners and E-preconditioners, and each optimality criterion for a C-preconditioner corresponds to an optimality criterion for an E-preconditioner. For example, minimizing the width $\mathrm{w}(\tilde{\boldsymbol{x}}_i)$ for a C-preconditioner corresponds to maximizing the width $\mathrm{w}(\boldsymbol{x}_i \setminus \tilde{\boldsymbol{x}}_i)$ for an

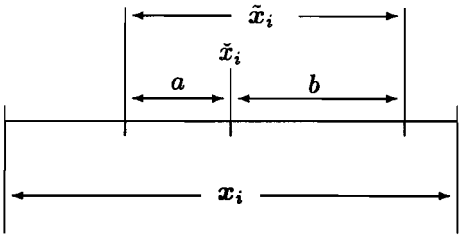


Figure 3.4 Action of a magnitude-optimal C-preconditioner: $\max\{a, b\}$ is minimized.

E-preconditioner. (See Figures 3.1 and 3.2.) The interval $x_i \setminus \tilde{x}_i$ will be called the *i-th gap* or the *i-th solution complement.*

Definition 3.5 *Let Y_i be an E-preconditioner. Then:*

1. *Y_i is* width-optimal, *or an* E^{W}-preconditioner, *provided it maximizes the width of $x_i \setminus \tilde{x}_i$ over all E-preconditioners.*
2. *Y_i is* mignitude-optimal, *or an* E^{M}-preconditioner, *provided it maximizes the mignitude of $(\tilde{x}_i - \check{x}_i)$ over all E-preconditioners.*
3. *Y_i is* left-optimal, *or an* E^{L}-preconditioner, *provided it minimizes the right endpoint of the left component of $\tilde{x}_i$ over all E-preconditioners.*
4. *Y_i is* right-optimal, *or an* E^{R}-preconditioner, *provided it maximizes the left endpoint of the right component of $\tilde{x}_i$ over all E-preconditioners.*

At the time of writing, there has been limited practical experience in the use of E-preconditioners. Since $\tilde{x}_i \cap x_i$ consists of two intervals when Kahan–Novoa–Ratz arithmetic is used, E-preconditioners generally produce two sub-boxes. Since, other things being equal, proliferation of subboxes reduces the overall efficiency of global search algorithms for roots or optima, E-preconditioners are effective only in the appropriate context, embedded in appropriate algorithms with good heuristics. Choice of when to apply a "split" was discussed in [78], and then in [77, §8.8], where a heuristic for when to split the box is given. Experimental results involving splitting were also reported in [85], [194], and

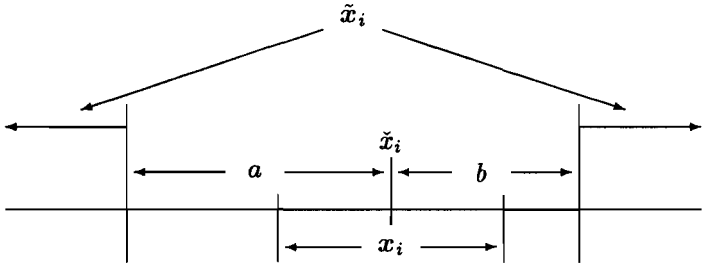


Figure 3.5 Action of a mignitude-optimal E-preconditioner: $\min\{a, b\}$ is maximized.

in a polished form in [196]. However, none of these experiments used optimal E-preconditioners.

The mignitude-optimal E-preconditioner plays a dual role to the magnitude-optimal C-preconditioner. Whereas the magnitude-optimal C-preconditioner maximizes the likelihood that existence or uniqueness will be verified, (see page 126), the mignitude-optimal E-preconditioner, in a sense, maximizes the likelihood that $\tilde{x}_i \cap x_i = \emptyset$, i.e. that non-existence within X can be verified. (See Figure 3.5.) Further experimentation with such preconditioners will probably be rewarding.

3.2.2 Linear Programming Formulations

Each of the preconditioners in definitions 3.4 and 3.5 can be formulated as a nonlinear optimization problem. For example, a normal C^W-preconditioner Y_i is a solution to the nonlinear optimization problem

$$\min_{\underline{d}_i(Y_i)=1} w\left(\frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)}\right). \tag{3.17}$$

Computation of such preconditioners is practical because the problems such as Problem 3.17 can be reformulated as linear programming problems.

The formulation technique involves rewriting each of the components of Y_i as a difference of positive and negative parts, using certain properties of interval arithmetic, such as those of Lemma 3.1, and using standard techniques (such as in [60, §4.2.3] to transform terms involving $\max\{\cdot,\cdot\}$, $\min\{\cdot,\cdot\}$, and $|\cdot|$ into linear terms and constraints. A theoretical issue is what correspondence exists between such linear programming formulations and the original optimization

problem such as Problem 3.17. A practical issue is how to implement the solution process for the linear programming problem: there is substantial structure, and specially-designed codes are several times faster than off-the-shelf solvers.

In unpublished work, Manuel Novoa proposed a general framework for showing, essentially, a one-to-one correspondence between solutions of the original nonlinear optimization problems and equivalence classes of solutions of the linear programming problems, for each of the preconditioners in definitions 3.4 and 3.5 except for the width-optimal E-preconditioner. In such preconditioners, special simplifications are possible if $\check{x}_j = \underline{x}_j$, $\check{x}_j = \overline{x}_j$ or $\check{x}_j = \mathrm{m}(\boldsymbol{x}_j)$. Novoa's general unifying formulation took account of all such possibilities. However, this generality required introduction of notation that made the exposition somewhat harder to interpret and use. For readability, only $\check{x}_j = \mathrm{m}(\boldsymbol{x}_j)$ and the C^{W}-preconditioner, C^{M}-preconditioner, and E^{M}-preconditioner will be considered in depth here, although other choices of $\check{x}_j$ and other optimal preconditioners may be useful in algorithms. Also, for readability of the formulas in the context of primary interest here, the preconditioners will be formulated in terms of the interval Newton system 1.43: $\boldsymbol{A}(X - \check{X}) = -F(\check{X})$, rather than the general linear system $\boldsymbol{AX} = \boldsymbol{B}$, as in Novoa's work. The reader can use the techniques illustrated here to derive additional preconditioners, as appropriate.

One generality will be represented in the formulas presented here: The preconditioned Gauss–Seidel method can be applied for $F : \mathbb{R}^n \to \mathbb{R}^m$, where $m \neq n$. Consideration of such systems could be advantageous in constraint propagation methods, in handling constrained optimization problems, or in taking advantage of different interval extensions arising from rearranging algebraic expressions.

Characterizations of the Gauss–Seidel Expressions

Some additional elementary properties of interval arithmetic are useful here:

Lemma 3.4 *Suppose $y, z \in \mathbb{R}$, suppose $\boldsymbol{x}, \boldsymbol{a} \in \mathbb{IR}$, and denote the positive and negative parts of y by $y^+ = \max\{y, 0\}$ and $y^- = \max\{-y, 0\}$. Then*

1. $y\boldsymbol{a} = [\underline{a}y^+ - \overline{a}y^-, \overline{a}y^+ - \underline{a}y^-] = y\mathrm{m}(a) + \frac{1}{2}|y|\mathrm{w}(\boldsymbol{a})[-1, 1]$.
2. *If* $0 \in \boldsymbol{x}$, *then* $\boldsymbol{ax} = \left[-\max\left\{\overline{a}^+|\underline{x}|, \underline{a}^-|\overline{x}|\right\}, \max\left\{\overline{a}^+|\overline{x}|, \underline{a}^-|\underline{x}|\right\}\right]$.
3. *If* $\mathrm{m}(\boldsymbol{x}) = 0$, *then* $\boldsymbol{ax} = \frac{1}{2}|\boldsymbol{a}|\mathrm{w}(\boldsymbol{x})[-1, 1]$.

4. $|\boldsymbol{x}| = \max\{-\underline{x}, \overline{x}\}$.

5. *If* $\boldsymbol{x} \geq 0$, *then* $\boldsymbol{ax} = \left[\underline{a}^+|\underline{x}| - \underline{a}^-|\overline{x}|, \overline{a}^+|\overline{x}| - \overline{a}^-|\underline{x}|\right]$.

6. *If* $\boldsymbol{x} \leq 0$, *then* $\boldsymbol{ax} = \left[\overline{a}^-|\overline{x}| - \overline{a}^+|\underline{x}|, \underline{a}^-|\underline{x}| - \underline{a}^+|\overline{x}|\right]$.

7. $\max\{y, z\} = y + (z - y)^+ = y + (y - z)^- = z + (y - z)^+ = z + (z - y)^-$.

Now consider the numerator and denominator in the Gauss–Seidel iteration Formula 1.44, namely:

$$\mathbf{n}_i(Y_i) = Y_i F(\check{X}) + \sum_{j=1}^{i-1} Y_i \boldsymbol{A}_j(\tilde{\boldsymbol{x}}_j - \check{\boldsymbol{x}}_j) + \sum_{j=i+1}^{n} Y_i \boldsymbol{A}_j(\boldsymbol{x}_j - \check{\boldsymbol{x}}_j)$$

and

$$\mathbf{d}_i(Y_i) = \sum_{k=1}^{m} y_k \boldsymbol{a}_{k,i}.$$

This numerator and denominator have the following characterization.

Lemma 3.5 *Suppose* $F = (f_1, \ldots, f_n) : \mathbb{R}^n \to \mathbb{R}^m$, $Y_i = (y_1, \ldots y_m)$, *and suppose* $\check{x}_j = \mathrm{m}(\boldsymbol{x}_j)$ *for* $1 \leq j \leq n$. *Then the numerator in the Gauss–Seidel iteration Formula 1.44 is*

$$\begin{aligned} \mathbf{n}_i(Y_i) \quad = \quad & \sum_{k=1}^{m} y_k \mathrm{m}\big(f_k(\check{X})\big) + \frac{1}{2}\sum_{k=1}^{m} |y_k| \mathrm{w}\big(f_k(\check{X})\big)[-1, 1] \\ & + \frac{1}{2}\sum_{\substack{j=1 \\ j \neq i}}^{n} \mathrm{w}(\boldsymbol{x}_j) \left| \sum_{k=1}^{m} y_k \boldsymbol{a}_{k,j} \right| [-1, 1], \end{aligned} \tag{3.18}$$

while the denominator is

$$\mathbf{d}_i(Y_i) = \sum_{k=1}^{m} \mathrm{m}(\boldsymbol{a}_{k,i}) + \frac{1}{2}\sum_{k=1}^{m} |y_k| \mathrm{w}(\boldsymbol{a}_{k,i})[-1, 1]. \tag{3.19}$$

Lemma 3.5 follows directly from Lemma 3.4.

To represent $\underline{n}_i(Y_i)$, $\overline{n}_i(Y_i)$, and $\mathrm{w}(\mathbf{n}_i(Y_i))$ in the linear programming problems, variables:

$$v_j = \overline{\sum_{k=1}^{m} y_k \boldsymbol{a}_{k,j}} + \sum_{k=1}^{m} y_k \boldsymbol{a}_{k,j}, \quad 1 \leq j \leq n, \quad j \neq i \tag{3.20}$$

are used. To see how these v_j fit into Formula (3.18), observe first that Equation (1) of Lemma 3.4 implies:

$$\begin{aligned} \overline{\sum_{k=1}^{m} y_k a_{k,j}} &= \sum_{k=1}^{m} \left(y_k^+ \overline{a}_{k,j} - y_k^- \underline{a}_{k,j}\right) \qquad \text{and} \\ \underline{\sum_{k=1}^{m} y_k a_{k,j}} &= \sum_{k=1}^{m} \left(y_k^+ \underline{a}_{k,j} - y_k^- \overline{a}_{k,j}\right). \end{aligned} \tag{3.21}$$

Formula (4) of Lemma 3.4 then implies

$$\left|\sum_{k=1}^{m} y_k a_{k,j}\right| = \max\left\{-\sum_{k=1}^{m}\left(y_k^+ \underline{a}_{k,j} - y_k^- \overline{a}_{k,j}\right), \sum_{k=1}^{m}\left(y_k^+ \overline{a}_{k,j} - y_k^- \underline{a}_{k,j}\right)\right\}. \tag{3.22}$$

Also, directly adding the two right-hand-sides in Equation (3.21) gives

$$\begin{aligned} v_j &= \overline{\sum_{k=1}^{m} y_k a_{k,j}} - \left\{-\underline{\sum_{k=1}^{m} y_k a_{k,j}}\right\} \\ &= \sum_{k=1}^{m}\left(y_k^+ \overline{a}_{k,j} - y_k^- \underline{a}_{k,j}\right) + \sum_{k=1}^{m}\left(y_k^+ \underline{a}_{k,j} - y_k^- \overline{a}_{k,j}\right) \\ &= \sum_{k=1}^{m}(y_k^+ - y_k^-)(\underline{a}_{k,j} + \overline{a}_{k,j}). \end{aligned} \tag{3.23}$$

Identifying y in Equation (7) of Lemma 3.4 with $-\underline{\sum_{k=1}^{m} y_k a_{k,j}}$ and identifying z in Equation (7) of Lemma 3.4 with $\overline{\sum_{k=1}^{m} y_k a_{k,j}}$, equations (3.22) and (3.23) combined with the first part of Equation (7) of Lemma 3.4 give

$$\left|\sum_{k=1}^{m} y_k a_{k,j}\right| = \left\{\sum_{k=1}^{m}\left(-y_k^+ \underline{a}_{k,j} + y_k^- \overline{a}_{k,j}\right)\right\} + v_j^+. \tag{3.24}$$

Similarly, the last part of Equation (7) of Lemma 3.4 gives

$$\left|\sum_{k=1}^{m} y_k a_{k,j}\right| = \left\{\sum_{k=1}^{m}\left(y_k^+ \overline{a}_{k,j} - y_k^- \underline{a}_{k,j}\right)\right\} + v_j^-. \tag{3.25}$$

For flexibility (and to tune the linear programming problems for numerical stability in their solution), equations (3.24) and (3.25) can be added to obtain

$$\left|\sum_{k=1}^{m} y_k \boldsymbol{a}_{k,j}\right| = \delta\left\{\sum_{k=1}^{m}\left(-y_k^+ \underline{a}_{k,j} + y_k^- \overline{a}_{k,j}\right) + v_j^+\right\} + (1-\delta)\left\{\sum_{k=1}^{m}\left(y_k^+ \overline{a}_{k,j} - y_k^- \underline{a}_{k,j}\right) + v_j^-\right\}, \quad (3.26)$$

where, for various reasons, $\delta \in [0,1]$.

Similarly, the denominator $\mathbf{d}_i(Y_i)$ in the Gauss–Seidel iteration Formula (1.44) can be represented as

$$\begin{aligned}\mathbf{d}_i(Y_i) &= \sum_{k=1}^{m} y_k \boldsymbol{a}_{k,i} \\ &= \left[\sum_{k=1}^{m}\left(y_k^+ \underline{a}_{k,i} - y_k^- \overline{a}_{k,i}\right), \sum_{k=1}^{m}\left(y_k^+ \overline{a}_{k,i} - y_k^- \underline{a}_{k,i}\right)\right], \end{aligned} \quad (3.27)$$

and

$$|\mathbf{d}_i(Y_i)| = \delta\left\{v_i^+ - \sum_{k=1}^{m}\left(y_k^+ \underline{a}_{k,i} - y_k^- \overline{a}_{k,i}\right)\right\} + (1-\delta)\left\{v_i^- + \sum_{k=1}^{m}\left(y_k^+ \overline{a}_{k,i} - y_k^- \underline{a}_{k,i}\right)\right\}. \quad (3.28)$$

In the linear programming problems, y_k^+, y_k^-, v_j^+ and v_j^- will be viewed as $2(m+n-1)$ domain variables, subject to non-negativity constraints. The definition of v_j and Equation (3.23) thus lead to the equality constraints

$$v_j^+ - v_j^- - \sum_{k=1}^{m}(y_k^+ - y_k^-)(\underline{a}_{k,j} + \overline{a}_{k,j}) = 0, \quad 1 \le j \le n, \quad j \ne i \quad (3.29)$$

Lemma 3.5 will be used with (3.26), (3.28), and (3.27) to construct the objective function and constraints corresponding to normalization conditions.

The Width-Optimal Contraction Preconditioner

Recall that the width-optimal C^W-preconditioner is the solution to (3.17), namely

$$\min_{\underline{d}_i(Y_i)=1} \mathrm{w}\left(\frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)}\right).$$

First observe

Lemma 3.6 *Assume*[13] $0 \in \mathbf{n}_i(Y_i)$ *and* $\underline{d}_i(Y_i) = 1$ *(valid if* $\|F(\check{X})\|$ *is small). Then*

$$\mathbf{n}_i(Y_i)/\mathbf{d}_i(Y_i) = \mathbf{n}_i(Y_i).$$

On the other hand, Lemma 3.5 gives

$$\mathrm{w}(\mathbf{n}_i(Y_i)) = \sum_{k=1}^{m} |y_k| \mathrm{w}\big(f_k(\check{X})\big) + \sum_{\substack{j=1 \\ j\neq i}}^{n} \mathrm{w}(\boldsymbol{x}_j) \left| \sum_{k=1}^{m} y_k \boldsymbol{a}_{k,j} \right|. \tag{3.30}$$

Combining (3.30) with (3.26) gives the objective function

$$\begin{aligned} \mathcal{O}_{C^W} \;=\; & \sum_{k=1}^{m} (y_k^+ + y_k^-) \mathrm{w}\big(f_k(\check{X})\big) \\ & + \quad \delta \sum_{\substack{j=1 \\ j\neq i}}^{n} \mathrm{w}(\boldsymbol{x}_j) \left\{ v_j^+ + \sum_{k=1}^{m} \left(y_k^- \overline{a}_{k,j} - y_k^+ \underline{a}_{k,j} \right) \right\} \\ & + (1-\delta) \sum_{\substack{j=1 \\ j\neq i}}^{n} \mathrm{w}(\boldsymbol{x}_j) \left\{ v_j^- + \sum_{k=1}^{m} \left(y_k^+ \overline{a}_{k,j} - y_k^- \underline{a}_{k,j} \right) \right\}. \end{aligned} \tag{3.31}$$

The constraint corresponding to the normalization condition $\underline{d}_i(Y_i) = 1$ of the optimization problem (3.17) makes use of (3.27):

$$\sum_{k=1}^{m} \left(y_k^+ \underline{a}_{k,i} - y_k^- \overline{a}_{k,i} \right) = 1. \tag{3.32}$$

[13] If $0 \notin \mathbf{n}_i(Y_i)$, then *any* preconditioner will result in $\mathrm{w}(\tilde{\boldsymbol{x}}_i \cap \boldsymbol{x}_i) \leq .5\mathrm{w}(\boldsymbol{x}_i)$; see [111].

Definition 3.6 *Any solution of the linear programming problem with the* $2(m+n-1)$ *variables*

$$(y_1^+, \ldots, y_m^+, y_1^-, \ldots, y_m^-, v_1^+, \ldots, v_{i-1}^+, v_{i+1}^+, \ldots, v_n^+, v_1^- \ldots, v_{i-1}^-, v_{i+1}^-, \ldots, v_n^-)$$

with the objective function $\mathcal{O}_{\mathrm{C^W}}$ *in (3.31), and with the n constraints (3.29) and (3.32), and subject to non-negativity constraints on all of the variables, is called a* $\mathrm{C^W}$-LP-preconditioner *for the i-th variable.*

There is a slight abuse of notation in Definition 3.6, since y_k^+, y_k^-, v_j^+, and v_j^- are not necessarily the positive and negative parts of numbers. In fact, if the $\mathrm{C^W}$-LP preconditioner is denoted

$$(y_1', \ldots, y_m', y_1'', \ldots, y_m'', v_1', \ldots, v_{i-1}', v_{i+1}', \ldots, v_n', v_1'' \ldots, v_{i-1}'', v_{i+1}'', \ldots, v_n''),$$

then an actual preconditioner is formed by taking

$$y_k = y_k' - y_k'', \quad 1 \le k \le m. \tag{3.33}$$

Conditions under which this preconditioner solves the optimization problem (3.17) are discussed briefly in §3.2.3, and are verified in [180].

The Magnitude-Optimal Contraction Preconditioner

The $\mathrm{C^M}$-preconditioner is the solution to the nonlinear optimization problem

$$\min_{\underline{d}_i(Y_i)=1} \left| \frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)} \right|, \tag{3.34}$$

where $\mathbf{n}_i(Y_i)/\mathbf{d}_i(Y_i) = \mathbf{n}_i(Y_i)$ if $\underline{d}_i(Y_i) = 1$ and[14] $0 \in \mathbf{n}_i(Y_i)$. Three variables

$$M = |\mathbf{n}_i(Y_i)|, \quad \overline{s}, \quad \underline{s} \tag{3.35}$$

will now be introduced. Then

$$M = \max\{-\underline{n}_i(Y_i), \overline{n}_i(Y_i)\},$$

$M \ge -\underline{n}_i(Y_i)$, $M \ge \overline{n}_i(Y_i)$. Thinking of $\overline{s}$ and $\underline{s}$ as slack variables, we obtain $M = -\underline{n}_i(Y_i) + \underline{s}$, $M = \overline{n}_i(Y_i) + \overline{s}$. This and Equation (4) of Lemma 3.4 will

[14]This assumption is valid in cases in which the magnitude-optimal preconditioner is most likely to be effective, namely, when $\check{X} = \mathrm{m}(\boldsymbol{X})$ and $F(\check{X}) \approx 0$.

now be used for inequality constraints to characterize M. First, (3.18) along with the definition of y_k^+ and y_k^- and (3.26) imply

$$\begin{aligned}\underline{n}_i(Y_i) &= \sum_{k=1}^{m}(y_k^+ - y_k^-)\mathrm{m}\big(f_k(\check{X})\big) - \frac{1}{2}\sum_{k=1}^{m}(y_k^+ + y_k^-)\mathrm{w}\big(f_k(\check{X})\big) \\ &\quad -\frac{\delta}{2}\sum_{\substack{j=1\\ j\neq i}}^{n}\mathrm{w}(x_j)\left\{\sum_{k=1}^{m}\left(y_k^-\overline{a}_{k,j} - y_k^+\underline{a}_{k,j}\right) + v_j^+\right\} \\ &\quad -\frac{1-\delta}{2}\sum_{\substack{j=1\\ j\neq i}}^{n}\mathrm{w}(x_j)\left\{\sum_{k=1}^{m}\left(y_k^+\overline{a}_{k,j} - y_k^-\underline{a}_{k,j}\right) + v_j^-\right\}. \quad (3.36)\end{aligned}$$

Similarly,

$$\begin{aligned}\overline{n}_i(Y_i) &= \sum_{k=1}^{m}(y_k^+ - y_k^-)\mathrm{m}\big(f_k(\check{X})\big) + \frac{1}{2}\sum_{k=1}^{m}(y_k^+ + y_k^-)\mathrm{w}\big(f_k(\check{X})\big) \\ &\quad +\frac{\delta}{2}\sum_{\substack{j=1\\ j\neq i}}^{n}\mathrm{w}(x_j)\left\{\sum_{k=1}^{m}\left(y_k^-\overline{a}_{k,j} - y_k^+\underline{a}_{k,j}\right) + v_j^+\right\} \\ &\quad +\frac{1-\delta}{2}\sum_{\substack{j=1\\ j\neq i}}^{n}\mathrm{w}(x_j)\left\{\sum_{k=1}^{m}\left(y_k^+\overline{a}_{k,j} - y_k^-\underline{a}_{k,j}\right) + v_j^-\right\}. \quad (3.37)\end{aligned}$$

The objective function for the magnitude-optimal preconditioner is thus

$$\mathcal{O}_{\mathrm{C}^{\mathrm{M}}} = M. \quad (3.38)$$

Equation (4) of Lemma 3.4 and the definitions of M, $\overline{s}$, and $\underline{s}$, along with the assumption that $0 \in \mathbf{n}_i(Y_i)$ and the fact

$$|a| = \max\{\underline{a}^-, \overline{a}^+\},$$

are now used to formulate the constraints

$$\begin{aligned}\underline{s} - M - \underline{n}_i(Y_i) &= 0, \\ \overline{s} - M + \overline{n}_i(Y_i) &= 0.\end{aligned} \quad (3.39)$$

These formulas lead to

Definition 3.7 *Any solution of the linear programming problem with the* $2(m+n-1)+3$ *variables*

$$\left(y_1^+, \ldots, y_m^+, y_1^-, \ldots, y_m^-, v_1^+, \ldots, v_{i-1}^+, v_{i+1}^+, \ldots, v_n^+, v_1^- \ldots, v_{i-1}^-, v_{i+1}^-, \ldots, v_n^-, M, \overline{s}, \underline{s}\right)$$

with the objective function $\mathcal{O}_{\mathrm{C^M}}$ *in (3.38), and with the* $n+2$ *constraints (3.29), (3.32) and (3.39) (with* $\underline{n}_i(Y_i)$ *and* $\overline{n}_i(Y_i)$ *represented by (3.36) and (3.37), respectively), and subject to non-negativity constraints on all of the variables, is called a* $\mathrm{C^M}$-LP-preconditioner *for the* i*-th variable.*

Analogously to the width-optimal preconditioner, denote a $\mathrm{C^M}$-LP-preconditioner by

$$\left(y_1', \ldots, y_m', y_1'', \ldots, y_m'', v_1', \ldots, v_{i-1}', v_{i+1}', \ldots, v_n', v_1'' \ldots, v_{i-1}'', v_{i+1}'', \ldots, v_n'', M, M', M''\right),$$

then form the actual preconditioner by

$$y_k = y_k' - y_k'', \quad 1 \leq k \leq m.$$

Theory of when this preconditioner solves the optimization problem (3.34) is similar to that for the width-optimal LP-preconditioner. See §3.2.3, and see [180] for a thorough and generalized consideration of the subject.

The Mignitude-Optimal Splitting Preconditioner

Assuming $0 \in \mathbf{d}_i(Y_i)$, and $\mathbf{n}_i(Y_i) > 0$, Kahan–Novoa–Ratz arithmetic gives

$$\frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)} = \left(-\infty, \frac{\underline{n}_i(Y_i)}{\underline{d}_i(Y_i)}\right] \bigcup \left[\frac{\overline{n}_i(Y_i)}{\overline{d}_i(Y_i)}, \infty\right).$$

Thus, the $\mathrm{E^M}$-preconditioner is the solution to the nonlinear optimization problem

$$\max_{\underline{n}_i(Y_i)=1} \min\left\{-\frac{1}{\underline{d}_i(Y_i)}, \frac{1}{\overline{d}_i(Y_i)}\right\} \tag{3.40}$$

However, $0 \in \mathbf{d}_i(Y_i)$ implies

$$\begin{aligned} \min\left\{-\frac{1}{\underline{d}_i(Y_i)}, \frac{1}{\overline{d}_i(Y_i)}\right\} &= \frac{1}{\max\{|\underline{d}_i(Y_i)|, |\overline{d}_i(Y_i)|\}} \\ &= \frac{1}{|\mathbf{d}_i(Y_i)|}, \end{aligned}$$

and $1/|\mathbf{d}_i(Y_i)|$ is maximum when $|\mathbf{d}_i(Y_i)|$ is minimum. Thus, Problem (3.40) can be replaced by

$$\min_{\underline{n}_i(Y_i)=1} |\mathbf{d}_i(Y_i)|. \tag{3.41}$$

Problem (3.41) can be used directly to define a linear programming problem for the mignitude-optimal LP-preconditioner. With (3.36), the normalization condition $\underline{n}_i(Y_i) = 1$ can be written as the constraint

$$\begin{aligned}\sum_{k=1}^{m}(y_k^+ - y_k^-)\mathrm{m}\big(f_k(\check{X})\big) - \frac{1}{2}\sum_{k=1}^{m}(y_k^+ + y_k^-)\mathrm{w}\big(f_k(\check{X})\big) \\ -\frac{\delta}{2}\sum_{\substack{j=1\\ j\neq i}}^{n}\mathrm{w}(x_j)\left\{\sum_{k=1}^{m}\left(y_k^-\overline{a}_{k,j} - y_k^+\underline{a}_{k,j}\right) + v_j^+\right\} \\ -\frac{1-\delta}{2}\sum_{\substack{j=1\\ j\neq i}}^{n}\mathrm{w}(x_j)\left\{\sum_{k=1}^{m}\left(y_k^+\overline{a}_{k,j} - y_k^-\underline{a}_{k,j}\right) + v_j^-\right\} \;=\; 1. \end{aligned} \tag{3.42}$$

Definition 3.8 *Any solution of the linear programming problem with the* $2(m+n-1)$ *variables*

$$(y_1^+, \ldots, y_m^+, y_1^-, \ldots, y_m^-, v_1^+, \ldots, v_{i-1}^+, v_{i+1}^+, \ldots, v_n^+, v_1^- \ldots, v_{i-1}^-, v_{i+1}^-, \ldots, v_n^-)$$

with the objective function consisting of the right member of (3.28), and with the n *constraints consisting of (3.42) and (3.29), and subject to non-negativity constraints on all of the variables, is called an* E^{M}-LP-preconditioner *for the* i*-th variable.*

As with the C^{W}-preconditioner and C^{M}-preconditioner, an actual preconditioner is formed from a computed E^{M}-LP-preconditioner of the form

$$(y_1', \ldots, y_m', y_1'', \ldots, y_m'', v_1', \ldots, v_{i-1}', v_{i+1}', \ldots, v_n', v_1'' \ldots, v_{i-1}'', v_{i+1}'', \ldots, v_n'')$$

by taking

$$y_k = y_k' - y_k'', \quad 1 \leq k \leq m.$$

Theory of when this preconditioner actually solves Problem (3.41) can also be found in [180]. An interesting property of the solution to the optimization problem 3.41 is the following.

Lemma 3.7 *Suppose a preconditioner Y_i solves Problem 3.41.*

1. *If $0 \in \mathbf{d}_i(Y_i)$, then Y_i solves Problem 3.40, and hence is a magnitude-optimal splitting preconditioner.*
2. *If $\mathbf{d}_i(Y_i) > 0$ then Y_i is a left-optimal contraction preconditioner.*
3. *If $\mathbf{d}_i(Y_i) < 0$ then Y_i is a left-optimal contraction preconditioner.*

Proof: Y_i solves Problem 3.41, resulting in $\mathbf{n}_i(Y_i) = [1, a]$ for some $a > 0$. If $0 \in \mathbf{d}_i(Y_i)$, then

$$\frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)} = \frac{[1,a]}{[\underline{d}_i(Y_i), \overline{d}_i(Y_i)]} = \left(-\infty, \frac{1}{\underline{d}_i(Y_i)}\right] \cup \left[\frac{1}{\overline{d}_i(Y_i)}, \infty\right).$$

If $\mathbf{d}_i(Y_i) > 0$, then

$$\frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)} = \frac{[1,a]}{[\underline{d}_i(Y_i), \overline{d}_i(Y_i)]} = \left[\frac{1}{\overline{d}_i(Y_i)}, \frac{a}{\underline{d}_i(Y_i)}\right].$$

Since $\overline{d}_i(Y_i)$ is minimized in this case, this is a left-optimal preconditioner. Similarly, if $\mathbf{d}_i(Y_i) < 0$, then

$$\frac{\mathbf{n}_i(Y_i)}{\mathbf{d}_i(Y_i)} = \frac{[1,a]}{[\underline{d}_i(Y_i), \overline{d}_i(Y_i)]} = \left[\frac{a}{\underline{d}_i(Y_i)}, \frac{1}{\overline{d}_i(Y_i)}\right].$$

Since $-\overline{d}_i(Y_i)$ is minimized in this case, this is a right-optimal preconditioner. □

3.2.3 On Theory of LP Preconditioners

It is instructive to know, for example, when the C^W-LP-preconditioner as in Definition 3.6 on page 134 corresponds to the width-optimal preconditioner as in Definition 3.4. In [180], Novoa develops a general theory of when the solution of a linear programming problem corresponds to an optimal preconditioner. Novoa sets up a one-to-one correspondence between preconditioners satisfying optimality conditions (such as width-optimality) and corresponding solutions to linear programming problems (such as in Definition 3.6).

Novoa defines the set of *normal* solutions to the linear programming problem to be that set of solutions of the form

$$(y_1^+, \ldots, y_m^+, y_1^-, \ldots, y_m^-, v_1^+, \ldots, v_{i-1}^+, v_{i+1}^+, \ldots, v_n^+, v_1^- \ldots, v_{i-1}^-, v_{i+1}^-, \ldots, v_n^-)$$

(or of the form

$$\left(y_1^+, \ldots, y_m^+, y_1^-, \ldots, y_m^-, v_1^+, \ldots, v_{i-1}^+, v_{i+1}^+, \ldots, v_n^+, v_1^- \ldots, v_{i-1}^-, v_{i+1}^-, \ldots, v_n^-, M, \overline{s}, \underline{s}\right)$$

in the case of the mignitude-optimal splitting preconditioner) for which $y_k^+ y_k^- = 0$, $v_j^+ v_j^- = 0$ and $\overline{s}\underline{s} = 0$ for all relevant k and j. Novoa then defines an equivalence class of solutions to be that set of solutions to the linear programming problem that map to nonzero scalar multiples of the same preconditioner under the mapping P defined by $y_k \leftarrow y_k^+ - y_k^-$. Novoa then asserts that

1. the linear programming problem is feasible if and only if the corresponding preconditioner exists;
2. the mapping P from the set of equivalence classes of solutions of the linear programming problem to the set of equivalence classes of optimal preconditioner rows[15] is one-to-one and onto.

Details are available in TEX, "dvi" and Postscript forms at the FTP site

`interval.usl.edu`

A general treatment is in the directory

`pub/interval_math/papers/Novoa's-unpublished-work/1993_work/`

while a more readable version, but without some concepts, is found in

`pub/interval_math/papers/Novoa's-unpublished-work/1991_work/`

in the files `lp_preconditioners.dvi` or `lp_preconditioners.ps`.

The above papers also treat formulations of linear programming problems when $\check{x}_k \neq \mathrm{m}(x_k)$, possibly of use in various circumstances.

3.2.4 Structure in the Linear Programming Problems

The linear programming problems for various optimal preconditioners have a structure that should be utilized. Beginning with the experiments in [111], it

[15] where two preconditioner rows are equivalent if they differ by a scalar multiple

has been observed that full utilization of this structure results in approximately a factor of five improvement in execution speed for width-optimal preconditioner computations when the derivative or slope matrix $\boldsymbol{A}$ is dense. (Similar speed-ups should also be observable for sparse $\boldsymbol{A}$ when both the intrinsic structure and sparsity structure, rather than just sparsity structure, are utilized.) Also, for dense $\boldsymbol{A} \in \mathbb{IR}^{n \times n}$, storage for a tableau for the simplex method can be reduced from $4n^2 + 3n - 2$ to $n^2 + 7n + 3$, as in the routine[16] C_LP_DENSE in INTOPT_90.

The structure will be reviewed here for the width-optimal C^{W}-preconditioner with $\delta = 1$, following [94]. The linear programming problem as in Definition 3.6 can be written as

$$\begin{array}{l} \text{minimize } \mathcal{C}^T \mathcal{X} \\ \text{subject to} \\ \quad \mathcal{A}\mathcal{X} = \mathcal{B}, \\ \quad \mathcal{X} \geq 0, \end{array} \tag{3.43}$$

where

$$\begin{aligned} \mathcal{A} &= \begin{pmatrix} \check{A}_{\neg i}^T & -\check{A}_{\neg i}^T & -I_{n-1} & I_{n-1} \\ \underline{A}_i^T & -\overline{A}_i^T & 0_{1\times(n-1)} & 0_{1\times(n-1)} \end{pmatrix}, \\ \mathcal{B} &= \begin{pmatrix} 0_{(n-1)\times 1} \\ 1 \end{pmatrix}, \\ \mathcal{C} &= \begin{pmatrix} -\mathrm{w}(X_{\neg i})^T \underline{A}_{\neg i}^T, & \mathrm{w}(X_{\neg i})^T \overline{A}_{\neg i}^T, & \mathrm{w}(X_{\neg i})^T, & 0_{1\times(n-1)} \end{pmatrix}, \\ \mathcal{X} &= \left(Y^+, Y^-, U^+, U^-\right)^T, \end{aligned}$$

where

$$\begin{aligned} Y^+ &= (y_1^+, \ldots, y_n^+)^T, \\ Y^- &= (y_1^-, \ldots, y_n^-)^T, \\ U^+ &= \tfrac{1}{2}\left(v_1^+, \ldots, v_{i-1}^+, v_{i+1}^+, \ldots, v_n^+\right) \\ U^- &= \tfrac{1}{2}\left(v_1^-, \ldots, v_{i-1}^-, v_{i+1}^-, \ldots, v_n^-\right), \end{aligned}$$

$\check{A}_{\neg i} \in \mathbb{R}^{n\times(n-1)}$ is the midpoint matrix of $\boldsymbol{A}$ with the i-th column removed, $\underline{A}_{\neg i} \in \mathbb{R}^{n\times(n-1)}$ is the lower bound matrix of $\boldsymbol{A}$ with the i-th column removed, $\overline{A}_{\neg i}$ is the upper bound matrix of $\boldsymbol{A}$ with the i-th column removed, $\mathrm{w}(X_{\neg i})$ is the vector $\mathrm{w}(\boldsymbol{X})$ with the i-th component removed, $\underline{A}_i^T$ is the i-th column of the lower bound matrix of $\boldsymbol{A}$, $\overline{A}_i^T$ is the i-th column of the upper bound matrix

[16] C_LP_DENSE is a minor modification of a FORTRAN-77 routine due to Manuel Novoa.

of $\boldsymbol{A}$, $I_{n-1} \in \mathbb{R}^{(n-1)\times(n-1)}$ is the identity matrix, and $0_{p,q} \in \mathbb{R}^{p\times q}$ is a matrix of 0's.

When carrying out the simplex method, only the midpoint matrix, corresponding to only the first column of $\mathcal{A}$, need be stored and updated as pivoting proceeds. Values in the other columns may be obtained quickly from these values.

3.2.5 Advice on Choice of Preconditioner

Our empirical observations have shown that, for small, dense $\boldsymbol{A}$ and utilizing the structure of the linear programming problem, the C^W-preconditioner can be computed as quickly as the inverse midpoint preconditioner. In such cases, the C^W-preconditioner is preferable to the inverse midpoint preconditioner when the interval Gauss–Seidel method is used. Furthermore, in such situations, either the interval Gauss–Seidel method or interval Gaussian elimination is preferable to the Krawczyk method, since the image under the interval Gauss–Seidel method is never larger than the image under the Krawczyk method.

Depending on how sparsity is utilized in solution of the linear programming problem, the C^W-preconditioner should be preferable to the inverse midpoint preconditioner for large sparse problems.

The C^W-preconditioner is a good general-purpose preconditioner. However, the magnitude-optimal contraction preconditioner is, in theory, better in existence and uniqueness tests. Furthermore, the mignitude-optimal splitting preconditioner can be an efficient alternative to generalized bisection in branch and bound algorithms for nonlinear systems and global optimization. Further research into both these preconditioners and into the branch and bound algorithms will be useful.

The inverse midpoint preconditioner, optimal for the Krawczyk method, is certainly appropriate when the Krawczyk method is used, such as in special applications such as the sensitivity analysis in [208, §2.6]. Furthermore, the difference in power between the inverse midpoint preconditioner and optimal preconditioners is bounded in existence and uniqueness tests. Referring to (3.2) and [208, (1.14)], a matrix is strongly regular if and only if

$$\rho\left(|\mathrm{m}(\boldsymbol{A})^{-1}|\Delta\right) < 1,$$

where ρ represents the spectral radius. In this situation, the inverse midpoint preconditioner and the Krawczyk method can result in $\mathbf{K}(\boldsymbol{X}, \check{X}) \subset \boldsymbol{X}$ [208]. On the other hand, [208, (1.16)] and [210, Proposition 7.3], show that if

$$\rho\left(|\mathrm{m}(\boldsymbol{A})^{-1}|\Delta\right) > 2.321n^{1.7}, \tag{3.44}$$

then $\boldsymbol{A}$ must contain a singular matrix, and hence, no preconditioner and no interval Newton method can prove existence or uniqueness. Furthermore, examination of the constraints in the corresponding linear programming problem shows that the width-optimal contraction preconditioner must approach the inverse midpoint preconditioner as $\mathrm{w}(\boldsymbol{A}) \to 0$. Finally, software for solving structured linear systems may be more generally available than specialized linear programming solvers for LP-preconditioners. Thus, the inverse midpoint preconditioner may be a good choice in existence or uniqueness verification procedures for narrow $\boldsymbol{X}$, in the absence of appropriate software for LP-preconditioners.

There has been some experience with C^{M} and E^{M} preconditioners.

Example 3.2 *(First presented in [128]) Consider*

$$\begin{aligned} f_i(X) &= x_i + \sum_{1 \le j \le n} x_j - n - 1, \quad 1 \le i \le n-1, \\ f_n(X) &= \prod_{1 \le j \le n} x_j - 1, \end{aligned}$$

with $n = 5$ and initial box $\boldsymbol{X} = [0, 0.5] \times [0, 0.5] \times [0, 0.5] \times [0, 0.5] \times [0, 17]$. The mean value extension, with a slope matrix, over the initial box is:

$$\boldsymbol{F}_2(\boldsymbol{X}) = \begin{bmatrix} [-6, 13.5] \\ [-6, 13.5] \\ [-6, 13.5] \\ [-6, 13.5] \\ [-1.996, 0.0625] \end{bmatrix}.$$

In Example 3.2, there is no root in the initial box, but the second-order extension cannot show this, since $0 \in \boldsymbol{F}_2(\boldsymbol{X})$. (Also, a natural interval extension gives

$$F(X) \in \left([-6, 13.5], [-6, 13.5], [-6, 13.5], [-6, 13.5], [-1, 0.0625]\right)^T,$$

so non-existence cannot be obtained from that natural extension.) Thus (since non-existence with a second-order extension is equivalent to non-existence with the Jacobi method without a preconditioner), a preconditioner would be required. However, the E^M-preconditioner with the interval Gauss–Seidel method gives $\tilde{x}_1 \cap x_1 = \emptyset$, so only one preconditioner need be applied to show that there are no roots. In contrast, an empty intersection is not obtained in multiple sweeps of the interval Gauss–Seidel method when C^W-preconditioners or C^M-preconditioners are used.

Even though the C^M-preconditioner is appropriate for existence/uniqueness verification, in many cases the somewhat simpler C^W-preconditioner is also appropriate. In particular, the C^M-preconditioner and C^W-preconditioner are approximately the same when the guess point $\check{X}$ is the center of the box $\boldsymbol{X}$ and when $F(\check{X}) \approx 0$:

Theorem 3.8 (first appeared in [128]) *Suppose $\tilde{x}_j$ is chosen to be the midpoint of x_j if $j > i$, and the midpoint of $\tilde{x}_j$ if $j \leq i$, and suppose $F(\check{X}) = 0$. Then the C^W-preconditioner and C^M-preconditioner for $\tilde{x}_i$ are the same.*

Theorem 3.8 says that, if the guess point $\check{X}$ in Formula 1.44 is a solution of $F(X) = 0$, and if the box $\boldsymbol{X}$ is constructed so that $\check{X}$ is the midpoint of $\boldsymbol{X}$, then the width-optimal preconditioner gives the preconditioner most likely to make the image $\tilde{x}_1$ under the Gauss–Seidel step be contained in the original box. Also, our computer codes for the width-optimal preconditioner with $\check{X}$ the midpoint of $\boldsymbol{X}$ are at present more efficient than corresponding computer codes for other preconditioners. Thus, the width-optimal preconditioner is a good preconditioner to use for existence verification or uniqueness verification in ϵ-inflation algorithms.

In practice, perhaps due to Theorem 3.8, instances are seldom found where the C^M-preconditioner gives improvement in an overall root-finding algorithm over the C^W-preconditioner.

The linear programming preconditioners described here are appropriate for the interval Gauss–Seidel method. Although they can also be used effectively with interval Gaussian elimination, analogous preconditioners can be derived specially for interval Gaussian elimination. Such preconditioners have not yet been investigated.

3.2.6 Exercises

1. Verify Lemma 3.4 from the definitions of the interval operations.
2. Verify Lemma 3.5.
3. Write an analogue of Lemma 3.5 under the assumption that
   (a) $\check{x}_j = \underline{x}_j$ for $1 \leq j \leq n$;
   (b) $\check{x}_j = \overline{x}_j$ for $1 \leq j \leq n$;
   (c) $n = 5$, $\check{x}_j = \underline{x}_j$ for $j = 1, 2$, $\check{x}_3 = \mathrm{m}(\boldsymbol{x}_3)$, and $\check{x}_j = \overline{x}_j$ for $j = 4, 5$.
4. State what expressions in Equation (3.25) correspond to y and z in Equation (7) of Lemma 3.4.
5. Prove Equation (3.27) and Equation (3.28).
6. Assume $0 \in \mathbf{n}_i(Y_i)$, and $M = |\mathbf{n}_i(Y_i)|$. Show that equations (3.39) hold.
7. Assume $0 \in \mathbf{n}_i(Y_i)$ and equations (3.39) hold. Assuming $\overline{s} = \max\{M, 0\}$ and $\underline{s} = \max\{-M, 0\}$, use Equation (4) of Lemma 3.4 to show that $M = |\mathbf{n}_i(Y_i)|$.
8. *(This one is somewhat lengthy.)* Check the computations in the three examples based on Example 3.1 on page 120 and the inverse-midpoint or width-optimal preconditioners. You may use the software described in §2.2 or alternate systems, such as described in §2.3 on page 102.

4

VERIFIED SOLUTION OF NONLINEAR SYSTEMS

Recall that a main goal of this book is solution of Problem 1.2, that is, finding all solutions of an equation $F(X) = 0$, $F : \mathbb{R}^n \to \mathbb{R}^n$ within a box $\boldsymbol{X} \in \mathbb{IR}^n$. In previous chapters, techniques useful in parts of the solution of this problem have been explained. Here, tessellation processes appear that combine these pieces into an overall algorithm. The output to this overall algorithm will be a list $\mathcal{R}$ of small boxes that have been verified to contain unique roots and a list $\mathcal{U}$ of small boxes that have neither been verified to contain roots nor verified to not contain roots[1].

Effective use of the verification tools presented in previous chapters depends on efficient tessellation of the original box $\boldsymbol{X}$ into sufficiently small boxes, and on ϵ-inflation. The tessellation is done with an exhaustive branch and bound search, using generalized bisection, and with a box complementation process. These processes are explained for general nonlinear equations in this chapter, while versions taking account of objective function inclusions for constrained and unconstrained global optimization are explained in Chapter 5. This chapter concludes with an explanation of the general nonlinear systems software within INTOPT_90.

Many of the ideas in this chapter appeared in [114]. The ϵ-inflation was explained in [208], while the box complementation scheme appeared in [245]. Also, interval global search algorithms generally predate the author's efforts in the subject. See [165, Ch. 6] or [175, §5.6] for nonlinear equations, and [77] or [191] for global optimization.

[1] In general, $\mathcal{U}$ is necessary, since roots at which the Jacobi matrix is singular cannot be verified to be unique, although boxes in which such roots exist can be verified. However, in successful algorithms, both the number and total volume of the boxes in $\mathcal{U}$ is as small as possible.

4.1 AN OVERALL BRANCH AND BOUND ALGORITHM

The branch and bound algorithm is presented here in summary. Specifics concerning bisection, box complementation, and ϵ-inflation appear in subsequent sections, while additional details appear in [114]. Also, the actual implementation in INTOPT_90 is available.

In what follows, $\|\mathrm{w}(\boldsymbol{X}^{(c)})\|_{\mathrm{rel}}$ will denote the relative diameter of the current box $\boldsymbol{X}^{(c)}$:

Definition 4.1 *The* relative diameter *of a box* $\boldsymbol{X}$ *is*

$$\|\mathrm{w}(\boldsymbol{X})\|_{\mathrm{rel}} = \max_{1 \le i \le n} \left\{ \frac{\mathrm{w}(x_i)}{\max\{1, |\mathrm{m}(x_i)|\}} \right\}.$$

The individual quotient $\mathrm{w}(x_i)/\max\{1, |\mathrm{m}(x_i)|\}$ *will be called the* relative width *of the* i*-th side.*

The relative diameter corresponds to the relative error norm in traditional floating point computations.

Algorithm 7 (Overall Tessellation / Complementation Process)

INPUT: an initial box $\boldsymbol{X}^{(0)}$, a symbolic representation for the function F, the maximum allowable number of subboxes M to be processed, and a domain tolerance ϵ_d.

OUTPUT: One of the following:

1. *If the search was successful:* a list $\mathcal{R}$ such that each box $\boldsymbol{X} \in \mathcal{R}$ has been verified to contain a unique root, and a list $\mathcal{U}$, each of whose boxes has relative diameter on the order of $\sqrt{\epsilon_d}$, such that all roots of F in $\boldsymbol{X}^{(0)}$ not in boxes in $\mathcal{R}$ are in boxes in $\mathcal{U}$.
2. *If the search did not complete with* M *boxes processed:* a list $\mathcal{R}$ as above, a list $\mathcal{U}$ of boxes with diameters on the order of $\sqrt{\epsilon_d}$ that may contain roots, and a list $\mathcal{L}$ of boxes that have not been fully analyzed, in the set complement of the union of the boxes in $\mathcal{R}$ and $\mathcal{U}$.

■ *Place* $\boldsymbol{X}^{(0)}$ *onto an empty list* $\mathcal{L}$.

- **Overall box processing loop:** DO $k = 1$ to M WHILE $\mathcal{L} \neq \emptyset$.

  1. *Remove the first box from $\mathcal{L}$ and place it in the current box $\boldsymbol{X}^{(c)}$.*
  2. (Check function and find as many approximate roots as possible first.)
     DO WHILE *this step changes $\mathcal{L}$, $\mathcal{R}$ or $\mathcal{U}$:*
     (a) Try to verify that $0 \notin \boldsymbol{F}^u(\boldsymbol{X}^{(c)})$, using first, then second order extensions. IF $0 \notin \boldsymbol{F}^u(\boldsymbol{X}^{(c)})$ *is verified* THEN CYCLE **overall box processing loop**.
     (b) (Find approximate roots in the current box.)
     IF $\|\mathrm{w}(\boldsymbol{X}^{(c)})\|_{\mathrm{rel}} > \epsilon_d$ THEN
     i. (Algorithm 9 below) *Depending on a heuristic parameter, try to find an approximate root $\check{X} \in \boldsymbol{X}^{(c)}$.* IF *such a root is found,*
     THEN
     A. Use ϵ-inflation to verify uniqueness of the root near $\check{X}$ within a box $\boldsymbol{X}^*$ (see §4.2 below).
     B. Place $\boldsymbol{X}^*$ in either $\mathcal{R}$ or $\mathcal{U}$, take the complement of $\boldsymbol{X}^*$ in $\mathcal{L}$ (see Algorithm 11 below), remove the new first box from the list $\mathcal{L}$, and place it into $\boldsymbol{X}^{(c)}$.
     END IF
     ii. IF *$\mathcal{L}$ is empty after complementation,* THEN EXIT **overall box processing loop**.
     END IF
     END DO
  3. *Reduce the widths of the current box with a combination of Gauss–Seidel iteration and generalized bisection. (Both $\boldsymbol{X}^{(c)}$ and $\mathcal{L}$ are in general changed by this step; see Algorithm 8 below.)*
  4. (After Step 3, the box $\boldsymbol{X}^{(c)}$ is either are proven to have no roots, or is small and of "unknown" status.)
     IF *the status of $\boldsymbol{X}^{(c)}$ is still "unknown,"*
     THEN
     (a) Try to verify, using first, then second-order extensions[2], *that* $0 \notin \boldsymbol{F}^u(\boldsymbol{X}^{(c)})$.
     IF $0 \notin \boldsymbol{F}^u(\boldsymbol{X}^{(c)})$ *is verified* THEN CYCLE **overall box processing loop**.
     (b) (Unconditionally attempt to find a root in the small box.)
     i. (Algorithm 9 below) *Try to find an approximate root $\check{X} \in \boldsymbol{X}^{(c)}$.*
     IF *such a root is found,*
     THEN
     A. Verify uniqueness of the root near $\check{X}$ within a box $\boldsymbol{X}^*$ with ϵ-inflation (see §4.2 below).

[2]The second-order extension is particularly important here, since the boxes are small, and the second-order extension is asymptotically better as the widths tend to zero.

B. Place $\boldsymbol{X}^*$ in either $\mathcal{R}$ of $\mathcal{U}$, take the complement of $\boldsymbol{X}^*$ in $\mathcal{L}$ (see Algorithm 11 below), remove the new first box from the list $\mathcal{L}$, and place it into $\boldsymbol{X}^{(c)}$.

END IF

ii. IF *$\mathcal{L}$ is empty after complementation,* THEN EXIT **overall box processing loop.**

(c) (Attempt to avoid clusters at singular or ill-conditioned (near) roots by artificial expansion.)

IF *Step 4b did not result any change in $\boldsymbol{X}^{(c)}$, $\mathcal{R}$ or $\mathcal{U}$,* THEN

i. For $i = 1$ to n replace $x_i^{(c)}$ by $[\underline{x}^{(c)} - \sigma_i, \overline{x}^{(c)} + \sigma_i]$ where $\sigma_i = \max\left\{|x_i^{(c)}|\sqrt{\epsilon_d}, \sqrt{\epsilon_m/\epsilon_d}\right\}$, where ϵ_m is the machine epsilon.

ii. Take the set complement of $\boldsymbol{X}^{(c)}$ in $\mathcal{L}$.

iii. Take the set complement of $\boldsymbol{X}^{(c)}$ in $\mathcal{U}$.

iv. Insert $\boldsymbol{X}^{(c)}$ into $\mathcal{U}$.

ELSE *Push $\boldsymbol{X}^{(c)}$ onto $\mathcal{L}$.*

END IF (cluster avoidance)

END IF (processing the small unknown box)

END DO **overall box processing loop**

- IF *k has exceeded* M *in the overall processing loop*
  THEN *Return $\mathcal{L}$, $\mathcal{R}$, and $\mathcal{U}$.*
  ELSE *Return $\mathcal{R}$, $\mathcal{U}$, and performance statistics.*

END IF

End Algorithm 7

Step 3 of Algorithm 7 combines generalized bisection (see §4.3.2 below) and interval Gauss–Seidel iteration: The input to Step 3 is a box $\boldsymbol{X}^{(c)}$ of relatively large diameter, while the output is a new box $\boldsymbol{X}^{(c)}$ of diameter at most ϵ_d. The process also includes checking $0 \notin \boldsymbol{F}(\boldsymbol{X}^{(c)})$, to avoid more expensive computations when not necessary. The process is:

Algorithm 8 (Process the current box $\boldsymbol{X}^{(c)}$ in Algorithm 7.)

INPUT: the current box $\boldsymbol{X}^{(c)}$, the internal symbolic representation for $\boldsymbol{F}$, and the current list of boxes to search $\mathcal{L}$.

OUTPUT: the following:

1. a new or altered box $\boldsymbol{X}^{(c)}$;
2. the status "unknown" or "has no root" associated with $\boldsymbol{X}^{(c)}$, such that, if the status of $\boldsymbol{X}^{(c)}$ is "unknown", then the maximum relative width of a coordinate of $\boldsymbol{X}^{(c)}$ is on the order of ϵ_d;
3. a (possibly) altered search list $\mathcal{L}$.

DO WHILE $\|\mathrm{w}(\boldsymbol{X}^{(c)})\|_{\mathrm{rel}} > \epsilon_d$:

1. *Compute the slope matrix* $\mathbf{S}(F, \boldsymbol{X}^{(c)}, \check{X}^{(c)})$, *where* $\check{X}^{(c)}$ *is the midpoint vector* $\mathrm{m}(\boldsymbol{X}^{(c)})$.
2. *Compute* $\boldsymbol{F}(\check{X}^{(c)})$ *(using interval arithmetic to bound roundoff errors).*
3. *Perform a Gauss–Seidel sweep, beginning with* $\boldsymbol{X}^{(c)}$.
4. IF *the Gauss–Seidel sweep proved that* $\boldsymbol{X}^{(c)}$ *could not contain any roots* THEN EXIT.
5. IF *the Gauss–Seidel sweep did not result in a change in* $\boldsymbol{X}^{(c)}$, $\mathcal{L}$, *or*[3] $\mathcal{U}$, THEN
   (a) *Bisect* $\boldsymbol{X}^{(c)}$, *modifying* $\boldsymbol{X}^{(c)}$ *and* $\mathcal{L}$ *(see §4.3.2).*
   (b) *Use the natural first-order extension to check if the range of* F *over the new current box* $\boldsymbol{X}^{(c)}$ *returned from bisection contains zero; if not, then mark* $\boldsymbol{X}^{(c)}$ *as not root-containing and* EXIT.
   (c) (Find approximate roots in the new current box.)
   Depending on the value of a heuristic parameter, find and verify approximate roots within $\boldsymbol{X}^{(c)}$, *and modify the list* $\mathcal{L}$, *the list* $\mathcal{R}$ *of root-containing boxes and the list* $\mathcal{U}$ *of small boxes of unknown status, using Algorithm 9 and Algorithm 11 below.*

   END IF (bisection process)

END DO

End Algorithm 8

In Step 2b of a Algorithm 7 and Step 5c of Algorithm 8, a heuristic is used to determine when to try to find a root within the current box. The heuristic is based on a parameter α between 0 and 1: $\alpha = 0$ implies that Algorithm 9 is always attempted, while $\alpha = 1$ implies Algorithm 9 is never attempted, except

[3]The interval Gauss–Seidel method can produce two boxes at each step if extended interval arithmetic is used. In such cases, one of the boxes is placed into $\mathcal{L}$, and the other one becomes $\boldsymbol{X}^{(c)}$, or else both boxes are placed into $\mathcal{L}$, and a third box from $\mathcal{L}$ becomes $\boldsymbol{X}^{(c)}$.

for starting points within boxes of diameters less than the domain tolerance. Approximate root-finding and verification are attempted within a box with relative diameter greater than ϵ_d, whenever

$$\min\{|\underline{F}_i|, |\overline{F}_i|\}/\max\{|\underline{F}_i|, |\overline{F}_i|\} > \alpha, \tag{4.1}$$

where $\boldsymbol{F}_i(\boldsymbol{X}^{(c)}) = [\underline{F}_i, \overline{F}_i]$. The presumption is that, when zero is centered in the interval estimate for the range, it is more likely that the actual range, without overestimation, contains zero. Here, $0 \leq \alpha \leq 1$; $\alpha = 0$ implies an attempt to find an approximate root is always made, while $\alpha = 1$ implies approximate root-finding is never attempted, except for starting points within boxes with relative diameters less than the domain tolerance ϵ_d. In a significant set of experiments reported in [114], $\alpha = 0.5$ was generally found to be a good choice, but the optimal α depends on the relative speeds of the floating point computations in the approximate solver and of the interval arithmetic[4], and probably also varies from problem to problem.

Existence and uniqueness verification are not attempted in Algorithm 8, whose main chore is to reduce the size of the current box $\boldsymbol{X}^{(c)}$ as efficiently as possible. Verification is usually easier when a root X^* is approximately centered within the box, while this does not usually occur while iterating the interval Gauss–Seidel method with a width-optimal preconditioner. Furthermore, attempting to prove existence or uniqueness within Algorithm 8 significantly complicates the bookkeeping in implementations.

4.2 APPROXIMATE ROOTS AND EPSILON-INFLATION

It improves the overall branch and bound algorithm to make use of approximate roots when they are available. In particular, if an approximate root $\check{X} \approx X^*$ is accurately known, then a box $\boldsymbol{X}^*$ can be *centered* at $\check{X}$, so that $\|F(\check{X})\|$ is small. The centering and small norm make existence or uniqueness verification easier; see Lemma 6.2 below on page 221. Alternately, if the Jacobi matrix is ill-conditioned or singular at the root X^*, floating point methods can often still obtain good approximations $\check{X} \approx X^*$, even though interval Newton methods cannot reduce all coordinate widths of any box containing X^*. In such cases, efficiency of the search process is increased if a box $\boldsymbol{X}^*$ containing X^* can be

[4]In [114], the system INTLIB_90 was used, with relative speeds given by Table 2.10 on page 105.

constructed about $\check{X}$, placed on the list $\mathcal{U}$ of possible root-containing boxes, and removed from the search region. These considerations are reflected in Algorithm 7 in steps 2b, 4b, and 4c.

The process of constructing a box $\boldsymbol{X}^*$ about an approximate root $\check{X}$, such that existence or uniqueness can be verified within $\boldsymbol{X}^*$, is termed *ϵ-inflation* [159, 205, 208]. It has been explained as an iterative process, starting with a small box about $\check{X}$, then increasing the widths until it is possible to verify existence or uniqueness. Once existence or uniqueness is verified, the box size may continue to be increased to obtain as large a box as possible within which uniqueness can be verified. This large box may then be removed from the search region, making it easier to eliminate the remaining boxes in the list $\mathcal{L}$ due to $0 \notin \boldsymbol{F}(\boldsymbol{X}^{(c)})$.

Uniqueness can be verified with slopes, based on Theorem 1.23 on page 64. This results in the following algorithm (an elaboration of an algorithm in Rump [208]).

Algorithm 9 (Find an approximate root, and verify uniqueness within as large a box as possible about that root.)

INPUT: the initial box (overall bounds) $\boldsymbol{X}^{(0)}$ and the current box $\boldsymbol{X}^{(c)} \subseteq \boldsymbol{X}^{(0)}$.

OUTPUT: one of the following:

1. an approximate root $\check{X} \in \boldsymbol{X}^{(c)}$ and a box $\boldsymbol{X}^* \subseteq \boldsymbol{X}^{(0)}$, as large as possible, $\check{X} \in \boldsymbol{X}^*$, such that it is proven F has a unique root in $\boldsymbol{X}^*$;
2. an approximate root $\check{X} \in \boldsymbol{X}^{(c)}$ and a box $\boldsymbol{X}^* \subseteq \boldsymbol{X}^{(0)}$, $\check{X} \in \boldsymbol{X}^*$, such that existence of a root within $\boldsymbol{X}^*$ has been proven;
3. an approximate root $\check{X}$ (according to the approximate solver), but no box $\boldsymbol{X}^*$;
4. failure to compute an approximate root.

1. (The "else" branch in the following is necessary because an approximate solver sometimes behaves erratically near roots[5].)
IF *the relative diameter of the coordinates of* $\boldsymbol{X}^{(c)}$ *is greater than* $\sqrt{\epsilon_d}$,
THEN

[5]when the tolerance is small

Use an approximate solver with domain tolerance[6] $\epsilon_d^{1.5}$ *to find an approximate root* $\check{X}$ *of* F *within* $\boldsymbol{X}^{(c)}$.

ELSE

Take the center of $\boldsymbol{X}^{(c)}$ *to be the approximate root.*

END IF

IF $\check{X} \notin \boldsymbol{X}^{(c)}$ *or no solution is found,* THEN EXIT.

2. (Verify existence in as small a box as possible.)

   (a) Construct a box $\check{\boldsymbol{X}}$*, centered at* $\check{X}$*, such that the* i*-th coordinate is*

   $$\check{\boldsymbol{x}}_i = [x_i^* - \epsilon_d s, x_i^* + \epsilon_d s],$$

   where $s = \max\{|x_i^*|, 100\epsilon_m\}$ *and* ϵ_m *is the machine epsilon.*

   (b) (Expand $\check{\boldsymbol{X}}$ one coordinate at a time until existence can be proven.)
   DO UNTIL *a larger box* $\check{\boldsymbol{X}} \in \boldsymbol{X}^{(0)}$ *cannot be constructed.*
   Attempt to verify existence within $\check{\boldsymbol{X}}$ *using a computed slope* $\mathbf{S}(F, \check{\boldsymbol{X}}, \check{X})$
   IF *existence was verified,*
   THEN EXIT *Step 2b.*
   ELSE
   i. IF *preconditioner computation was unsuccessful or the attempt to verify existence led to a disconnected image,* THEN EXIT *Step 2b without having verified existence*[7].
   ii. Expand $\check{\boldsymbol{X}} \in \boldsymbol{X}^{(0)}$ *around* $\check{X}$ *by widening one or more coordinates.*
   iii. If $\check{\boldsymbol{X}}$ *could not be expanded in the previous step, then* EXIT *Step 2b without having verified existence.*
   END IF
   END DO

   End existence verification

3. (Verify uniqueness within as large a box as possible: similar to Step 2b.) IF *existence was verified in Step 2b* THEN

   (a) Initialize to $\check{\boldsymbol{X}}$ *the box* $\boldsymbol{X}^*$ *in which to prove uniqueness.*

   (b) (Expand $\boldsymbol{X}^*$ as much as possible, subject to uniqueness verification.)
   DO UNTIL *a larger box* $\boldsymbol{X}^*$ *cannot be constructed.*
   Attempt to verify uniqueness within $\boldsymbol{X}^*$*, using a computed slope* $\mathbf{S}(F, \boldsymbol{X}^*, \check{\boldsymbol{X}})$.
   IF *uniqueness was verified on this iteration,* THEN

[6] This power of the minimum size of a side of a box, in terms of the scaled width $\tilde{\mathrm{w}}(\boldsymbol{x}) = (\overline{x} - \underline{x})/\max\{1, |\boldsymbol{x}|\}$; this ensures that the actual root will be near the center of the constructed boxes.

[7] Nonetheless, it is marked that an approximate root has been found. This fact can be used in an expansion step in the main algorithm.

i. Expand $\boldsymbol{X}^* \in \boldsymbol{X}^{(0)}$ *around* $\check{X}$ *by widening one or more coordinates.*

ii. IF $\boldsymbol{X}^*$ *could not be expanded* THEN EXIT *Step 3b.*

ELSE IF *uniqueness has not yet been verified on any iteration of Step 3b,* THEN

i. If preconditioner computation in the verification attempt was unsuccessful or the uniqueness verification attempt led to a disconnected image, THEN *exit Step 3b without having verified uniqueness.*

ii. Expand $\boldsymbol{X}^* \in \boldsymbol{X}^{(0)}$ *around* $\check{X}$ *by widening one or more coordinates.*

iii. IF $\boldsymbol{X}^*$ *could not be expanded,* THEN EXIT *without having proven uniqueness.*

ELSE (Here, uniqueness was proven with the previous box $\boldsymbol{X}^*$.)

i. Reset $\boldsymbol{X}^*$ *to its value before the last expansion attempt.*

ii. Possibly try to expand $\boldsymbol{X}^*$ *again, in a different coordinate direction than one previously found to lead to failure.*

iii. If $\boldsymbol{X}^*$ *could not be expanded in the new coordinate direction, then* EXIT, *having proven uniqueness within* $\boldsymbol{X}^*$.

END IF

END DO

END IF (uniqueness verification)

End Algorithm 9

The three expansion steps, in Step 2(b)ii and in the branches of Step 3b, are implemented in INTOPT_90 with a coordinate-by-coordinate procedure, in subroutine

INFLATE_DENSE_NLE_SLOPE.

A *minimum smear scheme,* similar to the maximum smear scheme in §4.3.2 below, is used. Alternate schemes are also possible, and more experimentation may reveal better ones.

The list $\mathcal{L}$ was effectively a last-in-first-out stack in the experiments in [114]. However, since the complement of the current list is taken when approximate roots are found, and since approximate roots may be found more rapidly if the region is searched in a certain order, it may be advantageous to consider $\mathcal{L}$ an ordered linked list. It is not clear, however, what the optimal orderings for solving nonlinear equations are[8].

[8]This is not so for global optimization: the list may be ordered in order of increasing lower bounds on the objective function. See Chapter 5.

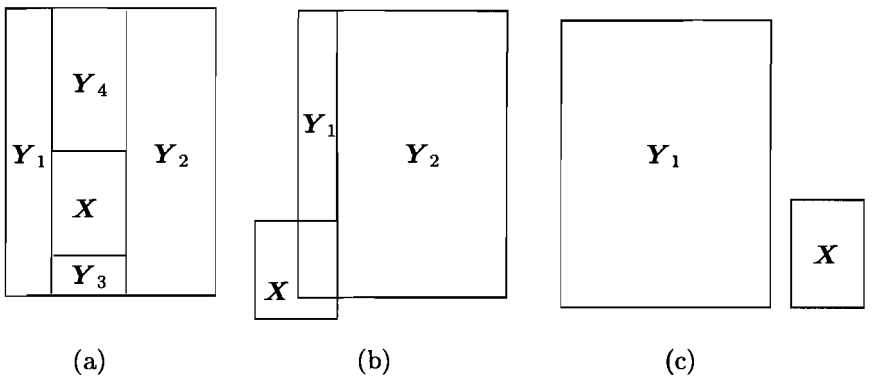


Figure 4.1 Complementation of a box in a box

4.3 TESSELLATION SCHEMES

Two processes, box complementation and generalized bisection, are important in Algorithm 7 (the global root-finding algorithm). The box complementation algorithm allows us to remove root-containing or difficult-to-analyze boxes from the search region, while generalized bisection, if done well, is effective at rapidly producing sub-regions in which either an interval Newton method will converge or non-existence can be proven. Although an unambiguous process, the power and simplicity of the box complementation algorithm is worthy of study. It is less clear how to proceed with generalized bisection, but various heuristics can be compared.

4.3.1 Box Complementation

Abstractly viewed, box complementation produces a list or stack of boxes whose union is the complement (in the set-theoretic sense) of a box $\boldsymbol{X}^*$ in the union of an original list of boxes. The following two algorithms were proposed in [245] in conjunction with the representation of one-dimensional solution manifolds associated with parametrized systems. They are related to the *trisection* algorithm proposed in [110]; there, boxes that contained roots at which the Jacobi matrix was singular were removed. Algorithms similar to those proposed here are also used by Ratz to remove singularities [194, §2.5.7] and by Van Iwaarden [233] (Van Iwaarden calls the ϵ-inflation technique "back-boxing.")

Algorithm 10 (Take the complement of a box $\boldsymbol{X}$ in a box $\boldsymbol{Y}$.)

INPUT: the boxes $\boldsymbol{X}$ and $\boldsymbol{Y}$.

OUTPUT: the following:

1. a list $\mathcal{L}_{\boldsymbol{Y}}$ such that $\bigcup_{\boldsymbol{W}\in\mathcal{L}_{\boldsymbol{Y}}}\boldsymbol{W} = \boldsymbol{X}\setminus\boldsymbol{Y}$;
2. $\boldsymbol{X} \leftarrow \boldsymbol{X}\cap\boldsymbol{Y}$, provided $\boldsymbol{X}\cap\boldsymbol{Y}\neq\emptyset$.

1. *Initialize a list $\mathcal{L}_{\boldsymbol{Y}}$ to $\emptyset$.*
2. IF $\boldsymbol{X}\cap\boldsymbol{Y}=\emptyset$

   THEN *Insert $\boldsymbol{Y}$ into $\mathcal{L}_{\boldsymbol{Y}}$.*

   ELSE

   (a) Do *for $i = 1$ to n*

   i. *Set $\boldsymbol{z} = [\underline{z}, \overline{z}] = \boldsymbol{x}_i \cap \boldsymbol{y}_i$.*

   ii. IF $\underline{z} > \underline{y}$ THEN

   A. *Form a new box $\boldsymbol{W}$ whose i-th coordinate is $[\underline{y}, \underline{z}]$ and whose other coordinates are the same as those of $\boldsymbol{Y}$.*

   B. *Insert $\boldsymbol{W}$ into $\mathcal{L}_{\boldsymbol{Y}}$.*

   END IF

   iii. IF $\overline{z} < \overline{y}$ THEN

   A. *Form a new box $\boldsymbol{W}$ whose i-th coordinate is $[\overline{z}, \overline{y}]$ and whose other coordinates are the same as those of $\boldsymbol{Y}$.*

   B. *Insert $\boldsymbol{W}$ into $\mathcal{L}_{\boldsymbol{Y}}$.*

   END IF

   iv. *Replace the i-th coordinate of $\boldsymbol{Y}$ by $\boldsymbol{z}$.*

   END DO

   (b) *Replace $\boldsymbol{X}$ by $\boldsymbol{Y}$.*

   END IF

End Algorithm 10

Possible ways that Algorithm 10 can cut a box $\boldsymbol{Y}$ are illustrated for $n = 2$ in Figure 4.1. In general, if $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{IR}^n$, at most $2n$ boxes are produced. This corresponds to case (a) of Figure 4.1, where $\boldsymbol{X} \subset \text{int}(\boldsymbol{Y})$, and this list can be produced in $\mathcal{O}(n)$ time. In steps 2(b)iB, 4(b)iB, and 4(c)i of Algorithm 7 (the overall search algorithm), the complement of a box in the union of a *list* of boxes must be taken. This is done by applying Algorithm 10 to each box in the list. This results in

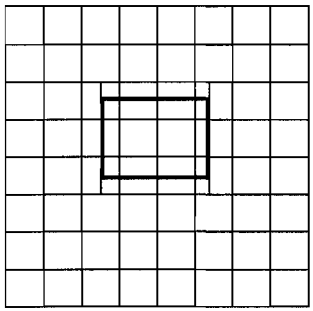


Figure 4.2 Complementation of a box in a list of boxes

Algorithm 11 (Take the complement of a box $\boldsymbol{X}$ in a list $\mathcal{L}$.)

INPUT: the box $\boldsymbol{X}$ and the list of boxes $\mathcal{L}$.

OUTPUT: The list $\mathcal{L}$ is replaced by that list the union of whose members is the set complement of $\boldsymbol{X}$ in the union of the boxes in $\mathcal{L}$.

1. *Initialize a new list* $\mathcal{L}_{\text{new}}$.
2. DO WHILE $\mathcal{L} \neq \emptyset$.
   (a) *Remove the first item* $\boldsymbol{Y}$ *from* $\mathcal{L}$ *and store it in* $\boldsymbol{Y}$.
   (b) *Form a list* $\mathcal{L}_{\boldsymbol{Y}}$ *from the complement of* $\boldsymbol{X}$ *in* $\boldsymbol{Y}$, *using Algorithm 10.*
   (c) *Append* $\mathcal{L}_{\boldsymbol{Y}}$ *to* $\mathcal{L}_{\text{new}}$, *and reinitialize* $\mathcal{L}_{\boldsymbol{Y}}$.

   END DO
3. *Replace* $\mathcal{L}$ *by* $\mathcal{L}_{\text{new}}$.

End Algorithm 11

Although it is possible for Algorithm 10 to produce $2n$ boxes for $\boldsymbol{Y} \setminus \boldsymbol{X}$, for given $\boldsymbol{X}$ and $\boldsymbol{Y}$, in general, Algorithm 11 does not always significantly increase the total number of boxes in $\mathcal{L}$. For example, in Figure 4.2, the original list $\mathcal{L}$ represents a uniform subdivision of an original box $\boldsymbol{X}^{(0)}$ into 64 subboxes. The list obtained by complementing with $\boldsymbol{X}$ only contains 66 boxes. Experiments with the overall root finding algorithm in [114] indicate that list complementation (Algorithm 11) is a net advantage with approximate roots. However, additional experience with the purely geometrical aspects should increase understanding of how complementation increases the total number of boxes in the list.

4.3.2 Generalized Bisection

As box complementation, generalized bisection operates on the list $\mathcal{L}$ of boxes within which all roots must lie. However, instead of eliminating regions from the search list, generalized bisection replaces a single box by two smaller boxes. This allows either the overestimation in interval extensions to be reduced, so $0 \notin \boldsymbol{F}^u(\boldsymbol{X})$ can be verified, or else allows consideration of smaller boxes in which an interval Newton method can converge.

Variants of bisection for nonlinear systems are discussed in [165, pp. 78–81], [77, §8.8], and for optimization in [20, 199]. Bisection usually proceeds with the following algorithm.

Algorithm 12 (Generalized bisection of a box $\boldsymbol{X}$)

INPUT: The current box $\boldsymbol{X}$ and the list $\mathcal{L}$.

OUTPUT: A new current box $\boldsymbol{X}$ and an updated list $\mathcal{L}$.

1. *Choose the coordinate k to bisect.*
2. *Form two new boxes $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$, where $\boldsymbol{x}_j^{(p)} = \boldsymbol{x}_j$ if $j \neq k$,*

$$\boldsymbol{x}_k^{(1)} = [\underline{x}_k, (\underline{x}_k + \overline{x}_k)/2] \quad \text{and} \quad \boldsymbol{x}_k^{(2)} = [(\underline{x}_k + \overline{x}_k)/2, \overline{x}_k].$$

3. *Place one of $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$ onto $\mathcal{L}$, and replace $\boldsymbol{X}$ with the other one*[9].

End Algorithm 12

Four proposed choices of bisection coordinate k were given in [165, pp. 78–81]. A natural choice would be a k for which $\mathrm{w}(\boldsymbol{x}_k)$ is maximum. However, k corresponding to maximum width (or even maximum relative width) is not always most effective at reducing overestimation or producing boxes in which an interval Newton method will converge. The *maximum smear* heuristic, proposed in [127], appears to work better in most cases.

Definition 4.2 *A coordinate of* maximum smear *is defined to be a k such that $s_k = \max_{1 \leq j \leq n} s_j$, where*

$$s_j = \max_{1 \leq i \leq n} \{|\boldsymbol{A}_{i,j}|\} \, \mathrm{w}(\boldsymbol{x}_j). \tag{4.2}$$

[9] or else place both $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$ into the list, then replace $\boldsymbol{X}$ by a third box from the list.

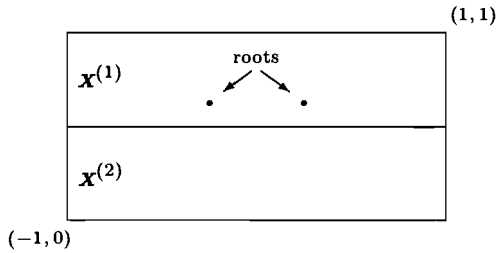


Figure 4.3 Bisection of the second coordinate according to maximum smear, for Example 4.1

Example 4.1 *Suppose* $F(X) = (f_1(X), f_2(X))^T : \mathbb{R}^2 \to \mathbb{R}^2$ *with*

$$f_1(x_1, x_2) = (4x_1^2 - 1)(10x_2^5 + 1), \quad f_2(x_1, x_2) = (4x_1^2 + 1)(10x_2^5 - 1),$$

and consider an initial box $\boldsymbol{X} = ([-1, 1], [0, 1])^T$. *Then the set of roots of* F *is*

$$\{(-1/4, -1/10^{0.2}), (1/4, -1/10^{0.2})\} \approx \{(-0.25, -0.63), (0.25, -0.63)\},$$

and both roots are in the initial box.

The interval Jacobi matrix in Example 4.1 is

$$\boldsymbol{F}'(\boldsymbol{X}) = \begin{pmatrix} [-88, 88] & [-50, 150] \\ [-72, 72] & [0, 250] \end{pmatrix}.$$

Following Formula (4.2), $s_1 = (88)(2) = 176$, while $s_2 = (250)(1) = 250$, $k = 2$, and $\boldsymbol{X}$ is bisected in the direction of its second coordinate. This is illustrated in Figure 4.3.

The maximum smear heuristic is similar to the heuristic proposed in [77, §8.8], except that "$\sum_{i=1}^{n}$" is used in [77, §8.8] in place of the "$\max_{1 \le i \le n}$" of (4.2). Both heuristics attempt to choose that coordinate k such that the function components vary most across $\boldsymbol{x}_k$; $\boldsymbol{X}_k$ is, roughly speaking, a maximum-width coordinate in the *range*.

A careful study of heuristics for bisection in a simplified verified global optimization algorithm appears in [199]. There, experimental results seemed to indicate that an analogue of maximum smear, applied to the *objective function*, leads to the best results in the overall search algorithm. That is, in global optimization codes, the coordinate k for which

$$\left| \frac{\partial \phi}{\partial x_k}(\boldsymbol{X}) \right| \mathrm{w}(\boldsymbol{x}_k) = \max_{1 \le i \le n} \left| \frac{\partial \phi}{\partial x_i}(\boldsymbol{X}) \right| \mathrm{w}(\boldsymbol{x}_i) \tag{4.3}$$

appears to be a good choice.

Whether to choose $\boldsymbol{X} \leftarrow \boldsymbol{X}^{(1)}$ or $\boldsymbol{X} \leftarrow \boldsymbol{X}^{(2)}$ in Step 3 of Algorithm 12 was an issue in [165, pp. 78–81], since, unlike in Algorithm 7, the goal in [165] was to find, as fast as possible, *at least one* box within which Newton's method or an interval Newton method would converge. Thus, a good heuristic would make the box most likely to contain a root into the current box $\boldsymbol{X}$. If *all* roots are sought, as in Algorithm 12, then the choice is not as critical. However, since root-containing regions are deleted once a root is found, there still may be an advantage in Algorithm 7 to choosing the new $\boldsymbol{X}$ to be that box most likely to contain a root.

In algorithms for optimization corresponding to Algorithm 7, *both* $\boldsymbol{X}^{(1)}$ and $\boldsymbol{X}^{(2)}$ are inserted into a list that is ordered according to increasing $\underline{\phi}(\boldsymbol{X})$, then the first box in this list (possibly neither $\boldsymbol{X}^{(1)}$ nor $\boldsymbol{X}^{(2)}$), becomes the new current box. Such a scheme would be appropriate in nonlinear equations if the list were somehow ordered according to increasing likelihood of containing roots.

4.4 DESCRIPTION OF PROVIDED SOFTWARE

A subroutine

`ROOTS_DELETE(INITIAL_X, ROOT_LIST, UNKNOWN_LIST)`

in `INTOPT_90` implements Algorithm 7. The program

`RUN_ROOTS_DELETE`

provides input and output, and can either be used as-is to solve nonlinear systems, or as a template for calling `ROOTS_DELETE`. In `RUN_ROOTS_DELETE`, a code list produced by a program such as those in Figures 2.4, 2.7, or 2.8 of §2.2.2 is input, as well as a file containing a tolerance, initial box coordinates, and, optionally, an initial guess for a root and an assessment of how good that initial guess is. Additionally, `RUN_ROOTS_DELETE` uses the configuration file `OVERLOAD.CFG` as in Figure 2.6. It is important that the code list used by `RUN_ROOTS_DELETE` was created with the same value of the logical variable `BINARY_CODELIST` (in `OVERLOAD.CFG`) as is used when the code list is input to `RUN_ROOTS_DELETE`.

Here, use of `RUN_ROOTS_DELETE` as a stand-alone program is first explained, then use of `ROOTS_DELETE`.

4.4.1 Stand-Alone Computation

The stand-alone program RUN_ROOTS_DELETE uses a second configuration file

ROOTSDL.CFG

that contains options for the root-finder. The first line of ROOTSDL.CFG is a comment line that is ignored by the program, while the second line[10] contains the following values, in this order:

EPS_DOMAIN: the domain tolerance ϵ_d.

PROOTSDL: a parameter specifying the amount of printing in ROOTS_DELETE. A value of 0 indicates no printing, while a value of 5 indicates a large amount of information (for debugging, etc.).

PRINT_INFLATE: a parameter specifying the amount of printing in the ϵ-inflation process (Algorithm 9). A value of 0 indicates no printing, while a value of 4 means a large amount of information is printed.

DO_INTERVAL_NEWTON is normally set to "T". If it is set to "F", then Steps 1 to 4 of Algorithm 8 are not done. That is, if DO_INTERVAL_NEWTON = "F", then interval Newton steps are not done except in ϵ-inflation, so that the only means of reducing the size of a box is bisection.

SLOPE_NOT_JACOBI: is set to "T" if slope matrices are to be used, and to "F" if interval Jacobi matrices are to be used.

SECOND_ORDER: is set to "T" if second-order interval extensions are to be used to try to determine $0 \notin \boldsymbol{F}^u(\boldsymbol{X}^{(c)})$ after first-order extensions have failed, and is set to "F" otherwise[11].

REFINEMENT_IN_VERIFICATION: is set to "T" if an interval Newton method is to be applied to the small boxes in Algorithm 9 that have been proven to contain roots, and is set to "F" otherwise.

MAXITR: is an upper bound on the number of boxes that should be processed by the overall search algorithm (i.e. the number M in Algorithm 7).

USE_SUBSIT: is set to "T" if the algorithm is to compute with user-defined constraints, as explained in Chapter 7 below, and is set to "F" otherwise.

[10] and/or subsequent lines: The parameters in this file are input in list-directed format on one or more lines.

[11] However, second-order extensions are *always* tried on boxes of small diameter of unknown status, in Step 4a of Algorithm 7.

```
PROGRAM BRANIN
USE OVERLOAD
TYPE(CDLVAR), DIMENSION(2) ::  X
TYPE(CDLLHS), DIMENSION(2) ::  F
TYPE (CDLVAR) ::  FX1PX2
OUTPUT_FILE_NAME='BRANIN.CDL'
CALL INITIALIZE_CODELIST(X)
FX1PX2 = 4*(X(1)+X(2))
F(1) = FX1PX2
F(2) = FX1PX2 + (X(1)-X(2)) * ( (X(1)-2)**2 + X(2)**2-1 )
CALL FINISH_CODELIST
END PROGRAM BRANIN
```

Figure 4.4 Program to produce a code list for a counterexample to a method of Branin

PRINT_SUBSIT: controls the amount of printing in computations with user-defined constraints, if such computations are done, with a value of 0 indicating no printing, and larger values indicating more printing.

GOU, or *general output unit*, is the Fortran unit to which most printing is to be done.

PRINT_LENGTH: indicates how many digits are to be printed. A value of 1 indicates four digits, while a value of 2 indicates eighteen digits[12].

The parameters define algorithm options studied in [114]. For the most part, users will only be interested in adjusting EPS_DOMAIN, MAXITR, and PRINT_LENGTH, while the other parameters are usually best as in the ROOTSDL.CF file that comes with the package.

For example, suppose a code list were produced by running the program in Figure 4.4, with OVERLOAD.CFG file:

```
NEQMAX, NROWMAX, NCONSTMX, BRANCHMX, BINARY_CODELIST
100 10000 2000 100 F
```

so that an ASCII code list is placed in the file BRANIN.CDL. Suppose ROOTSDL.CFG has all print control variables set to zero, and PRINT_LENGTH set to 1. Suppose a box data file appears[13] as in Figure 4.5. Then a sample invocation of RUN_ROOTS_DELETE for the file in BRANIN.CDL appears in Figure 4.6.

[12]Most printing of floating point and interval values is done with a generic subroutine PRINT_VECTOR defined in a module PRINT_ROUTINES.

[13]The first line, a tolerance, is not used in RUN_ROOTS_DELETE, but is included so files in INTOPT_90 have a universal format for several purposes.

```
1d-9
-2 2
-2 2
```

Figure 4.5 Box data file BRANIN.DT1

```
interval% run_roots_delete
Input the code list file name without its assumed suffix "CDL"
BRANIN
Input a digit for the suffix for the input file; the prefix
is the same as the code list file name and the first two
letters of the suffix are "DT"
1
Input the heuristic parameter alpha for determining
when to try to find a root in a box, based on the
interval function values.
.5
interval%
```

Figure 4.6 RUN_ROOTS_DELETE for BRANIN.CDL and BRANIN.DT1

The output corresponding to Figure 4.6 is written to the file BRANIN.RO1, exhibited in Figure 4.7.

The data file in Figure 4.5 does not contain an initial guess for a root, or an assessment of the quality of that initial guess. However, if we suspect that $X = (0,0)^T$ is a root, then we may form a data file as in Figure 4.8, and use RUN_ROOTS_DELETE to verify it. The output, in file BRANIN.RO2, is then as in Figure 4.9.

4.4.2 Calling from Other Programs

The subroutine ROOTS_DELETE may be called as a subroutine. The arguments are as follows.

INITIAL_X is an interval vector. On entry, it must contain the coordinates of the search box.

ROOT_LIST is a variable of type DENSE_NLE_BOX_LIST (explained below) that will contain the list of boxes that have been verified to contain unique roots (the list $\mathcal{R}$ in Algorithm 7), upon return.

UNKNOWN_LIST is a variable of type DENSE_NLE_BOX_LIST that will contain the list of small boxes that could not be verified to contain unique roots, but could not be verified to not contain roots (the list $\mathcal{U}$ in Algorithm 7), upon return.

```
Output from RUN_ROOTS_DELETE on: 19950917 225532.877
Codelist file name is: BRANIN.CDL
Box data file name is: BRANIN.DT1
Heuristic parameter alpha: 0.5000000000000000
Singular expansion factor: 1.0000000000000000E+03

Configuration settings:
print_roots_delete: 0
print_inflate: 0
slope_not_jacobi: T
use_second_order: T
refinement_in_verification: F
VERY_GOOD_INITIAL_GUESS: F
USE_SUBSIT: F

Initial box coordinates:
-0.2000D+01 0.2000D+01 -0.2000D+01 0.2000D+01


EPS_DOMAIN: 1.0000000000000002E-06
EPS_CHECK: 5.0000000000000003E-02

Solver statistics
Number of bisections: 10
Number of dense interval residual evaluations: 51
Total number of boxes pushed on the list: 14
Number of orig. system inverse midpoint preconditioner rows: 19
Number of orig. system C-LP preconditioner rows: 34
Total number of forward_substitutions: 247
Number of Gauss-Seidel steps on the dense system: 53
Number point dense residual evaluations: 19
Total number of dense slope matrix evaluations: 46
Total number second-order interval evaluations of the
original function: 12
Number of times Epsilon-inflation was attempted: 1

CPU time: 9.9999999627470970E-02
Time in approximate solver: 3.0000004917383194E-02
Time in point function: 0.0000000000000000E+000
Time in SUBSIT: 0.0000000000000000E+000

The following boxes have been verified to contain unique roots:

Box no.: 1
Box coordinates:
-0.3822D-02 0.3822D-02 -0.3822D-02 0.3822D-02

Level: 0
Box contains the following approximate root:
0.0000D+00 0.0000D+00
Interval residuals over the box:
-0.3058D-01 0.3058D-01 -0.5363D-01 0.5363D-01

Unknown = F Contains_root = T
Changed coordinates:
T T
--------------------

THERE WERE NO UNRESOLVED BOXES
```

Figure 4.7 Sample output to `RUN_ROOTS_DELETE`

```
1d-9
-.1 .1
-.1 .1
0
0
T
```

Figure 4.8 Box data file `BRANIN.DT2`

```
Output from RUN_ROOTS_DELETE on: 19950918 002244.606
Codelist file name is: BRANIN.CDL
Box data file name is: BRANIN.DT2
Heuristic parameter alpha: 0.5000000000000000
Singular expansion factor: 1.0000000000000000E+03

Configuration settings:
print_roots_delete: 0
print_inflate: 0
slope_not_jacobi: T
use_second_order: T
refinement_in_verification: F
VERY_GOOD_INITIAL_GUESS: T
USE_SUBSIT: F

Initial box coordinates:
-0.1000D+00 0.1000D+00 -0.1000D+00 0.1000D+00


EPS_DOMAIN: 1.0000000000000002E-06
EPS_CHECK: 5.0000000000000003E-02

Solver statistics
Number of dense interval residual evaluations: 2
Number of orig. system inverse midpoint preconditioner rows: 20
Total number of forward_substitutions: 42
Number of Gauss-Seidel steps on the dense system: 20
Total number of dense slope matrix evaluations: 10
Number of times Epsilon-inflation was attempted: 1

CPU time: 1.9999999552965164E-02
Time in approximate solver: 0.0000000000000000E+000
Time in point function: 0.0000000000000000E+000
Time in SUBSIT: 0.0000000000000000E+000

The following boxes have been verified to contain unique roots:

Box no.: 1
Box coordinates:
-0.1000D+00 0.1000D+00 -0.1000D+00 0.1000D+00

Level: 0
Box contains the following approximate root:
0.0000D+00 0.0000D+00
Interval residuals over the box:
-0.8000D+00 0.8000D+00 -0.1484D+01 0.1484D+01

Unknown = F Contains_root = T
Changed coordinates:
T T
--------------------------

THERE WERE NO UNRESOLVED BOXES
```

Figure 4.9 Use of `RUN_ROOTS_DELETE` for uniqueness verification

The variable type DENSE_NLE_BOX_LIST is defined and supported in the module

DENSE_NONLINEAR_EQ_BOX_LIST,

a part of INTOPT_90. These lists are dynamically allocated linked lists that must be *initialized* before being passed to ROOTS_DELETE. The initialization is as follows:

```
ROOT_LIST = NEW_DENSE_NLE_BOX_LIST()
UNKNOWN_LIST = NEW_DENSE_NLE_BOX_LIST()
```

To properly recover storage from an empty list, the statements

```
CALL DISCARD_EMPTY_LIST(UNKNOWN_LIST)
CALL DISCARD_EMPTY_LIST(ROOT_LIST)
```

are used. The lists are actually lists of variables of data type DENSE_NLE_BOX, with structure as follows:

```
TYPE DENSE_NLE_BOX
   TYPE (INTERVAL), DIMENSION(:), POINTER:: X
   INTEGER LEVEL
   LOGICAL CONTAINS_APPROX_ROOT
   DOUBLE PRECISION, DIMENSION(:), POINTER:: APPROX_ROOT
   TYPE(INTERVAL), DIMENSION(:), POINTER :: RESIDUALS
   LOGICAL UNKNOWN, CONTAINS_ROOT
   LOGICAL, DIMENSION(:), POINTER:: CHANGED_COORDINATES
   INTEGER, DIMENSION(:), POINTER :: CHANGED_COORDINATE_INDICES
   INTEGER NUMBER_OF_CHANGED_COORDINATES
END TYPE DENSE_NLE_BOX
```

Within this structure,

X contains the box coordinates,

CONTAINS_APPROX_ROOT is set to "true" if the approximate root-finder converged to a point within X,

APPROX_ROOT contains the coordinates of the approximate root, if one has been found,

RESIDUALS contains interval function values $\boldsymbol{F}(\boldsymbol{X})$, i.e. bounds on the range of F over X.

UNKNOWN is set to "true" if uniqueness could not be proven, and is set to "false" otherwise,

CONTAINS_ROOT is set to "true" if existence was verified in the ϵ-inflation process, and is set to "false" otherwise.

Besides initialization, the module DENSE_NONLINEAR_EQ_BOX_LIST contains the following support functions, given generic names in the module GENERIC_LIST. Here, L is of type DENSE_NLE_BOX_LIST, and B is of type DENSE_NLE_BOX.

SUBROUTINE INSERT(L, B) inserts the box B into the list L.

FUNCTION EMPTY(L) returns"true" if and only if L is empty.

SUBROUTINE REMOVE_FIRST_ITEM(L,B) removes the first box from L and places it in B.

SUBROUTINE GET_FIRST_ITEM(L,B) places the first box from L into B without removing it from L.

SUBROUTINE PRINT_LIST(L) prints the list L.

In using ROOTS_DELETE as a subroutine, the parameters that RUN_ROOTS_DELETE normally inputs from the file ROOTSDL.CFG should be set. It is safest to use RUN_ROOTS_DELETE as a template.

4.4.3 Installation

The driver RUN_ROOTS_DELETE and the subroutine ROOTS_DELETE are available with the entire package INTOPT_90, including INTLIB_90 and INTLIB, from the FTP site

interval.usl.edu

in the directory

pub/interval_math/Fortran_90_software/INTOPT_90

in the Unix compressed "tar" file

INTOPT90.tar.Z

When INTOPT90.tar.Z is uncompressed and extracted, the portable Fortran 90 code will be in the directories

access,	augsys,	default_solvers,	examples,	f90intbi,
function,	intlib.alt,	iterate,	linpack.alt,	listops,
matrixop,	overload,	precond,	splines,	symbolic.

There is an overall `makefile`, and there are `makefiles` for each directory. Operating system-dependent commands, such as directory names, the command for invoking the `Fortran-90` compiler, etc., are singled out as variables. There is a `READ.ME` file with installation instructions in the root directory.

The author should be consulted if it is necessary to obtain `INTOPT_90` in another form.

4.5 ALTERNATE ALGORITHMS AND IMPROVEMENTS

The subroutine `ROOTS_DELETE` makes use only of the C^W-preconditioner, which practical experience seems to indicate is the best general-purpose preconditioner[14]. It is not so clear when splitting preconditioners can be used effectively, but the E^M-preconditioner holds some promise. In particular, splitting preconditioners should be effective for boxes that lie between roots. The heuristic (4.1), used to decide if it is likely that a box contains a root, might be used effectively in reverse. That is, an E^M-preconditioner may be effective in proving no roots in $\boldsymbol{X}^{(c)}$ whenever

$$\min\{|\underline{F}_i|, |\overline{F}_i|\}/\max\{|\underline{F}_i|, |\overline{F}_i|\} < \beta, \tag{4.4}$$

for some β between 0 and 1 (say, $\beta = .25$).

[14] Also, we have an implementation of the simplex method, due to Manuel Novoa, that has been optimized for width-optimal contraction preconditioner computations.

5

OPTIMIZATION

Here, the techniques explained so far will be used to construct solution algorithms to Problem (1.3), that is, to the problem

Given $\phi : \boldsymbol{X} \to \mathbb{R}$ and constraints

$$C(X) = (c_1(X), \ldots c_m(X))^T : \mathbb{R}^n \to \mathbb{R}^m,$$

rigorously find upper and lower bounds to the values of ϕ that solve

$$\begin{array}{lllll} \text{minimize} & \phi(X) & & & \\ \text{subject to} & c_i(X) & = & 0, & i = 1, \ldots, m, \\ & \underline{x}_{i_j} \leq & x_{i_j}, & & j = 1, \ldots q - \mu, \\ & x_{i_j} \leq & \overline{x}_{i_j}, & & j = \mu + 1, \ldots q, \end{array} \tag{1.3}$$

and, for each minimizer $X^* \in \boldsymbol{X}$, find bounds $a_i \leq x_i^* \leq b_i$ such that

- $b_i - a_i$ is small, $1 \leq i \leq n$, and
- it is mathematically but automatically proven that there is a unique critical point of ϕ within each

$$\check{\boldsymbol{X}} = \{X = (x_1, \ldots, x_n) \mid a_i \leq x_i \leq b_i, 1 \leq i \leq n\}.$$

In Problem (1.3), inequality constraints $g_j(X) \leq 0$, $1 \leq j \leq \mu$ may also be included by adding slack variables: $c_{m+j}(X) = g_j(X) + x_{n+j}$, $x_{n+j} \geq 0$. Also,

the box $\boldsymbol{X} = (x_1, \ldots, x_n)$ defines an artificial limitation on the search region, while $\underline{x}_{i_j} \leq x_{i_j} \leq \overline{x}_{i_j}$ represent actual bound constraints for a subset of the coordinates. In fact, if actual bound constraints exist, it can be advantageous to treat bound constraints separately from the inequality and equality constraints.

Many of the same tools as those in Chapter 4 can be used. In principle, global optima can be found by computing roots of the Fritz John conditions presented in §5.2.5 below. (Such roots represent critical points of the constrained problem.) However, merely finding critical points ignores much important elementary and available structure, namely rigorous bounds on the values of the objective function ϕ. In contrast to nonlinear equations algorithms such as Algorithm 7 on page 146, global optimization algorithms benefit greatly from ordering the list $\mathcal{L}$ of boxes to be considered, and from deleting boxes $\boldsymbol{X}$ from $\mathcal{L}$ for which a rigorous lower bound $\underline{\phi}(\boldsymbol{X})$ over $\boldsymbol{X}$ is greater than a rigorous upper bound for an actually computed value $\phi(\check{X})$. Inclusion of these and other techniques makes a branch and bound algorithm for global optimization significantly more powerful and practical than nonlinear equations solvers applied to the same problems.

A significant amount of work has already appeared in interval methods in global optimization. The book by Ratschek and Rokne [191] gives an overview of the basic techniques. The more recent book of Hansen [77] also gives a comprehensive overview, as well as descriptions of other innovative techniques of the author. A number of researchers, including Hansen [77] and Ratz [194] have developed computer codes for rigorous global optimization, and have reported the results of numerical experiments, including, more recently, work on parallelization. However, most such experimental work has reported only results for unconstrained problems.

General techniques and work to date will first be reviewed in §5.1. Section 5.2 then describes methods for handling constraints, including some proposed new techniques. The software available in `INTOPT_90` is described in §5.3.

5.1 BACKGROUND AND HISTORICAL ALGORITHMS

We begin by reviewing the overall structure in global search algorithms for optimization.

Algorithm 13 (Abstract Branch-and-Bound Pattern for Optimization)

INPUT: an initial box $\boldsymbol{X}_0$.

OUTPUT: a list $\mathcal{C}$ of boxes that have been proven to contain critical points and a list $\mathcal{U}$ of boxes with small objective function values, but which could not otherwise be resolved.

1. *Initialize a list of boxes $\mathcal{L}$ by placing the initial search region $\boldsymbol{X}_0$ in $\mathcal{L}$.*
2. DO WHILE $\mathcal{L} \neq \emptyset$.
   (a) *Remove the first*[1] *box $\boldsymbol{X}$ from $\mathcal{L}$;*
   (b) (Process $\boldsymbol{X}$) *Do one of the following:*
   - *reject $\boldsymbol{X}$;*
   - *reduce the size of $\boldsymbol{X}$;*
   - *determine that $\boldsymbol{X}$ contains a unique critical point, then find the critical point to high accuracy;*
   - *subdivide $\boldsymbol{X}$ to make it more likely to succeed at rejecting, reducing, or verifying uniqueness.*

   (c) *Insert one or more boxes derived from $\boldsymbol{X}$ onto $\mathcal{L}$, $\mathcal{U}$ or $\mathcal{C}$, depending on the size of the resulting box(es) from Step 2b and whether the (possible) computational existence test in that step has determined a unique critical point.*

   END DO

End Algorithm 13

Algorithm 13 represents a general description, and many details, such as stopping criteria and tolerances, are absent. Such details differ in particular actual algorithms.

A combination of several techniques is used in state-of-the-art interval global optimization codes for Step 2b of Algorithm 13. Some of these techniques are outlined as the following algorithm.

[1]The boxes in $\mathcal{L}$ are in general inserted in a particular order, depending on the actual algorithm.

Algorithm 14 (Range Check and Critical Point Verification)

INPUT: a box $\boldsymbol{X}$ and a current rigorous upper bound $\overline{\phi}$ on the global minimum.

OUTPUT: one or more boxes derived from $\boldsymbol{X}$, or the information that $\boldsymbol{X}$ cannot contain a global minimum, or information that $\boldsymbol{X}$ contains a critical point.

1. (Feasibility check; for constrained problems only)
   (a) (Exit if infeasibility is proven) DO *for* $i = 1$ *to* m:
      i. *Compute an enclosure* $c_i(\boldsymbol{X})$ *for the range of* c_i *over* $\boldsymbol{X}$.
      ii. IF $0 \notin c_i(\boldsymbol{X})$ THEN *discard* $\boldsymbol{X}$ *and* EXIT.
   (b) *Verify, if possible, that there exists at least one feasible point in* $\boldsymbol{X}$.
2. (Range check or "midpoint test")
   (a) *Compute a lower bound* $\underline{\phi}(\boldsymbol{X})$ *on the range of* ϕ *over* $\boldsymbol{X}$.
   (b) IF $\underline{\phi}(\boldsymbol{X}) > \overline{\phi}$ THEN *discard* $\boldsymbol{X}$ *and* EXIT.
3. (Update the upper bound on the minimum.) IF *the problem is unconstrained or feasibility was proven in Step 1b,* THEN
   (a) *Use interval arithmetic to compute an upper bound* $\overline{\phi}(\boldsymbol{X})$ *of the objective function* ϕ *over* $\boldsymbol{X}$.
   (b) $\overline{\phi} \leftarrow \min\{\overline{\phi}, \overline{\phi}(\boldsymbol{X})\}$.
4. ("monotonicity test")
   (a) *Compute an enclosure* $\nabla\phi(\boldsymbol{X})$ *of the range of* $\nabla\phi$ *over* $\boldsymbol{X}$. *(Note: If* $\boldsymbol{X}$ *is "thin", i.e. if some bound constraints are active over* $\boldsymbol{X}$*, then a* reduced gradient *can be used; see §5.2.3 and [113].)*
   (b) IF $0 \notin \nabla\phi(\boldsymbol{X})$ THEN *discard* $\boldsymbol{X}$ *and* EXIT.
5. ("concavity test") *If the Hessian matrix*[2] $\nabla^2\phi$ *cannot be positive definite anywhere in* $\boldsymbol{X}$ THEN *discard* $\boldsymbol{X}$ *and* EXIT.
6. (Quadratic convergence and computational existence/uniqueness) *Use an interval Newton method*[3] *(with the Fritz John equations as in §5.2.5 in the constrained case) to possibly do one or more of the following:*
   - *reduce the size of* $\boldsymbol{X}$;
   - *discard* $\boldsymbol{X}$;
   - *verify that a unique critical point exists in* $\boldsymbol{X}$.
7. (Bisection or geometric tessellation) *If Step 6 did not result in a sufficient change in* $\boldsymbol{X}$*, then bisect* $\boldsymbol{X}$ *along a coordinate direction (or otherwise tessellate* $\boldsymbol{X}$*), returning all resulting boxes for subsequent processing.*

End Algorithm 14

[2] or reduced Hessian matrix, as in Step 4

[3] possibly in a subspace, as in steps 4 and 5

Since techniques for constrained problems are somewhat more involved, Step 1, checking for infeasibility and verifying existence of a feasible point, will be explained separately in §5.2.1.

Step 2 is called the "midpoint test" because the upper bound $\overline{\phi}$ is often obtained by evaluating ϕ at the midpoint vector of $\boldsymbol{X}$, properly taking account of rounding errors for rigor. Of course, Step 4 is called the "monotonicity test" since ϕ is monotone in the i-th variable over $\boldsymbol{X}$ if the i-th component of $\nabla\phi$ does not vanish over $\boldsymbol{X}$.

Improved techniques for carrying out Step 5, checking non-convexity, are desirable. Presently, sufficient conditions, such as checking the sign of the diagonal entries of an interval evaluation $\nabla^2\phi(\boldsymbol{X})$, can be used. One method of verifying convexity appears as Theorem 14.1 in [70] and Lemma 2.7.2 in [194]. Also, Neumaier [177] has shown that every element matrix of an interval matrix $\boldsymbol{A}$ is positive-definite, provided some point matrix $A \in \boldsymbol{A}$ is positive definite and $\boldsymbol{A}$ is regular. This result can be sharpened as follows.

Theorem 5.1 (Shi, [222, Appendix B]) *Suppose $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ is an n by n interval matrix. If $A \in \boldsymbol{A}$ is symmetric, A has p negative eigenvalues, and $\boldsymbol{A}$ is regular, then every symmetric point matrix in $\boldsymbol{A}$ has p negative eigenvalues.*

Theorem 5.1 allows a sharper concavity test.

5.1.1 Early and Simplified Algorithms

Early algorithms worked only with the list $\mathcal{L}$, without lists $\mathcal{U}$ and $\mathcal{C}$. Also, although the processes in steps 2 through 6 of Algorithm 14 make actual implementations practical and efficient, they are not an essential part of the branch and bound structure. In the early but well-known Moore–Skelboe algorithm, only the list $\mathcal{L}$ appears, and the boxes $\boldsymbol{X} \in \mathcal{L}$ are ordered in order of increasing $\underline{\phi}(\boldsymbol{X})$. In Step 7 of Algorithm 14, $\boldsymbol{X}$ is bisected along the largest coordinate direction, and both progeny are placed in order in $\mathcal{L}$. Steps 2 through 6 of Algorithm 14 are absent. When the algorithm is terminated, the first box in the list is taken to approximate the global minimizer.

An algorithm attributed to Ichida [91] improves upon the Moore–Skelboe algorithm by including the midpoint test (Step 2 of Algorithm 14) to avoid placing boxes generated during bisection onto $\mathcal{L}$ if they cannot contain optimizers.

Additionally, the algorithm described in [91] contains a method for grouping together clusters of boxes corresponding to particular minimizers.

Hansen's algorithms, described in [74], [75] and [77] generally use second-order information (Step 6 of Algorithm 14) and other sophisticated techniques. However an algorithm sometimes called "Hansen's algorithm" is a simplified version. In this simplified "Hansen's algorithm," $\mathcal{L}$ is ordered not in terms of the function, but in order of decreasing diameter (i.e. width of largest coordinate interval) of the $\boldsymbol{X}$. Furthermore, in the midpoint test, the entire list $\mathcal{L}$ is culled (and not just the boxes that have just been produced by bisection) whenever a new $\overline{\phi}$ is obtained. (In Hansen's actual codes of the experiments in [77, 235], the list is ordered such that the first box is the one with smallest lower bound on ϕ. Hansen and Walster, as well as others, claim this is much better than ordering in terms of decreasing diameter.) Various modifications of the list ordering, such as that in [194, §2.2.5.1], have appeared more recently.

None of these simplified methods employs interval-Newton acceleration.

Convergence properties of the Moore–Skelboe, Ichida and Hansen algorithms, as well as some numerical experiments with Hansen's algorithm, are analyzed in [168].

5.1.2 Recent Practical Algorithms

More recent algorithms and practical implementations usually involve interval Newton methods for computational existence and uniqueness of critical points and for quadratic convergence properties. However, some successful newer algorithms are derivative-free, and concentrate on use of approximate optimizers, order in which the list is searched, properties of the inclusion function, or parallelization.

A thorough exposition of background, starting with the elements of interval arithmetic, and of numerous techniques for interval unconstrained optimization, along with a substantial number of careful numerical experiments, appears in Ratz's dissertation [194]. Some of these ideas are implemented in the Pascal-XSC code described in [70].

Ratz, continuing development of his algorithms, has concentrated on better choice of coordinate in the bisection process of Step 7 of Algorithm 14, and on *splitting strategies*, cf. [196, 197]. Regarding bisection strategies, Ratz claims

better success when choosing the coordinate to bisect according to the scaling

$$|\nabla\phi(\boldsymbol{X})|\mathrm{w}(\boldsymbol{X} - X), \tag{5.1}$$

rather than merely bisecting along the longest coordinate direction of $\boldsymbol{X}$; cf. [194], pp. 41–42, [199], or and [46]. Convergence of generalized bisection based global optimization algorithms with this coordinate selection strategy is also proven in [46]. This scheme is essentially the maximum smear scheme of Definition 4.2 on page 157, applied to the objective function rather than to the gradient and Hessian. Box splitting is a process, first discussed by Hansen in relation to the interval Gauss–Seidel method, by which extended interval arithmetic in the sense of Kahan and Ratz is used to obtain two disjoint boxes. If applied wherever possible, too many boxes are produced, thus slowing the overall branch and bound algorithm. Coordinate choice in bisection, box-splitting strategies, and the ordering in the list $\mathcal{L}$ can be crucial in an overall global optimization algorithm.

Hansen's book [77] provides an informal description of many techniques and heuristics for use in global optimization algorithms. Many examples are given, although results of numerical experiments appear only for unconstrained problems.

In [113], experimental results are reported for a FORTRAN-77 code containing techniques for the monotonicity test and iteration/verification, as well as use of a local optimization process for computing $\overline{\phi}$ ("midpoint test "). The optimal LP-preconditioners of §3.2, as well as a technique for handling bound constraints through the tessellation, are studied there[4].

In [97], Jansson and Knüppel have presented a method without derivatives (no gradient test or interval Newton method), but with an interacting use of bisection and local optimization. In particular, a local optimization (to obtain an approximate optimizer) is performed at certain stages of the process, and the results are used to update $\overline{\phi}$. Though heuristic, the algorithm performs well on many reasonably complicated functions, including non-differentiable ones, such as maximizing the smallest singular value of a matrix. The report [99] contains a collection of test examples, along with numerical results and three-dimensional graphs of those (numerous) test problems that are two-variable functions. Jansson [98, §2.5] proposes a variant in which derivatives are used only in an interval Newton method to verify and sharpen bounds for approximate optima. This variant is carefully tested on forty test problems in [99].

[4]The latter two techniques are more fully explored in [194].

In [28] Caprani, Godthaab, and Madsen also propose[5] use of an approximate minimizer obtained through a local method with floating point arithmetic. In their algorithm, an approximate local minimizer is found, then a box $\boldsymbol{X}$ is constructed about this minimizer. An interval Newton method is then applied to $\boldsymbol{X}$ to determine existence or uniqueness of a critical point. If existence can be proven, $\boldsymbol{X}$ is expanded as much as possible, subject to success of the interval Newton method in verifying uniqueness, then $\boldsymbol{X}$ is removed from the region by cutting the complement of $\boldsymbol{X}$ into remaining boxes to be processed. The minimizer-inflation technique is related to ϵ-inflation of §4.2 (page 150). It is illustrated in [28] that the Caprani/Godthaab/Madsen method parallelizes well.

In [166], basic algorithms for non-differentiable and differentiable objective functions are reviewed, then a coarse-grained algorithm for optimization on a distributed-memory multicomputer, implemented on a distributed system of workstations, is explained. In this algorithm, each processor shares a portion of the list $\mathcal{L}$. The load is dynamically balanced as the computations proceed. The algorithm was programmed in C++, based on a system for interval arithmetic developed by Leclerc. An encapsulated explanation appears in [152]. The numerical experiments feature a very difficult parameter-fitting problem.

In [55] and [56], Eriksson *et al.* also study parallelization of an unconstrained global optimization algorithm, implemented on an Intel hypercube. Various load balancing strategies are compared on a set of six test problems, one of which was designed specifically to test different parallelization schemes.

Berner [20] improves on parallel load balancing schemes of Madsen, Eriksson, and Moore/Hansen/Leclerc. She also uses a variant of "bisection" in which the *two* largest coordinate directions according to the scaling (5.1) are bisected, thus producing *four* boxes rather than two. This technique (also mentioned in a more general form in [77]) may be especially advantageous in parallel algorithms, since the additional boxes can be more rapidly analyzed by otherwise idle processors. Along these lines, Csallner [44] claims, based on experimentation with a serial algorithm, that *trisection* in a single coordinate direction is better than *bisection,* since it produces two disconnected regions, where the objective function behaves differently, and which non-sharp interval extensions can distinguish.

[5]The idea of using a local minimizer appears to go back to [29], where Caprani and Madsen cite the formulation of Wilkinson from [241]: "In general it is the best in algebraic computations to leave the use of interval arithmetic as late as possible so that it effectively becomes an a posteriori weapon."

Method / Authors	Midpoint Test	Monotonicity Test	Concavity Test	Interval Newton	Parallelization	Use of Local Minimizer	Ref.
Moore / Skelboe							[165] and [224]
Ichida	yes						[91]
"Hansen's algorithm"	yes, and to cull list						[191]
Hansen's actual	yes	yes	yes	yes			[77]
Kearfott '92	yes	yes		yes		yes	[113]
Ratz	yes	yes	yes	yes		yes	[194] and [70]
Jansson / Knüppel	yes			yes	yes	yes	[99]
Caprani / Godthaab / Madsen	yes			yes	yes	yes	[28]
Hansen / Moore / Leclerc	yes	yes	yes	yes	subject of study		[166]
Eriksson	yes	yes		yes	subject of study		[56]
Wolfe	yes	yes		yes			[242]
Berner	yes	yes			subject of study	yes	[20]

Table 5.1 Summary attributes of various global optimization algorithms

Theoretical and empirical consequences of the order of the interval extension used to obtain $\underline{\phi}(\boldsymbol{X})$ are studied in [52] and [125]. However, exhaustive studies on a practical algorithm do not appear there.

Some (but not all) of the attributes of the algorithms in this section and in §5.1.1 are summarized in Table 5.1. Here, the label "Kearfott '92" refers to the code of [113]. "Hansen's actual" is used to denote the most recent algorithms of Hansen, described in [77] and forming the basis of the work in [166] and [152]. Blank spaces in the table indicate that the feature is not present.

5.2 HANDLING CONSTRAINTS

The book [191] contains an explanation of fundamental interval means of handling inequality constraints, while [77] discusses at length many interval techniques for both inequality and equality constraints. However, few numerical results using these techniques (excepting those in [115]) have been published. (See Table 5.2 on the next page. There, blank spaces mean the feature is absent.)

Method / Authors	Bound Constraints	Inequality Constraints	Equality Constraints	Second Order	Use of approx. minimizer	Numerical Experiments
"Hansen's"						unconstrained
Hansen's book	(as equality constraints)	yes	yes	yes		unconstrained only
Opt. '92	yes				yes	yes
Ratz	yes					yes
Jansson / Knüppel	yes			variant in [98, §2.5]	yes	unconstrained only
Caprani / Godthaab / Madsen					yes	unconstrained
Hansen / Moore / Leclerc						unconstrained only
Eriksson						unconstrained
Wolfe		yes	yes	yes		yes

Table 5.2 Summary of handling of constraints in various global optimization algorithms

Handling of simple bound constraints through the tessellation process has been explored in [113] and [194]. In general, the computational effort for such tessellation techniques increases exponentially with the number of bound constraints, but the computation times could remain reasonable for many specific problems. However, bound-constrained problems are intrinsically hard [183]. Although bound constraints can be handled as inequality constraints (as in [77]), it is unclear without published experimentation how such algorithms behave: it is possible that large numbers of small boxes, clustered on the boundaries of the constraints, are produced through the n-dimensional tessellation.

Alternately, inequality constraints can be handled as bound constraints by introduction of slack variables.

We elaborate on these concepts in the remainder of this section.

5.2.1 Checking Feasibility/Infeasibility

The following computations may be done with the constraints:

- using the constraints to delete portions of a region $\boldsymbol{X}$ that are infeasible;
- proving feasibility or infeasibility of inequality constraints;
- proving feasibility or infeasibility of equality constraints.

These possibilities are discussed in the remainder of this section.

In [77, §11.6], Hansen proposes heuristics for using inequality constraints to delete portions of a box $\boldsymbol{X}$ that cannot be feasible. Alternately, if the inequality constraints are converted into equality constraints first, the optimal preconditioner techniques of [111] and [126] in conjunction with the interval Gauss–Seidel method or interval Gaussian elimination may be used directly on the underdetermined m by n system of constraints. The latter provides a certain theoretical optimality not present in the heuristics of [77], with a smaller system than with the entire Fritz John system; this appears in [115]. However, experiments in [115] indicate that the scheme described in §5.2.4 is usually better.

An algorithm for constructing large boxes within which inequality constraints of the form $g(x) < 0$ are rigorously verified appears in [144], along with numerical experimentation. We have not included this algorithm in our tables, however, since it is not a general global search algorithm, but a method of dealing with constraints.

As indicated in Step 1a of Algorithm 14, an elementary check for infeasibility of an entire box $\boldsymbol{X}$ with respect to the equality constraints c_i is to verify that $0 \notin c_i(\boldsymbol{X})$ for some i. There is a corresponding check if an inequality constraint $d(x) \leq 0$ is used: the region is infeasible if simply $d(\boldsymbol{X}) > 0$. Similarly, if only inequality constraints of the form $d_i \leq 0$ are present, feasibility is proven if $d_i(\boldsymbol{X}) \leq 0$ for each i. In this case, Hansen calls the region *certainly feasible.* An alternate method of proving infeasibility for a system of equality constraints is discussed in §5.6.

On the other hand, proving feasibility for problems with equality constraints $c_i(x) = 0$ (like problem (1.3)), although necessary to get useful rigorous upper bounds $\overline{\phi}$ on the global minimum, is more difficult. In [242], Wolfe proposes an algorithm, based on a penalty function, for handling equality constraints. However, that algorithm considers feasibility to be rigorously proven only provided $c_i(x) \in [\epsilon, \epsilon]$ for each i and some fixed ϵ.

In contrast, in a method proposed in Hansen [77, §12.3 ff] and investigated in [115], existence of a point in a box $\boldsymbol{X}$ in which the constraints $C(X) = 0$ are simultaneously satisfied is rigorously proven. See [115, 116] and §5.2.4 below.

5.2.2 On Equality, Inequality and Bound Constraints

The book [191] poses the optimization problem analogous to problem (1.3) with both equality constraints $c_i(x) = 0$ and inequality constraints $d_i(x) \leq 0$, while [77] contains separate chapters on inequality constrained problems and equality constrained problems. At first glance, inequality constrained problems seem easier than equality-constrained ones. This is because feasibility can sometimes be proven without use of an interval Newton method: We merely bound the range of each d_j using interval arithmetic, then check that the upper bounds so obtained are all negative, i.e. we check $\mathbf{d}_j(\boldsymbol{X}) \leq 0$ for each j. Also, besides the more sophisticated technique in [77, §11.6], similar verification that $\mathbf{d}_j(\boldsymbol{X}) > 0$ for each j proves infeasibility over all of $\boldsymbol{X}$, allowing $\boldsymbol{X}$ to be eliminated from the global search region. Such techniques *should* probably be used in practical algorithms as additional tools to verify feasibility and to eliminate subregions. However, in such analyses, it is ignored that the inequality constraints can be *active*, that is, that they are in effect equality constraints.

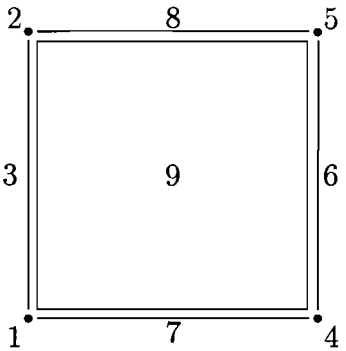
Various theoretical results have been published in recent years showing that global optimization problems containing inequality constraints are NP-complete in the number of constraints. For example, it is shown in [183] that quadratic programming problems with one negative eigenvalue are NP-complete.

The possibility that computational effort can increase exponentially in the number of constraints becomes apparent if we examine the algorithm for bound-constrained problems in [113], reviewed in §5.2.3 below. However, as explained in §5.2.3, it is possible for heuristics to reduce the running time for specific problems to less than that predicted by the exponential worst-case bounds, without compromising rigor.

5.2.3 Bound Constraints

In [113] and here, the bound constraints $\underline{x}_{i_j} \leq x_{i_j} \leq \overline{x}_{i_j}$ are handled by separating the region into all possible subregions of lower dimensions, as is illustrated in Figure 5.1 for $n = 2$. These subregions are placed on the list $\mathcal{L}$ and processed as usual, except that reduced gradients and reduced Hessian matrices[6] are used in the interval Newton method on lower-dimensional regions. If all boxes are stored in $\mathcal{L}$, it is not difficult to see that the total number of boxes of

[6] i.e. with rows and columns corresponding to variables held fixed deleted



There are 1 box in $\mathbb{R}^2$, 4 boxes in $\mathbb{R}^1$, and 4 boxes in $\mathbb{R}^0$.

Figure 5.1 "Peeling" a box to produce lower-dimensional boundary elements

all dimensions so obtained is 3^p, where p is the number of bound constraints. However, if a good upper bound $\overline{\phi}$ on the global minimum (as can sometimes be obtained with conventional algorithms such as that of [35]) is available, then many of the boxes can be rejected during the "peeling" process.

The structure of the algorithm for producing the list of lower-dimensional boxes can be described simply in recursive form, as follows.

Algorithm 15 ("Peeling" the Boundary)

INPUT: The box $\boldsymbol{X}$.

OUTPUT: A list $\mathcal{L}$ of boxes consisting of $\boldsymbol{X}$ and the lower-dimensional boundary elements of $\boldsymbol{X}$.

1. $i \leftarrow 1$, $\mathcal{L} \leftarrow \emptyset$.
2. IF $i \notin \mathcal{I}$ THEN
   (a) (Process the lower boundary box.)
      i. *Set all coordinates of* $\tilde{\boldsymbol{X}}$ *but the* i*-th to corresponding coordinates of* $\boldsymbol{X}$. *Set the* i*-th coordinate of* $\tilde{\boldsymbol{X}}$ *to* $\underline{x}_i$.
      ii. *Set the index list* $\mathcal{I}_{\text{new}}$ *to* $\mathcal{I}$ *with* i *appended.*
      iii. IF $i = n$
         THEN *store* $\tilde{\boldsymbol{X}}$ *in* $\mathcal{L}$.
         ELSE *execute Step 2 with* $i + 1$, $\tilde{\boldsymbol{X}}$, *and* $\mathcal{I}_{\text{new}}$ *replacing* i, $\boldsymbol{X}$, *and* $\mathcal{I}$, *respectively.*

END IF

(b) (Process the upper boundary box.)

i. Set all coordinates of $\tilde{\boldsymbol{X}}$ *but the* i*-th to corresponding coordinates of* $\boldsymbol{X}$*. Set the* i*-th coordinate of* $\tilde{\boldsymbol{X}}$ *to* $\overline{x}_i$*.*

ii. Set the index list $\mathcal{I}_{\text{new}}$ *to* $\mathcal{I}$ *with* i *appended.*

iii. IF $i = n$

THEN *store* $\tilde{\boldsymbol{X}}$ *in* $\mathcal{L}$*.*

ELSE *execute Step 2 with* $i + 1$*,* $\tilde{\boldsymbol{X}}$*, and* $\mathcal{I}_{\text{new}}$ *replacing* i*,* $\boldsymbol{X}$*, and* $\mathcal{I}$*, respectively.*

END IF

(c) (Process the interior box.)

i. Set $\tilde{\boldsymbol{X}}$ *to* $\boldsymbol{X}$*.*

ii. Set the index list $\mathcal{I}_{\text{new}}$ *to* $\mathcal{I}$ *with* i *appended.*

iii. IF $i = n$

THEN *store* $\tilde{\boldsymbol{X}}$ *in* $\mathcal{L}$*.*

ELSE *execute Step 2 with* $i + 1$*,* $\tilde{\boldsymbol{X}}$*, and* $\mathcal{I}_{\text{new}}$ *replacing* i*,* $\boldsymbol{X}$*, and* $\mathcal{I}$*, respectively.*

END IF

END IF

End Algorithm 15

The numbering of the nine boxes of dimensions 2, 1, and 0 in Figure 5.1 represents the order they would appear in $\mathcal{L}$ if each box generated with $i = n$ in steps 2(a)iii, 2(b)iii, and 2(c)iii of Algorithm 15 were stored. The processing order in Algorithm 15 can be viewed as traversing a ternary tree, as in Figure 5.2. The levels of this tree correspond to the coordinates i, with the root at $i = 1$ at the top and the leaves at $i = n$ at the bottom. As drawn in Figure 5.2, the order the leaves eventually appear in $\mathcal{L}$ is from left to right.

Of course, actual implementations of Algorithm 15 have additional steps to

- eliminate the boxes $\boldsymbol{X}$ and $\tilde{\boldsymbol{X}}$ before further processing or storage in $\mathcal{L}$ by checking $\phi(\boldsymbol{X})$ or $\phi(\tilde{\boldsymbol{X}})$ and the reduced gradient of ϕ on $\boldsymbol{X}$ or $\tilde{\boldsymbol{X}}$;
- skip coordinates i for which the bounds $a_i \leq x_i \leq b_i$ represent the extent of the search region, and not actual bound constraints for the problem.

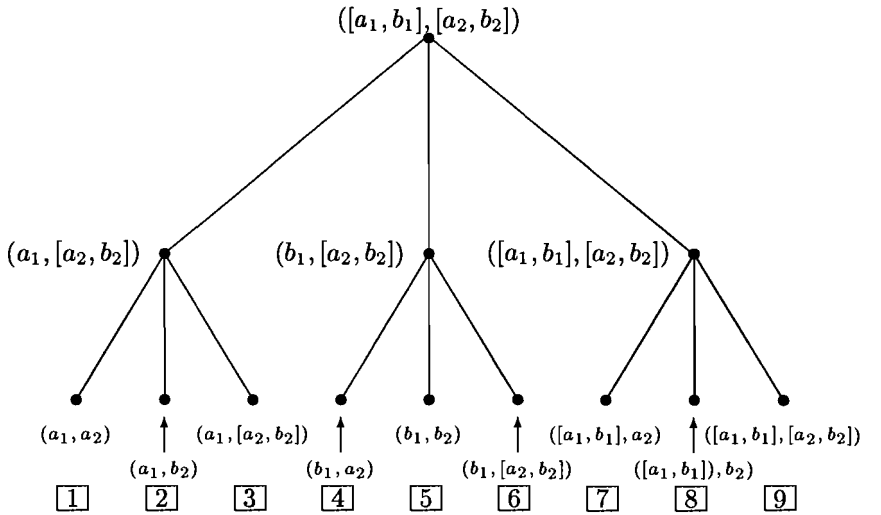


Figure 5.2 "Peeling" the box into lower-dimensional boundary elements

These steps have been left out of the presentation of Algorithm 15 for clarity in explaining the geometric process. However, they could be indispensable in reducing the number of boxes in $\mathcal{L}$ to a practical number. Observe that such steps can prune the tree in Figure 5.2 at a high level.

In [194, §2.6.3], Ratz proposes an alternative technique to avoid the possible exponential expansion in the number of boxes in $\mathcal{L}$ when the boundary is "peeled." In Ratz' scheme, portions of the boundary are put onto $\mathcal{L}$ only when a reduced interval Newton method rejects a boundary segment as a possibility. Ratz' algorithm is similar to the following algorithm.

Algorithm 16 (Handling the Boundary with an Interval Newton method)

INPUT:

1. The box $\boldsymbol{X}$, p of whose coordinates $x_{i_1}, \ldots, x_{i_p}$ correspond to intervals, and whose other coordinates correspond to active bound constraints;
2. The present list $\mathcal{L}$ of boxes to be processed.

OUTPUT:

1. An altered box $\tilde{\boldsymbol{X}}$
2. An altered list $\mathcal{L}$.

DO *for* $j = 1$ *to* p.

1. *Perform an interval Gauss–Seidel step*[7] *for the* i_j *coordinate* $\boldsymbol{x}_{i_j}$ *of* $\boldsymbol{X}$, *obtaining* $\tilde{\boldsymbol{x}}_{i_j}$.
2. IF $\underline{x}_{i_j}$ *corresponds to an active bound constraint and* $\tilde{x}_{i_j} > \underline{x}_{i_j}$
   THEN
   (a) *Form a box* $\boldsymbol{X}_{\text{bdy}}$ *whose* i*-th coordinate is* $\tilde{\boldsymbol{x}}_i$ *for* $i \neq i_j$ *and whose* i_j*-th coordinate is* $\underline{x}_{i_j}$.
   (b) *Place* $\boldsymbol{X}_{\text{bdy}}$ *onto* $\mathcal{L}$.
   END IF
3. IF $\overline{x}_{i_j}$ *corresponds to an active bound constraint and* $\tilde{x}_{i_j} > \overline{x}_{i_j}$
   THEN
   (a) *Form a box* $\boldsymbol{X}_{\text{bdy}}$ *whose* i*-th coordinate is* $\tilde{\boldsymbol{x}}_i$ *for* $i \neq i_j$ *and whose* i_j*-th coordinate is* $\overline{x}_{i_j}$.
   (b) *Place* $\boldsymbol{X}_{\text{bdy}}$ *onto* $\mathcal{L}$.
   END IF

END DO

End Algorithm 16

The structure of a bound-constrained optimization code that includes Algorithm 16 would be identical to that of an optimization code without bound constraints. The only difference would be that the interval Newton step would be replaced by Algorithm 16. Algorithm 16 holds promise of producing less boxes in $\mathcal{L}$ than the peeling process of Algorithm 15, but its practicality needs to be verified empirically. In particular, Algorithm 15 evaluates the objective function on many lower-dimensional parts of the boundary early in the branch and bound process; this may result in a better estimate for the global optimum, and hence in faster elimination of subsequent boxes that do not contain optima.

[7] If there are no equality constraints, the underlying system will correspond to the reduced gradient system $\nabla \phi_{\text{r}} = 0$, that is, the gradient of the system with respect to the p coordinates that are held fixed. If equality constraints are included, the system will be a corresponding reduced Fritz–John system, as in Equation (5.3) below.

5.2.4 Feasibility of Equality Constraints in Bound-Constrained Problems

In unconstrained and bound constrained problems, an upper bound $\overline{\phi}$ of the objective function over a box $\boldsymbol{X}$ can be obtained with an interval evaluation $\boldsymbol{\phi}(\boldsymbol{X})$. However, $\boldsymbol{\phi}(\boldsymbol{X})$ provides an upper bound in equality constrained problems, that is, in Problem (1.3) with $m > 0$, *only* if the points in $\boldsymbol{X}$ satisfy the bound constraints and $\boldsymbol{X}$ contains a point X with $C(X) = 0$. This section is devoted to techniques for proving this, i.e. to solve the following problem:

> Given an approximate feasible point $\check{X} \in \mathbb{R}^n$ for Problem (1.3), construct bounds
>
> $$\boldsymbol{X} = \{(x_1, \ldots, x_n)^T \in \mathbb{R}^n \mid \check{x}_i - \epsilon_i \leq x_i \leq \check{x}_i + \epsilon_i\}$$
>
> such that there exists at least one feasible point of Problem (1.3) in $\boldsymbol{X}$, i.e. a point $X \in \boldsymbol{X}$ with $C(X) = 0$, such that X also satisfies the bound constraints of Problem (1.3). (5.2)

Systems of equality constraints $C(X) = 0$, $C : \mathbb{R}^n \to \mathbb{R}^m$ are typically underdetermined, with $m < n$. In [77, §12.3 ff.], Hansen suggests holding $(n-m)$ of the coordinates fixed. The basic idea is to choose to be held fixed those coordinates to which the system is least sensitive. For example, if $n = 2$ and there is just one constraint (i.e. $m = 1$), then $C(X) = 0$ is a curve in $\mathbb{R}^2$, and an interval Newton method can prove the existence of a feasible point along a line parallel to one of the coordinate directions; see Figure 5.3 (next page). Heuristics for choosing which coordinates to hold fixed can depend on the constraint matrix $\nabla C(X)$ (and corresponding interval extensions or slope matrices). See page 187 below.

It is not unusual for the opposite problem to occur when there are many active bound constraints. In particular, it often happens that many bound constraints are active at the optimizers. This is also true of inequality-constrained problems, which we convert to equality- and bound-constrained problems (§5.2.2). An extreme case of many active bound constraints is linear programming, in which, barring degeneracy, a maximal number of constraints must be active at optimizers.

Thus, to get a good upper bound $\overline{\phi}$, a local optimizer is used to obtain an approximate feasible point $\check{X}$ that lies near or on many bound constraints.

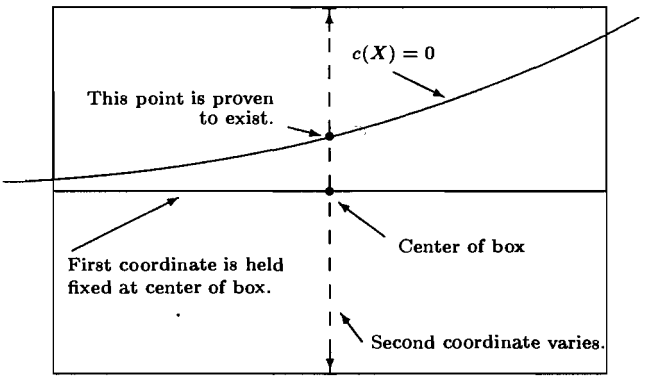


Figure 5.3 Proving that there exists a feasible point of an underdetermined constraint system

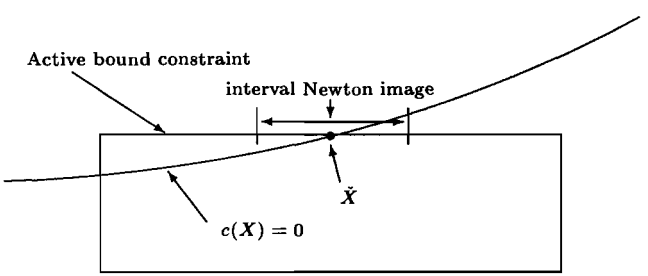


Figure 5.4 Proving existence in a reduced space when the approximate feasible point satisfies bound constraints

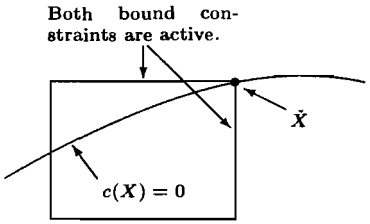


Figure 5.5 A common degenerate case, when $\check{X}$ must be perturbed

If a box $\boldsymbol{X}$ (in which a feasible point X will be proven to exist) is to then be constructed about the approximate feasible point $\check{X}$ and lying within the region defined by the bound constraints, then $\check{X}$ must be near the boundary of $\boldsymbol{X}$. If there are sufficient degrees of freedom left when the variables corresponding to bound constraints are held fixed, then the interval Newton method can be applied in a subspace. See Figure 5.4, as well as Example 5.2 on page 191 below.

Surprisingly, the degenerate case, illustrated for $n = 2$ and $m = 1$ in Figure 5.5, is common[8]. That is, the manifold $C(X) = 0$ can lie near a corner of $\boldsymbol{X}$, and the techniques embodied in Algorithm 17 below may fail to prove that a feasible point exists in $\boldsymbol{X}$. This is because the image of the interval Newton method tends to be centered at a solution of $F(X) = 0$, and if the solution is near the boundary of $\boldsymbol{X}$, the image $\boldsymbol{N}(F; \boldsymbol{X}, \check{X})$ may overlap with points outside $\boldsymbol{X}$. If $\check{X}$ lies on a corner of $\boldsymbol{X}$ (i.e. if too many bound constraints are active), then $\check{X}$ can be perturbed into the interior of the bounds, and the resulting point can somehow be projected back onto the manifold $C(X) = 0$, for example by using local optimization software in a hyperplane parallel to the bound constraints. Some details of these techniques are explained on page 191 below.

Underdetermined systems

To prove existence of a feasible point of $C(X) = 0$, $C : \mathbb{R}^n \to \mathbb{R}^m$, $m < n$, within a box $\boldsymbol{X} \in \mathbb{IR}^n$, Hansen proposed a technique [77, §12.3–§12.6] of holding $n - m$ of the variables held fixed. This general technique was investigated numerically in [115], with additional details in [116]. The following general algorithm encompasses Hansen's proposed feasibility verification methods and the variants in [115, 116]. The variants differ in how the variables to be held fixed are chosen, i.e. in the details of Step 2.

[8] in the experiments reported in [115], with problems from [58]

Algorithm 17 (Prove that a point within a small box satisfies equality constraints.)

INPUT: an approximation $\check{X}$ to a feasible point, obtained through a conventional local constrained optimization algorithm such as that of [35], and a tolerance ϵ.

OUTPUT: Either:

1. "failure", or
2. "success", and a box $\tilde{\boldsymbol{X}}$ of distance at most ϵ from $\check{X}$ (i.e. $\|X - \check{X}\|_\infty \leq \epsilon$ for every $X \in \tilde{\boldsymbol{X}}$), such that there exists a point $X \in \tilde{\boldsymbol{X}}$ such that $C(X) = 0$.

1. *Let $C(X) = (c_1(X), \ldots, c_n(X))^T$ and $\nabla C(X)$ represent the equality constraints and Jacobi matrix of the equality constraints, respectively ($C : \mathbb{R}^n \to \mathbb{R}^m$).*
2. *Choose coordinates $\{p_k\}_{k=1}^m$ to be varied in the interval Newton method in such a way that the resulting system is likely to be nonsingular.*
3. *Evaluate $C(\check{X})$ and $\boldsymbol{A}$, where $\boldsymbol{A}$ is either a Lipschitz matrix or a slope matrix for C over $\widehat{\boldsymbol{X}}$, where $\widehat{\boldsymbol{X}}$ has coordinates $\widehat{\boldsymbol{x}}_i = \check{x}_i$ if $i \neq p_k$ for any k and $\widehat{\boldsymbol{x}}_i = [\check{x}_i - \epsilon, \check{x}_i + \epsilon]$ for $i = p_k$ for some ϵ. That is, construct a box in an m-dimensional subspace that is, in a sense, most nearly perpendicular to the null space of $\nabla C(\check{X})$.*
4. *Apply an interval Newton method to the square m-dimensional system of equations $C(X) = 0$ obtained by identifying the variables in the expressions for C with indices $\{p_k\}_{k=1}^m$ as variables, and viewing the other variables as constant. Call the image $\tilde{\boldsymbol{X}}$.*
5. IF $\tilde{\boldsymbol{X}} \subseteq \widehat{\boldsymbol{X}}$,

   THEN *return "success" and $\tilde{\boldsymbol{X}}$.*

   ELSE *return "failure".*

End Algorithm 17

Experiments on this algorithm are reported in [115]. The general conclusions are that Algorithm 17 is effective, and that Step 2 is effective, *provided a good approximation to a feasible point is centered in a box whose center is at $\check{X}$*. For a detailed explanation and additional illustrations, see [116].

The experiments reported in [115] also indicate that Hansen's way of choosing the coordinates to be held fixed leads to more reliable results than several alternatives. Hansen's technique is as follows:

Algorithm 18 (Hansen's technique for choosing the fixed coordinates in Algorithm 17

INPUT: The matrix $\boldsymbol{A}$ from Step 3 of Algorithm 17.

OUTPUT: $n - m$ coordinate indices to be held fixed.

1. *Compute the midpoint matrix* $A \in \mathbb{R}^{m \times n}$ *of* $\boldsymbol{A}$.
2. *Perform Gaussian elimination with complete pivoting on the rectangular matrix* A.
3. *Choose the original indices of the columns of* A *that have been permuted into the last* $n - m$ *columns during the elimination process to be the indices of those variables to be held fixed (i.e. to be replaced by points) in the interval Newton method.*

End Algorithm 18

Alternate choices of coordinates to that of Algorithm 18 are described in [115, 116].

A more general alternative is also described in [115, 116]. In this alternative, an n-dimensional box $\boldsymbol{X}$ is first constructed about an approximate feasible point $\check{X}$. A C^{W}-preconditioner or C^{M}-preconditioner (as in §3.2) is then computed for *each* row of the m by n interval linear system

$$\boldsymbol{A}(X - \check{X}) = -C(\check{X})$$

while applying steps of the interval Gauss–Seidel method successively to each of the n coordinates of the box $\boldsymbol{X}$, thus obtaining an image $\tilde{\boldsymbol{X}}$. If $\tilde{\boldsymbol{x}}_i \subseteq \boldsymbol{x}_i$ for m coordinates $\{i_1, \ldots, i_m\}$, then, for each choice of the $x_j \in \boldsymbol{x}_j$, $j \notin \{i_1, \ldots, i_m\}$, there is a unique solution to $C(X) = 0$ within $\boldsymbol{X}$. Figure 5.6 illustrates this process for $n = 2$ and $m = 1$. In Figure 5.6, we imagine that a C^{W}-preconditioner or C^{M}-preconditioner has been applied to obtain first $\tilde{\boldsymbol{x}}_1 \subset \boldsymbol{x}_1$ and then $\tilde{\boldsymbol{x}}_2 \supseteq \boldsymbol{x}_2$. The smallest possible $\tilde{\boldsymbol{x}}_1$, based on how the curve $c(X) = 0$ intersects $\boldsymbol{X}$, is illustrated in Figure 5.6 (next page) with the broken vertical lines. The computation represented in Figure 5.6 shows that, for *every* $x_2 \in \boldsymbol{x}_2$, there is an $x_1 \in \tilde{\boldsymbol{x}}_1$ such that $c(X) = c(x_1, x_2) = 0$. This is a stronger statement than that provided by Algorithm 17. However, *both* coordinates are intervals in evaluating the matrix $\boldsymbol{A}$ for the interval Newton iteration. In contrast, only one coordinate in the expressions in the matrix for the interval Newton iteration in Algorithm 17 would be an interval. Thus, interval overestimation is less of a problem in Algorithm 17, and Algorithm 17 is

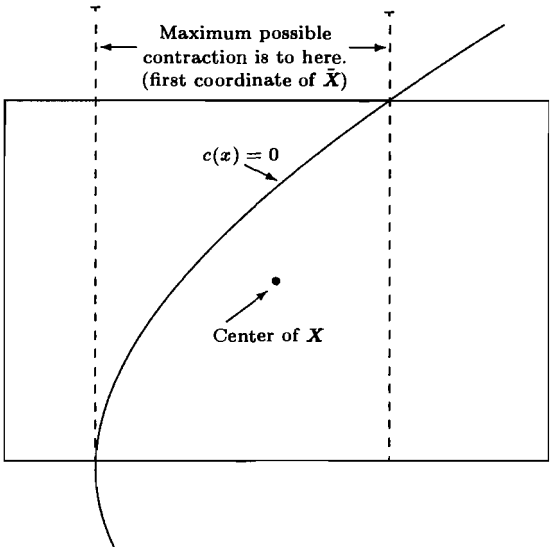


Figure 5.6 Geometry of a feasibility proof with LP preconditioners

more likely to give a positive result than preconditioning the entire rectangular system. For details, see [115, 116].

Example 5.1 *Consider*

$$\begin{aligned} \textit{minimize} \quad & \phi(X) = -(x_1 + x_2)^2 \\ \textit{subject to} \quad & c(X) = x_2 + 2x_1 = 0, \end{aligned}$$

and no bound constraints, and with approximate feasible point $\check{X} = \left(-\frac{1}{4}, \frac{1}{2}\right)^T$.

To use Algorithm 17 and Algorithm 18 on Example 5.1, first observe that $\nabla C \equiv (+2, 1)^T$. Therefore, Algorithm 18 states that x_2 should be held fixed at $x_2 = \frac{1}{2}$. Thus, to prove existence of a feasible point in a neighborhood of $\check{X}$, an interval Newton method can be applied to $f(x_1) = c(x_1, 0.5) = 0.5 - 2x_1$. We may choose initial interval $\boldsymbol{x}_1 = [-0.25 - \epsilon, -0.25 + \epsilon]$ with $\epsilon = 0.1$, to obtain

$$\begin{aligned} \boldsymbol{x}_1 &= [-.35, -.15], \\ \tilde{\boldsymbol{x}}_1 &= -0.25 - \frac{0}{-2} \\ &= [-0.25, -0.25] \subset \boldsymbol{x}_1, \end{aligned}$$

This computation proves that, for $x_2 = 0.5$, there is a feasible point of Example 5.1 for $x_1 \in [-0.25, -0.25]$.

To use the interval Gauss–Seidel method in the entire space to prove existence of a feasible point in Example 5.1, first observe that, since $m = 1$, all preconditioners are equivalent. Let's start with initial box $\boldsymbol{X} = \check{X} + ([-\epsilon, \epsilon], [-\epsilon, \epsilon])^T$, $\epsilon = 0.1$, i.e. $\boldsymbol{X} = ([-0.35, -0.15], [0.4, 0.6])^T$. We then compute $\tilde{\boldsymbol{x}}_1$ according to Equation 1.44, (with $Y = 1$):

$$\begin{aligned} \tilde{\boldsymbol{x}}_1 &= -0.25 - \{0 + 1[-.1, .1]\}/(-2) \\ &= -0.25 + [-0.05, 0.05] \\ &= [-0.3, -0.2] \subset [-0.35, -0.25] = \boldsymbol{x}_1 \end{aligned}$$

This computation proves that, for *every* choice of $x_2 \in [0.4, 0.6]$, there is a feasible point of Example 5.1 in the interval $\tilde{\boldsymbol{x}}_1 = [-0.3, -0.2]$.

Working directly with the rectangular system is applicable, with optimal preconditioning, for general n and $m < n$. It is related to the continuation method process developed in [130]. However, holding some coordinates fixed and working in a reduced space appears to succeed more often [115].

Overdetermined Systems – Many Active Bound Constraints

As mentioned on page 187 and in Figure 5.4, if some bound constraints are active at the approximate feasible point $\check{X}$, it may be possible to work in a reduced space.

Example 5.2 *Consider*

$$\begin{aligned} \textit{minimize} \quad \phi(X) &= -\left(x_1^2 + x_2^2\right) \\ \textit{subject to} \quad c(X) &= x_2 - 4x_1^2 = 0, \\ -1 &\leq x_1 \leq 1, \\ -1 &\leq x_2 \leq 1, \end{aligned}$$

and with approximate feasible point $\check{X} = (0.499, 1)^T$.

In Example 5.2, the bound constraint $x_2 \leq 1$ is active, so an interval Newton method can be applied in a one-dimensional subspace. In particular, to prove

the existence of a feasible point in a neighborhood of $\check{X}$, an interval Newton method may be applied to $f(x_1) = c(x_1, 1) = 1 - 4x_1^2$. Using initial box $\left[\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon\right]^T$ with $\epsilon = 0.1$, we obtain

$$\begin{aligned} \boldsymbol{x}_1 &= [.399, .599], \\ \tilde{\boldsymbol{x}}_1 &= .499 - \frac{3.996 \times 10^{-3}}{-8[.399, .599]} \\ &\subseteq .499 + [8.338 \times 10^{-4}, 1.252 \times 10^{-3}] \\ &\subseteq [.4998, .5003] \subset \boldsymbol{x}_1. \end{aligned}$$

This computation proves that there exists a feasible point of the problem in Example 5.2 for $x_2 = 1$ and within the box $\tilde{\boldsymbol{x}} = [.4998, .5003]$.

However, degenerate cases as illustrated in Figure 5.5 do occur [115]. The following simple example illustrates the phenomenon.

Example 5.3 *Consider*

$$\begin{aligned} \textit{minimize} \quad & \phi(X) = -\left(4x_1^2 + x_2^2 + 4x_3^2\right) \\ \textit{subject to:} \quad & c_1(X) = x_1^2 + x_2^2 + x_3 - 3 = 0, \\ & c_2(x) = x_1^2 + x_2 - x_3^2 - 1 = 0, \\ & -1 \leq x_1 \leq 1, \\ & 0 \leq x_2 \leq 2, \\ & -1 \leq x_3 \leq 1, \end{aligned}$$

with $\check{X} = (1, 1, 1)^T$.

The point $\check{X}$ corresponds to one of two global optimizers within the box defined by the bound constraints. However, two bound constraints (corresponding to x_1 and x_3) are active at $\check{X}$, and the image of a two-dimensional interval Newton method obtained by holding either x_1 or x_3 fixed will contain points that violate the bound constraints. (See Exercise 2.) However, if we perturb x_1, by $x_1 = 1 - 0.1 = 0.9$, then solve the constraint system

$$\begin{aligned} 0.9^2 + x_2^2 + x_3^2 - 3 &= 0, \\ 0.9^2 + x_2 - x_3^2 - 1 &= 0, \end{aligned}$$

we obtain the approximate feasible point $(.9, 1.1095, .9589)^T$. We may then hold x_1 fixed, and apply an interval Newton method to the above system, with

$$\boldsymbol{X} = (x_2, x_3)^T = ([1.1095, 1.1195], [0.9489, 0.9689])^T,$$

$$F(X) = (f_1(x_2, x_3), f_2(x_2, x_3))^T = (0.9^2 + x_2^2 + x_3^2 - 3, 0.9^2 + x_2 - x_3^2 - 1)^T,$$

to verify that a feasible point exists for $x_1 = 0.9$, $x_2 \in [1.1095, 11096]$, $x_3 \in [0.9589, 0.95893]$ (Exercise 2).

The following procedure was found to be reliable [116] at perturbing points away from bound constraints, in cases such as Example 5.3.

Algorithm 19 (Move coordinates off their bound constraints one at a time.)
INPUT: A domain tolerance ϵ_d, the approximate feasible point $\check{X}$, and the bound constraints.
OUTPUT: Either:

1. "success" and a new $\check{X}$ enough of whose coordinates do not lie on bound constraints, so existence of a feasible point can be proven, or
2. "failure."

1. *Let n_r be the dimension of the reduced space, i.e. let n_r be the number of coordinates of $\check{X}$ that do not lie on their bound constraints. Also let $\{i_j\}_{j=1}^{n-n_r}$ be the coordinate indices of $\check{X}$ corresponding to active bound constraints.*
2. DO $j = 1$ *to* $n - n_r$.

   IF $\check{x}_{i_j}$ *is on its bound coordinate,* THEN

   (a) *Replace $\check{x}_{i_j}$ by $\check{x}_{i_j} + \delta$, where $\delta = \max\{|\check{x}_{i_j}|, 1\}\sqrt{\epsilon_d}$ if $\check{x}_{i_j}$ was on its lower bound and $\delta = -\max\{|\check{x}_{i_j}|, 1\}\sqrt{\epsilon_d}$ if $\check{x}_{i_j}$ was on its upper bound.*

   (b) *If $\check{x}_{i_j}$ was originally on its lower bound, redefine the lower bound for the correction step to be the perturbed value of $\check{x}_{i_j}$. Similarly redefine the upper bound if $\check{x}_{i_j}$ was originally on its upper bound.*

   (c) (Correction Step) *Run the local bound-constrained optimizer, using the temporary bound constraints defined in step 2b to obtain a new $\check{X}$.*

   (d) *Recompute which bound constraints are active, possibly projecting onto bound constraints if coordinates of the new point are within a tolerance of a boundary; obtain a new reduced dimension n_r.*

   (e) IF $n_r \geq m$ THEN EXIT*with success.*

   END IF

   END DO

3. IF $n_r < m$ THEN EXIT *with failure.*

End Algorithm 19

After successful completion of Algorithm 19, a box whose coordinates are centered on coordinates of $\check{X}$ and whose semi-widths are on the order of ϵ_d can be constructed in the reduced space of n_r coordinates not on the bound constraints. An interval Newton method applied over this box should then lead to verification of existence of a feasible point.

One may think that the system of Fritz John equations described in §5.2.5 could handle bound constraints directly and easily. However, interval Jacobi matrices and slope extensions over regions that contain limits on many bound constraints often contain singular matrices.

Infeasibility of Systems of Equality Constraints

An elementary way of showing $C(X) = 0$ is infeasible over $\boldsymbol{X}$ is to compute an interval evaluation and verify $0 \notin \boldsymbol{C}(\boldsymbol{X})$. However, the same preconditioning technique described on page 189 to verify existence of feasible points could also be used to verify non-existence. The process is even valid when there are more constraints than variables.

Example 5.4 *Consider the system of constraints*

$$\begin{aligned} c_1(X) &= x_1 - x_2 = 0 \\ c_2(X) &= x_1 + x_2 = 0 \\ c_3(X) &= (x_1 + 0.1)^2 + x_2 - 1 = 0 \end{aligned}$$

over the box $\boldsymbol{X} = ([0,1],[-1,1])^T$.

The exact range is $\boldsymbol{C}^u(\boldsymbol{X}) = ([-1,1],[0,2],[-0.99,1.21])^T$, so interval evaluations cannot prove infeasibility. However, an E^{M}-preconditioner or other preconditioner can verify infeasibility here. In fact, if $\check{X} = (0,0)^T$, an slope matrix for C is

$$\mathbf{S}(C, \boldsymbol{X}, \check{X}) = \begin{pmatrix} 1 & -1 \\ 1 & 1 \\ [0.2, 1.2] & 1 \end{pmatrix},$$

and the interval Newton system is

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \\ [0.2, 1.2] & 1 \end{pmatrix} \left(X - \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}.$$

If the preconditioner row $Y_1 = [0 - 1, 1]$ is used, the resulting equation is

$$[-0.8, -0.2]x_1 = -1,$$

whence $x_1 \in \tilde{x}_1 = [1.25, 5]$. Since $\tilde{x}_1 \cap x_1 = \emptyset$, this computation proves that there are no feasible points X, $C(X) = 0$ in $\boldsymbol{X}$.

5.2.5 The Fritz John Conditions

In certain circumstances, we wish to simultaneously verify feasibility and local optimality. For instance, a conventional floating point constrained optimizer may have been used to obtain an approximate global optimum $\check{X}$. We may then wish to verify uniqueness of a critical point in as large a box $\boldsymbol{X}$ as possible about $\check{X}$, so that $\boldsymbol{X}$ may be excluded from further consideration in the exhaustive search[9]. In other cases, we wish to use interval Newton iteration to eliminate boxes $\boldsymbol{X}$ with $\boldsymbol{X} \cap \boldsymbol{N}(F; \boldsymbol{X}, \check{X}) = \emptyset$. In these cases, the system of equations to be used in the interval Newton method must therefore embody the necessary conditions for constrained optima. Such general conditions, not requiring "constraint qualification" assumptions, are the Fritz John conditions.

The Fritz John conditions have been advocated by Hansen *et al.* in [77] and [79]. Their use is thoroughly explained in those works, although empirical results are lacking. Here, we present and discuss first a variant we have found good, then a full variant that includes the bound constraints. That is, we give the function F and derivative matrix ∇F corresponding to the Fritz John conditions for our formulation in Equation (1.3).

The variables in our system are $X = (x_1, \ldots, x_n)$, $V = (v_1, \ldots, v_m)$, and u_0, for a total of $n + m + 1$ variables. We will write $W = (X, u_0, V)$. The function corresponding to the gradient of the Lagrange function is:

$$F(W) = \begin{pmatrix} u_0 \nabla \phi(X) + \sum_{i=1}^{m} v_i \nabla c_i(X) \\ c_1(X) \\ \vdots \\ c_m(X) \\ \left(u_0 + \sum_{i=1}^{m} v_i^2\right) - 1 \end{pmatrix} = 0, \tag{5.3}$$

The v_i are unconstrained and represent Lagrange multipliers for the equality constraints $c_i = 0$. The last equation is a normalization condition.

By not including the bound constraints $\underline{x}_{i_j} \leq x_{i_j}$ and $x_{i_j} \leq \overline{x}_{i_j}$ in this function, we reduce the size of the system by q. Furthermore, it is more flexible to

[9]This is done in the unconstrained case in [28] and for general nonlinear systems in [114].

include the bound constraints through the process reviewed in §5.2.3. Thus, Equation (5.3) is applicable for points X when none of the bound constraints are active. When one or more bound constraints are active, an analogue of Equation (5.3) in the appropriate lower-dimensional subspace is used. The technique also avoids singularities in the Jacobi matrix of the Fritz–John function that often occur when bound constraints are included.

The Jacobi matrix of F, used to form the matrix $\boldsymbol{A}$ for interval Newton methods, is:

$$H(W) = \left(\begin{array}{c|c|ccc} u_0\nabla^2\phi(X) + \sum_{i=1}^{m} v_i\nabla^2 c_i(X) & \nabla\phi(X) & \nabla c_1(X) & \cdots & \nabla c_m(X) \\ \hline (\nabla c_1(X))^T & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\nabla c_m(X))^T & 0 & 0 & \cdots & 0 \\ \hline 0 & 1 & 2v_1 & \cdots & 2v_m \end{array}\right). \tag{5.4}$$

The last equation in (5.3) is a normalization condition that can be chosen in various ways. In particular, Hansen and Walster [79] recommend that it be chosen as

$$\left(u_0 + \sum_{i=1}^{m} v_i[1, 1+\epsilon]\right) - 1, \tag{5.5}$$

where ϵ is on the order of the computational precision. If this is done, the last row of the matrix H in (5.4) becomes

$$\left(\begin{array}{c|c|ccc} 0 & 1 & [1,1+\epsilon], & \cdots & [1,1+\epsilon] \end{array}\right). \tag{5.6}$$

If Hansen slopes (of §1.3.2 on page 30) are then used to compute an interval analogue of the matrix (5.4), the "v" variables do not appear as intervals. Thus, if Gaussian elimination is used to bound the solution set to the corresponding preconditioned interval linear system, initial bounds on the Lagrange multipliers u_0 and v_i are not required. However, some of our experiments [121] indicate that the search algorithm is more efficient with the quadratic normalization in (5.4).

In theory, the bound constraints can also be included in the Lagrange function. If this is done, (5.3) becomes

$$F(W) = \begin{pmatrix} u_0\nabla\phi(X) + \sum_{j=1}^{q} u_j\nabla p_j + \sum_{i=1}^{m} v_i\nabla c_i(X) \\ u_1 p_1 \\ \vdots \\ u_q p_q \\ c_1(X) \\ \vdots \\ c_m(X) \\ \left(u_0 + \sum_{j=1}^{q} u_j + \sum_{i=1}^{m} v_i^2\right) - 1 \end{pmatrix} = 0, \tag{5.7}$$

where $u_i \geq 0$, $0 \leq i \leq q$, where q is the total number of bound constraints, and $p_j(X) = \underline{x}_{i_j} - x_{i_j}$ for $1 \leq j \leq \mu$, $p_j(X) = x_{i_j} - \overline{x}_{i_j}$ for $\mu + 1 \leq j \leq q$; see, for example, [60, §3.4.2]. Thus, $\nabla p_j = E_{i_j}$ for $1 \leq j \leq \mu$ and $\nabla p_j = -E_{i_j}$ for $\mu + 1 \leq j \leq q$, where E_i is the i-th coordinate vector.

The analogue to the matrix (5.4) is thus

$$\left(\begin{array}{c|c|ccccc|ccc} u_0\nabla^2\phi(X) + \sum_{j=1}^{q} u_j\nabla^2 p_j + \sum_{i=1}^{m} v_i\nabla^2 c_i(X) & \nabla\phi(X) & \nabla p_1 & \dots & \nabla p_q & \nabla c_1(X) & \dots & \nabla c_m(X) \\ \hline u_1(\nabla p_1)^T & 0 & p_1 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \dots & \vdots \\ u_q(\nabla p_q)^T & 0 & 0 & 0 & p_q & 0 & \dots & 0 \\ \hline (\nabla c_1(X))^T & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (\nabla c_m(X))^T & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \hline 0 & 1 & 1 & \dots & 1 & 2v_1 & \dots & 2v_m \end{array}\right). \tag{5.8}$$

For example, we may plug in the coordinates $x_1 = 0.5$, $x_2 = 1$ for the optimizer for Example 5.2 into (5.7), to get $u_1 = u_2 = u_3 = 0$, $v_1 = \frac{13-\sqrt{173}}{2}$, $u_0 = -4v_1$, and $u_4 = -9v_1$. The matrix (5.8) corresponding to these values becomes approximately

$$\left(\begin{array}{cc|c|cccc|c} 0 & 0 & -1 & -1.0 & 0 & 1.0 & 0 & -4.0000 \\ 0 & -0.6118 & -2 & 0 & -1 & 0 & 1 & 1.0000 \\ \hline 0 & 0 & 0 & -1.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.5 & 0 & 0 \\ 0 & -0.6883 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline -4 & 1.0000 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1.0 & 1 & 1.0 & 1 & -0.1529 \end{array}\right),$$

with condition number (in the 2-norm) of approximately 11.8.

Method	# Boxes	CPU time
RUN_ROOTS_DELETE	941	20.21
RUN_GLOBAL_OPTIMIZATION with (5.7)	10	3.02
RUN_GLOBAL_OPTIMIZATION with (5.3)	11	0.16

Table 5.3 Summary of 3 methods of handling constraints for Example 5.2

When dealing with the system (5.7), initial bounds of $[0, 1]$ may be used for u_j, $0 \leq j \leq q$, and bounds of $[-1, 1]$ may be used for v_i, $1 \leq i \leq m$. When these bounds were used for Example 5.2 in the program RUN_ROOTS_DELETE (cf. §4.4 on page 159), three critical points (including the two optima) were found with 941 boxes (in 8-dimensional space) considered and 20.21 CPU seconds on a Sun Sparc 20 model 51. When the general optimization code RUN_GLOBAL_OPTIMIZATION (see §5.3 below), configured to use (5.7) in interval Newton iteration, was used, then both optimizers were found with a total of 10 boxes (in 2-dimensional space, but an 8-dimensional interval Newton method). If RUN_GLOBAL_OPTIMIZATION was configured to use (5.3) instead of (5.7), a total of 11 boxes (in 2 dimensions, but a 4-dimensional interval Newton method) were considered, in only about 5% of the CPU time. These results are summarized in Table 5.3.

Additional experiments in [121] corroborate the above: It seems to usually be better not to include the bound constraints in the Fritz John function. These results may be different in high-dimensional problems in which the "peeling" process of §5.2.3 leads to a combinatorial explosion in the number of boxes.

In verifying points obtained by floating point constrained optimization software, approximate Lagrange multiplier values may be available along with the approximate optimizers.

5.2.6 Exercises

1. Draw a picture analogous to Figure 5.5 for the problem:

$$\begin{aligned} \text{minimize} \quad \phi(x) &= -\left(x_1^2 + x_2^2\right) \\ \text{subject to} \quad c(x) &= x_2 - x_1 = 0, \\ -1 \leq \quad x_1 &\leq 1 \end{aligned}$$

$$-1 \leq \quad x_2 \quad \leq 1,$$

$\check{X} = (1,1)^T$.

2. Check the statements below Example 5.3 by actually performing the computations.

   (Hint: The interval Newton method can be done with the program

   RUN_INTERVAL_NEWTON

   from INTOPT_90. *The computations were done using the* C^W-preconditioner *and interval Gauss–Seidel method.)*

3. Fill in the details of the computations below (5.8).

5.3 DESCRIPTION OF PROVIDED SOFTWARE

Similar to RUN_ROOTS_DELETE of §4.4 on page 159, the stand-alone program

RUN_GLOBAL_OPTIMIZATION

in INTOPT_90 runs a global search algorithm for constrained optimization. The program RUN_GLOBAL_OPTIMIZATION uses the configuration files

OVERLOAD.CFG, INTNEWT.CFG, and OPTTBND.CFG.

The file OVERLOAD.CFG, explained on page 85, defines the format for generation and subsequent use of the code list for the objective function, gradient, and constraints, while the files OPTTBND.CFG and INTNEWT.CFG contain various options for tolerances and limits, switches for algorithm variants, and integers that control the amount of printing in various parts of the algorithm.

5.3.1 Use

Solving an optimization problem with RUN_GLOBAL_OPTIMIZATION involves the following steps.

1. Create a code list for the objective function and constraints.
2. Symbolically differentiate the code list to produce a gradient code list.

3. Create the box data file.

4. Run RUN_GLOBAL_OPTIMIZATION.

Creating the Code List

Creation of the code list is as in §2.2.2 and (for solving nonlinear systems) in §4.4.1. Namely, a program is written to evaluate the objective function and constraints, but the program uses special code list variables for the independent and dependent variables. The dependent variables are in an array declared to be of type CDLVAR, the objective function value is stored in a variable of type CDLLHS, and the left-hand-sides of the equality constraints $C(X) = 0$ are stored in an array of type CDLINEQ.

For example, to create a code list for the optimization problem

$$\begin{array}{lrcl} \text{minimize} & & & (x_1 - 10)^3 + (x_2 - 20)^3 \\ \text{subject to} & c_1(X) & = & 100 - (x_1 - 5)^2 - (X_2 - 5)^2 + X_3 = 0, \\ & c_2(X) & = & (x_1 - 6)^2 + (x_2 - 5)^2 - 82.81 + x_4, = \ 0 \\ & x_1 & \geq & 13, \\ & x_i & \geq & 0, \quad i = 2, 3, 4, \end{array} \tag{5.9}$$

the program

```
PROGRAM GOULD
  USE OVERLOAD
      PARAMETER (NN=2)
      PARAMETER (NSLACK=2)
      TYPE(CDLVAR), DIMENSION(NN+NSLACK):: X
      TYPE(CDLLHS), DIMENSION(1):: PHI
      TYPE(CDLINEQ), DIMENSION(2):: C
      OUTPUT_FILE_NAME='GOULD.CDL'
      CALL INITIALIZE_CODELIST(X)

  PHI(1) = (X(1)-10)**3 + (X(2)-20)**3
  C(1) = 100 - (X(1)-5)**2 - (X(2)-5)**2 + X(3)
  C(2) = (X(1)-6)**2 + (X(2)-5)**2 -82.81D0 + X(4)

      CALL FINISH_CODELIST
END PROGRAM GOULD
```

could be run, to produce the code list file GOULD.CDL.

Differentiating the Code List

RUN_GLOBAL_OPTIMIZATION uses a gradient code list, so the code list generated as above must be differentiated, as described on page 91. In particular, the code list may be differentiated with the program MAKE_GRADIENT from INTOPT_90. If the executable program corresponding MAKE_GRADIENT is named **testgrad**, then the following terminal session (on a Unix-based machine) will differentiate the file GOULD.CDL created as above. (The "%" represents the command prompt.)

```
% testgrad
 Input the code list file name without its assumed suffix "CDL"
GOULD
%
```

The derivative code list will be placed in GOULDG.CDL.

The Box Data File

The name of the box data file is of the form

<filename>.DT?

where <filename> is any valid file name, and ? is a single digit. It is structured as follows.

1. The first lines are as in the box data files of §4.4.1, that is, the first line contains a domain tolerance, while lines through $n + 1$ contain the lower and upper bounds of the search region.
2. Lines $n + 2$ through $2n + 1$ contain logical variables that indicate which of the lower and upper bounds represent bound constraints and which merely represent limits on the search region.
3. Lines $2n + 2$ through $3n + 2$ are optional. Lines $2n + 2$ through $3n + 1$ contain a guess for an approximate global optimizer. Line $3n + 2$ contains a logical variable that indicates whether the approximate optimizer should be corrected by the approximate optimization code.

For example, if the search region for the problem (5.9) is to be

$$\boldsymbol{X} = ([13, 15], [0, 0.9], [0, 25], [0, 24])^T,$$

```
1D-5
13 15
0 0.9
0 25
0 24
T F
T F
T F
T F
14.095D0
.842960788D0
0
0
T
```

Figure 5.7 Box data file GOULD.DT1

and if an initial guess that does not need further correction is

$$\check{X} = (14.095, 0.842960788, 0, 0)^T,$$

then the data file will be as in Figure 5.7.

Running RUN_GLOBAL_OPTIMIZATION

Once a gradient code list, such as GOULDG.CDL, and a box data file, such as GOULD.DT1, are created, RUN_GLOBAL_OPTIMIZATION can be run. Assuming the executable corresponding to RUN_GLOBAL_OPTIMIZATION is called "opttest," the following terminal session will solve Problem 5.9.

```
% opttest
 Input the gradient code list file name.
 Do not include its assumed suffix "*G.CDL"
GOULD
 Input a digit representing the data file number:
1
%
```

This produces the output files GOULD.OT1, the general output file, and the file OPTTEST.TBL, a file with performance data that can be imported into a spreadsheet. Created on a Sparc 20, selected portions of the output file GOULD.OT1, corresponding to configuration values as in §5.3.2 below are:

```
Output from RUN_GLOBAL_OPTIMIZATION on 12/26/1995 at 19:09:42.
DATA WAS TAKEN FROM DATA FILE: WOLFE1.DT1

Initial box:
Box coordinates:
0.13000000000000000D+02 0.15000000000000000D+02
0.00000000000000000D+00 0.90000000000000022D+00
0.00000000000000000D+00 0.25000000000000000D+02
0.00000000000000000D+00 0.24000000000000000D+02

BOUND_CONSTRAINT:
T F T F T F T F

---------------------
CONFIGURATION VALUES:

EPS_DOMAIN: 0.1000D-05 MAXITR: 60000
DO_INTERVAL_NEWTON: T QUADRATIC: T FULL_SPACE: F
VERY_GOOD_INITIAL_GUESS: T
USE_SUBSIT: F
THERE WERE NO BOXES IN COMPLETED_LIST.

LIST OF BOXES CONTAINING VERIFIED FEASIBLE POINTS:

Box no.: 1
Box coordinates:
0.13954050000000005D+02 0.14235950000000008D+02
0.88296078799999952D+00 0.85296078799999970D+00
0.00000000000000000D+00 0.10000000000000002D-01
0.00000000000000000D+00 0.10000000000000002D-01

PHI:
-0.69796789071351713 6D+04 -0.69434722111306364 2D+04
B%LIUI(1,*):
F F F F
B%LIUI(2,*):
F F F F
B%SIDE(*):
F F F F
B%PEEL(*):
F F F F
Level: 0
Box contains the following approximate root:
0.14095000000000006D+02 0.84296078799999961D+00
0.00000000000000000D+00 0.00000000000000000D+00

OBJECTIVE ENCLOSURE AT APPROXIMATE ROOT:
-0.69618138769183588 0D+04 -0.69618138769183442 5D+04
Unknown = T Contains_root = T
Changed coordinates:
F F F F
-------------------------
ALGORITHM COMPLETED WITH LESS THAN THE MAXIMUM NUMBER, 60000 OF BOXES.
No. dense interval residual evaluations - gradient code list: 110
Number of orig. system inverse midpoint preconditioner rows: 2
Number of orig. system C-LP preconditioner rows: 142
Total number of forward_substitutions: 648
Number of Gauss-Seidel steps on the dense system: 144
Number of gradient evaluations from a gradient code list: 8
Total number of dense slope matrix evaluations: 110
Total number second-order interval evaluations of the
original function: 41
Total number dense interval constraint evaluations: 274
Total number dense interval constraint gradient component evaluations: 486
Total number dense interval reduced gradient evaluations: 101
Total number of calls to FRITZ_JOHN_RESIDUALS: 37
Number of times a box was rejected in the interval Newton
method due to an empty intersection: 2
Number of times the interval Newton method made a coordinate
interval smaller: 99
Number of times a box was rejected because the constraints were not satisfied: 4
Number of times a box was rejected because the gradient or
reduced gradient did not contain zero: 10
Total time spent doing linear algebra (preconditioners
and solution processes): 0.1799999177455902
Number of times the approximate solver was called: 1
Number Fritz-John matrix evaluations: 27
BEST_ESTIMATE: -6.9618129718904111E+03
Total number of boxes processed in loop: 16
Overall CPU time: 0.7099999859929085
CPU time in PEEL_BOUNDARY: 0.1300000026822090
CPU time in REDUCED_INTERVAL_NEWTON: 0.3599999845027924
```

5.3.2 The Configuration File OPTTBND.CFG

OPTTBND.CFG contains two long records. The first record, containing an explanatory header, is ignored, while the second record, input in list-directed format, contains variables that determine how much printing is done and which algorithms are used. The variables on the second line occur in the following order.

PRINT_OPTIMIZATION_TEST (integer, values 0–5) determines how much printing is done in the overall routine RUN_GLOBAL_OPTIMIZATION. Smaller values mean less printing.

PRINT_REDUCED_INTERVAL_NEWTON (integer, values 0–5) determines how much printing is done in whatever interval Newton method is used.

PRINT_REDUCED_GS (integer, values 0–5) determines how much printing is done in the interval Gauss–Seidel method.

PRINT_LANCELOT_OPT (integer, values 0–5) determines how much printing is done in the interface routine LANCELOT_OPT to the approximate optimization solver.

PRINT_FIND_APPROX_OPT (integer, values 0–5) determines how much printing is done in an overall routine FIND_APPROX_OPT that finds an approximate optimum, executes algorithms 17, 18, and 19, and does an ϵ-infiation to construct a small box verified to contain a feasible point.

PRINT_CERTAINLY_FEASIBLE (integer, values 0–5) determines how much printing is done in the routine that verifies if a feasible point exists in a certain box.

PRINT_CONSTRUCT_FEASIBLE (integer, values 0–5) determines how much printing is done in the routine CONSTRUCT_FEASIBLE_REGION that actually performs the ϵ-infiation and verifies that the resulting region contains a feasible point.

PRINTBDY (integer, values 0–5) determines how much printing is done in the routine PEEL_BOUNDARY_DENSE, in which Algorithm 15 is implemented.

PRINT_COMPLEMENT_LIST (integer, values 0–5) determines how much printing is done in the routine COMPLEMENT_DENSE_OPT in which Algorithm 10 is implemented for optimization.

PRINT_INFLATE (integer, values 0–5) determines how much printing is done in the routine INFLATE_DENSE_OPT_SLOPE that constructs a box about an approximate optimum in which a unique critical point of the Fritz John equations can be proven to lie.

MAXITR (integer) is the maximum number of boxes allowed to be process in the overall algorithm (i.e. the maximum number of times the body of the loop in Step 2b of Algorithm 13 is to be executed).

DO_INTERVAL_NEWTON (logical) is set to "T" if interval Newton steps are to be used to reduce the size of boxes, (i.e. if Step 6 of Algorithm 14 is to be done), and is set to "F" otherwise.

DO_MIDPOINT_TEST (logical) is set to "T" if the midpoint test, (i.e. Step 2 of Algorithm 14) is to be done, and is set to "F" otherwise.

QUADRATIC (logical) is set to "T" if the quadratic normalization conditions for the Fritz John equations (i.e. (5.3) and (5.4)) is to be used, and is set to "F" if the linear, interval normalization (i.e. (5.5) and (5.6)) is to be used.

FULL_SPACE (logical) is set to "T" if the bound constraints are to be included in the Fritz John equations, and is set to "F" otherwise.

EPS_DOMAIN (real) is the domain tolerance ϵ_d used to determine minimum box sizes in the tessellation and inflation[10].

USE_GRADIENT_TEST (logical) is set to "T" if the monotonicity test (i.e. Step 4 of Algorithm 14) is to be used, and is set to "F" otherwise.

USE_SUBSIT (logical) is set to "T" if the user-defined version of the substitution-iteration process of Chapter 7 is to be used, and is set to "F" otherwise.

An example OPTTBND.CFG file, split into three sets of columns for display, is as follows:

first five columns:

```
prnt ctrl: OPT._TEST, RED._INT._NWT, REDGS, PRINT_LANCE, FIND_APPROX_OPT
             0            0            0       0             0
```

middle seven columns:

```
 CERT._FEAS., PRINT_CONSTR., PRINTBDY, PRNTCMPL, INFLATE  MAXITR newton?
    0               0           0         0         0      60000     T
```

last six columns:

```
 midpt? quad, full, epsx USE_GRADIENT_TEST USE_SUBSIT
  T     T     F     1d-6 T                  F
```

These settings should be good for many problems.

5.3.3 Supplying a Local Constrained Optimizer

The local approximate constrained optimizer has a major effect on the overall performance of RUN_GLOBAL_OPTIMIZATION. The program is originally developed with the optimization package LANCELOT [35], not supplied with INTOPT_90 but is available for research purposes from the authors.

Interfacing other local approximate constrained optimizers with RUN_GLOBAL_OPTIMIZATION involves modifying interface routine LANCELOT_OPT. The argument list to LANCELOT_OPT is:

[10]This value overrides the value in the box data file. The quantity appears in the box data file to make the format compatible with other uses of box data.

```
SUBROUTINE LANCELOT_OPT(INITIAL_GUESS_, GOOD_INITIAL_GUESS_, &
                        INTERVAL_X, LOWER_BOUND_CONSTRAINT,&
                        UPPER_BOUND_CONSTRAINT, OPTIMIZER, INFO)
   USE INTERVAL_ARITHMETIC
   USE CODELIST_VARIABLES
   USE ROOT_SOLVER_STATISTICS
   USE ROOTS_DELETE_PARAMETERS

   IMPLICIT NONE
      DOUBLE PRECISION INITIAL_GUESS_(:)
      LOGICAL GOOD_INITIAL_GUESS_
      TYPE(INTERVAL), DIMENSION(:) :: INTERVAL_X
      LOGICAL, DIMENSION(:) :: LOWER_BOUND_CONSTRAINT, &
                               UPPER_BOUND_CONSTRAINT
      DOUBLE PRECISION OPTIMIZER(:)
      INTEGER INFO
```

where

`INITIAL_GUESS_` is an n-vector containing an initial guess for the local optimizer.

`GOOD_INITIAL_GUESS_` is set to "T" if the initial guess in `INITIAL_GUESS_` is thought to be accurate.

`INTERVAL_X` is the box $X24$, some of whose bounds represent bound constraints.

`LOWER_BOUND_CONSTRAINT` The I-th component is set to "T" if and only if `INTERVAL_X(I)%LOWER` represents a bound constraint, and not just a limit on a search region.

`UPPER_BOUND_CONSTRAINT` The I-th component is set to "T" if and only if `INTERVAL_X(I)%UPPER` represents a bound constraint, and not just a limit on a search region.

`OPTIMIZER` is an n vector that contains the optimizer, upon return.

`INFO` is set to

- 0 for a successful return;
- -1 if it appears that the problem has not been solved (such as when the problem seems infeasible, or when the maximum number of iterations has been exceeded).
- 1 if the problem may have been solved, but some numerical difficulty, such as a small step size or trust region radius, was encountered.

In addition to the arguments, various global variables, including the following, are available.

`NSMALL`: the number of independent variables;

`NINEQ/(NSMALL+1)`: the number of constraints;

EPS_DOMAIN: the domain tolerance ϵ_d;

USE_INITIAL_GUESS: set in other routines in INTOPT_90 to "T" if it is permissible to simply return INITIAL_GUESS_ in OPTIMIZER, without calling the local optimization procedure.

N_APPROX_SOLVER: a counter that should be incremented by 1 each time LANCELOT_OPT is called.

It is recommended that the tolerance passed to the local optimization package be approximately $\epsilon_d^{1.5}$. This is so the local optimizer returned by the local optimization package will be accurate relative to the size of the boxes constructed with ϵ-inflation.

Other building blocks in INTOPT_90, operating with gradient code lists, that may be of use are:

F_POINT_GRADIENT(X,FVAL) that computes point approximations to the objective function ϕ.

POINT_GRADIENT_G(X,GVAL) that computes a point approximation for the gradient at the point X.

F_POINT_GRADIENT(X,FVAL) that computes point approximations to the objective function ϕ.

POINT_HESSIAN_G(X, FP) that computes point approximations to the Hessian matrix of ϕ.

POINT_CONSTRAINTS(X,CVAL) that computes point approximations to the constraint values $C(X)$.

POINT_CONSTRAINT_GRADIENTS(X,GMAT) computes approximations to $\nabla C(X)$.

Also see pages 99 *ff* in Chapter 2.

5.3.4 Installation

The program RUN_GLOBAL_OPTIMIZATION comes with INTOPT_90. Obtaining and installing INTOPT_90 is as in §4.4.3.

5.3.5 Future Improvements

It is planned to constantly improve the program RUN_GLOBAL_OPTIMIZATION and corresponding subroutines, and to periodically place updated versions on interval.usl.edu. The following improvements are envisioned in the near future (by publication of this work or soon thereafter).

infeasibility of inequality constraints: Presently, RUN_GLOBAL_OPTIMIZATION does not use interval extensions $\boldsymbol{g}(\boldsymbol{X}) > 0$ of an inequality constraint $g(X) \leq 0$ to verify that g is infeasible over a box $\boldsymbol{X}$.

better use of the Fritz John equations: The full Fritz John equations can be used to verify existence and uniqueness of a critical point in as large a box as possible about a good approximation to a global optimum (analogously Algorithm 9 on page 151). Doing so would allow stronger statements in the output to RUN_GLOBAL_OPTIMIZATION, and possibly also improve the program's efficiency.

a more appropriate approximate optimization package: The routine LANCELOT_OPT was initially chosen for convenience and because it handled bound constraints. However, experimentation has hinted that other approximate optimization packages may be more appropriate. Also, LANCELOT_OPT does not have an unrestricted license.

6

NON-DIFFERENTIABLE PROBLEMS

In previous chapters, interval extensions and interval Newton methods were developed for verified global solution of nonlinear systems of equations and for global optimization. In Chapters 1 and 2, it was shown how to compute and use such interval extensions and interval Newton methods when the functions were given by smooth expressions, without conditional branches. In fact, however, many practical problems, in particular those containing expressions such as $|E(X)|$ and $\max\{E(X), F(X)\}$, $E, F : \mathbb{R}^n \to \mathbb{R}$, or functions defined by `IF-THEN-ELSE` branches, result in functions whose derivatives have jump discontinuities. However, if the function itself is continuous, order-1 interval extensions can be computed within the automatic differentiation framework of §1.4 and §2.2.

Such interval extensions of non-smooth functions can be used in the same contexts as other interval extensions. However, the width of such a derivative extension $\boldsymbol{F}'(\boldsymbol{X})$ does not, in general, tend to zero as $\mathrm{w}(\boldsymbol{X})$ tends to zero. Because of this, interval Newton iterations are only convergent in certain cases. However, ϵ-inflation algorithms can always be devised to prove existence or uniqueness for $\mathrm{w}(\boldsymbol{X})$ sufficiently small, provided merely that $\boldsymbol{F}'(\boldsymbol{X})$ is extended as described in this Chapter.

Special algorithms can be developed to handle non-smooth problems such as l_1 optimization and l_∞ optimization. However, simplicity is a major advantage of treating such non-smooth problems with the same techniques as smooth problems. This simpler, unified treatment is possible within the context described in this chapter.

First, required properties of interval extensions for non-smooth functions are discussed. Then, formulas appropriate for interval evaluation and symbolic differentiation are presented. Finally, a convergence and existence / uniqueness verification theory for interval Newton methods using such extensions is developed.

6.1 EXTENSIONS OF NON-SMOOTH FUNCTIONS

6.1.1 Required Properties

The crucial property of all such extensions is that they contain the range, namely, that they are actually interval extensions in the sense of Definition 1.3 on page 12. An interval extension of a derivative must be a Lipschitz matrix in the sense of Definition 1.10 on page 26. Similarly, an interval extension of a slope must be a slope matrix in the sense of Definition 1.12 on page 27.

Such interval extensions should be as *sharp as possible* (i.e. should overestimate actual ranges as little as possible) subject to the above conditions, and subject to the condition that they can be computed automatically. The formulas in the next section give such extensions.

6.1.2 Formulas

These formulas first appeared in [118]. They fall into the following groups:

1. rules for floating-point evaluation of the operation;
2. rules for floating-point evaluation of the derivative of the operation;
3. rules for interval extensions of the operation;
4. rules for symbolic differentiation of the operation;
5. rules for interval evaluation of the derivative of the operation, assuming the operation represents a continuous function, such as the absolute value;
6. rules for interval evaluation of the derivative of the operation, assuming the operation does not represent a continuous function, such as a function

```
PROGRAM CHI_EXAMPLE
  USE OVERLOAD
      TYPE(CDLVAR), DIMENSION(1):: X
      TYPE(CDLLHS), DIMENSION(1):: F
      OUTPUT_FILE_NAME='CHI_EXAMPLE.CDL'
      CALL INITIALIZE_CODELIST(X)
  F(1) = CHI(X(1)-1,X(1)**2,2*X(1)-1)
      CALL FINISH_CODELIST
END PROGRAM CHI_EXAMPLE
```

Figure 6.1 Program that generates a code list for $f(x) = x^2$ if $x < 1$, $f(x) = 2x - 1$ if $x \geq 1$

defined by separate formulas in separate intervals, with unmatching values at the break point;

7. rules for interval evaluation of slopes, assuming the operation represents a continuous function; and

8. rules for interval evaluation of slopes, assuming the operation does not represent a continuous function.

Here, each of these rules is presented for each of the functions χ (IF-THEN-ELSE branches), $|\circ|$ and max.

Formulas for IF-THEN-ELSE *Branches*

Introduced in [117], the function $x_p = \chi(x_s, x_q, x_r)$ is the device in INTLIB_90 that generates code lists corresponding to IF-THEN-ELSE branches. For floating-point values, χ is defined by

Formula 6.1 *Floating-point evaluation*

$$\chi(x_s, x_q, x_r) = \begin{cases} x_q & \textit{if } x_s < 0, \\ x_r & \textit{otherwise.} \end{cases}$$

For example, if

$$f(x) = \begin{cases} x^2 & \text{if } x < 1 \\ 2x - 1 & \text{if } x \geq 1, \end{cases} \tag{6.1}$$

then a code list for f is generated by the program in Figure 6.1.

Evaluation of the point derivative is straightforward.

OP no.	p	q	r	s	a	b
23	2	1	·	·	·	$[-1,-1]$
5	3	1	·	·	·	·
22	4	1	·	·	$[2,2]$	·
23	5	4	·	·	·	$[-1,-1]$
27	6	3	5	2	·	·
18	6	·	·	·	·	·

Figure 6.2 Operation lines of the code list produced by the program of Figure 6.1

Formula 6.2 *Floating-point evaluation of the derivative*

$$\frac{\partial\chi(x_s,x_q,x_r)}{\partial x_q} = \begin{cases} 1 & \text{if } x_s < 0; \\ 0 & \text{otherwise.} \end{cases}$$

$$\frac{\partial\chi(x_s,x_q,x_r)}{\partial x_r} = \begin{cases} 0 & \text{if } x_s < 0; \\ 1 & \text{otherwise.} \end{cases}$$

The following formula leads to an interval extension in the sense of Definition 1.3.

Formula 6.3 *Interval evaluation*

$$\chi(\boldsymbol{x}_s,\boldsymbol{x}_q,\boldsymbol{x}_r) = \begin{cases} \boldsymbol{x}_q & \text{if } \boldsymbol{x}_s < 0; \\ \boldsymbol{x}_r & \text{if } \boldsymbol{x}_s > 0; \\ \boldsymbol{x}_q \cup \boldsymbol{x}_r & \text{otherwise.} \end{cases}$$

When used to produce derivative code lists, the following formula leads to interval extensions of the derivative.

Formula 6.4 *Symbolic differentiation*

$$\chi'(x_s,x_q,x_r) = \chi(x_s,x_q',x_r')$$

Formula 6.4, the usual way of differentiating branches, can be combined with Formula 6.3 in cases where $0 \in \boldsymbol{x}_q$, or when $\boldsymbol{x}_q = 0$, even in computations that are otherwise floating point. If this is done, then the "if problem" noted by Beck and Fischer [17] does not occur. In particular, the interval derivative will always contain the correct derivative.

When used to compute interval enclosures for the derivative, the following two formulas lead to Lipschitz sets.

Formula 6.5 *Interval evaluation of $\partial\chi/\partial x_q$, $\partial\chi/\partial x_r$ and $\partial\chi/\partial x_s$ when χ is continuous in x_s, i.e. when possibly $x_q = x_r$ whenever $x_s = 0$.* (This formula may be applied in a reverse automatic differentiation process.)

$$\frac{\partial\chi(x_s, x_q, x_r)}{\partial x_q} = \begin{cases} 1 & \text{if } x_s < 0; \\ 0 & \text{if } x_s > 0; \\ [0,1] & \text{otherwise.} \end{cases}$$

$$\frac{\partial\chi(x_s, x_q, x_r)}{\partial x_r} = \begin{cases} 0 & \text{if } x_s < 0; \\ 1 & \text{if } x_s > 0; \\ [0,1] & \text{otherwise.} \end{cases}$$

$$\frac{\partial\chi(x_s, x_q, x_r)}{\partial x_s} = 0$$

Formula 6.6 *Interval evaluation of $\partial\chi/\partial x_q$, $\partial\chi/\partial x_r$ and $\partial\chi/\partial x_s$ when χ is possibly discontinuous in x_s i.e. when possibly $x_q \neq x_r$ at some places where $x_s = 0$* (appropriate for a reverse automatic differentiation process). *The formulas are the same as Formula 6.5 except:*

$$\frac{\partial\chi(x_s, x_q, x_r)}{\partial x_q} = (-\infty, \infty) \text{ if } 0 \in x_s$$

$$\frac{\partial\chi(x_s, x_q, x_r)}{\partial x_r} = (-\infty, \infty) \text{ if } 0 \in x_s$$

The following formula leads to slope matrices in the sense of Definition 1.12.

Formula 6.7 *Interval evaluation of the slope $\mathbf{S}(\chi, \boldsymbol{X}, \check{\boldsymbol{X}})$ when χ is continuous in x_s* (appropriate for a forward automatic differentiation process)

$$\mathbf{S}(\chi(x_s, x_q, x_r), \boldsymbol{X}, \check{\boldsymbol{X}}) = \begin{cases} \mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } x_s \cup \check{x}_s < 0; \\ \mathbf{S}(x_r, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } x_s \cup \check{x}_s > 0; \\ \mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) \cup \mathbf{S}(x_r, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{otherwise.} \end{cases}$$

Amazingly, meaningful slopes may be computed even when the function is discontinuous. This is important in computations such as l_1 or l_∞ optimization, since the objective function ϕ typically is continuous but non-smooth, and the gradient therefore has a discontinuity. However, it is necessary to use this gradient, and not a slope, when searching for critical points of ϕ. In an interval Newton method, a derivative of such a gradient would then be required.

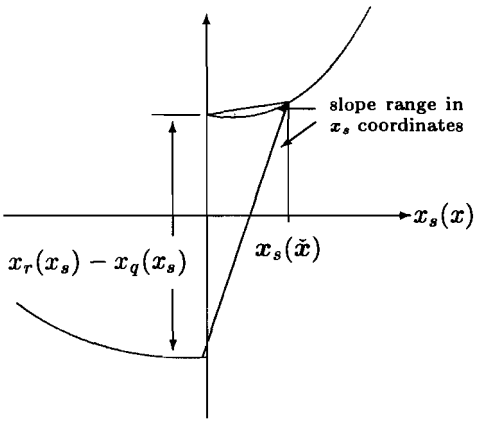


Figure 6.3 Computation of slope bounds of a discontinuous function

Example 6.1 *If*

$$\phi(x) = \max\{2 - x^2, x^2\}, \tag{6.2}$$

then ϕ has a minimum at $x = 1$. This minimum is at a cusp of ϕ, where the gradient

$$\nabla\phi(x) = \chi\big(x^2 - (2 - x^2), -2x, 2x\big) \tag{6.3}$$

has a jump discontinuity. Thus, the interval extension of the second derivative of ϕ (corresponding to the Hessian matrix) is $\mathbb{R}$.

Definition of the interval slopes of discontinuous functions, first explained in [122], is illustrated in Figure 6.3. In Figure 6.3, the slope

$$\mathbf{S}(\chi(x_s(x), x_q(x), x_r(x)), \boldsymbol{x}, \check{\boldsymbol{x}})$$

is shown in terms of x_s coordinates, that is, assuming $x_s = x$. It is seen that, if $0 \notin \check{x}$, the slope bound is finite. Furthermore, the slope bounds will include the jump (a finite change over an infinitely small interval), and interval Newton methods, if they converge, will converge on critical points, be they zeros or breaks in the gradient.

Based on Figure 6.3, the following formula can be derived.

Formula 6.8 *Interval evaluation of the slope* $\mathbf{S}(\chi, \boldsymbol{X}, \check{\boldsymbol{X}})$ *when* χ *is discontinuous in* x_s (appropriate for a forward automatic differentiation process). *The*

formula is the same as Formula 6.7 when $0 \notin \boldsymbol{x}_s \cup \check{\boldsymbol{x}}_s$. *When* $0 \in \boldsymbol{x}_s \cup \check{\boldsymbol{x}}_s$ *(as in Figure 6.3), the following is used.*

$$\mathbf{S}(\chi(x_s(X), x_q(X), x_r(X)), \boldsymbol{X}, \check{\boldsymbol{X}}) =$$

$$\begin{cases} \mathbf{S}(x_r(X), \boldsymbol{X}, \check{\boldsymbol{X}}) \cup \left\{ \frac{x_r(X) - x_q(X)}{[\underline{x}_s(\check{\boldsymbol{X}}), \overline{x}_s(\check{\boldsymbol{X}}) - \underline{x}_s(X)]} \mathbf{S}(x_s(X), \boldsymbol{X}, \check{\boldsymbol{X}}) \right\} & \text{if } \boldsymbol{x}_s(\check{\boldsymbol{X}}) > 0; \\ \mathbf{S}(x_q(X), \boldsymbol{X}, \check{\boldsymbol{X}}) \cup \left\{ \frac{x_r(X) - x_q(X)}{[-\overline{x}_s(\check{\boldsymbol{X}}), \overline{x}_s(X) - \underline{x}_s(\check{\boldsymbol{X}})]} \mathbf{S}(x_s(X), \boldsymbol{X}, \check{\boldsymbol{X}}) \right\} & \text{if } \boldsymbol{x}_s(\check{\boldsymbol{X}}) < 0; \\ \left\{ \left[\frac{1}{\overline{x}_s(\check{\boldsymbol{X}}) - \underline{x}_s(X)}, \infty \right) \cup \left[\frac{1}{\overline{x}_s(X) - \underline{x}_s(\check{\boldsymbol{X}})}, \infty \right) \right\} \big(x_r(X) - x_q(X)\big) \cdot \\ \qquad \mathbf{S}(x_s(X), \boldsymbol{X}, \check{\boldsymbol{X}}) \\ \qquad\quad \cup \mathbf{S}(x_q(X), \boldsymbol{X}, \check{\boldsymbol{X}}) \cup \mathbf{S}(x_r(X), \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } 0 \in \boldsymbol{x}_s(\check{\boldsymbol{X}}). \end{cases}$$

The first branch of Formula 6.8, for $\boldsymbol{x}_s(\check{\boldsymbol{X}}) > 0$, corresponds to Figure 6.3: The factor $1/\big[\boldsymbol{x}_s(\check{\boldsymbol{X}}), \boldsymbol{x}_s(\check{\boldsymbol{X}}) - \underline{x}_s(\boldsymbol{X})\big]$ represents the distance of $\boldsymbol{x}_s(\check{\boldsymbol{X}})$ to points on the other side of the break point, while $\big(\boldsymbol{x}_r(\boldsymbol{X}) - \boldsymbol{x}_q(\boldsymbol{X})\big)$ represents the jump at the break point, so the fraction $\big(\boldsymbol{x}_r(\boldsymbol{X}) - \boldsymbol{x}_q(\boldsymbol{X})\big)/\big[\boldsymbol{x}_s(\check{\boldsymbol{X}}), \boldsymbol{x}_s(\check{\boldsymbol{X}}) - \underline{x}_s(\boldsymbol{X})\big]$ represents the slope of the line from the point above $\boldsymbol{x}_s(\check{\boldsymbol{X}})$ to points on the other side of the break point. The factor $\mathbf{S}(x_s(X), \boldsymbol{X}, \check{\boldsymbol{X}})$ comes from the analogue of the chain rule for slopes. The part $\mathbf{S}(x_r(X), \boldsymbol{X}, \check{\boldsymbol{X}})$ takes account of the portion of the curve to the right of the break point. The other two branches of Formula 6.8 are analogous.

For example, suppose Formula 6.8 is applied to compute $\mathbf{S}(f, \boldsymbol{x}, \check{x})$, where $f(x) = \nabla\phi(x)$ of Example 6.1, with $\boldsymbol{x} = [1, 5]$ and $\check{x} = 3$. Then, following the notation in Formula 6.8, we have $x_s(x) = x^2 - (2 - x^2)$, $x_q(x) = -2x$, and $x_r(x) = 2x$. Thus, $x_s(\check{x}) = 3^2 - (2 - 3^2) = 16 > 0$, so the first branch of Formula 6.8 is taken. We have $1/x_s(\check{x}) = 1/16$, $\boldsymbol{x}_r(\boldsymbol{x}) - \boldsymbol{x}_q(\boldsymbol{x}) = [2, 10] - (-[2, 10]) = [4, 20]$, $\mathbf{S}(\boldsymbol{x}_s(\boldsymbol{x}), \boldsymbol{x}, \check{x}) = [8, 16]$, and $\mathbf{S}(x_r(x), \boldsymbol{x}, \check{x}) = 2$. (Here, the slopes may be computed with Theorem 1.13 (page 41.) The computed slope bound is thus

$$\begin{aligned} \mathbf{S}\Big(\chi\big(x^2 - (x^2 - 2)\big), -2x, 2x\Big) &= 2 \cup \frac{1}{16}([4, 20])[8, 16] \\ &= 2 \cup [2, 20] = [2, 20]. \end{aligned}$$

If this slope bound is used in an interval Newton method, then $f(\check{x}) = 6$, and

$$\tilde{\boldsymbol{x}} = \check{x} - \frac{f(\check{x})}{\mathbf{S}(f, \boldsymbol{x}, \check{x})} = 3 - \frac{6}{[2, 20]} = 3 - [.3, 3] = [0, 2.7].$$

Thus, $\tilde{\boldsymbol{x}} \cap \boldsymbol{x} = [1, 2.7]$, representing a significant reduction in width.

Additional examples will be given below.

Formula 6.8 is a general formula, allowing meaningful slopes of gradients of objective functions containing χ, $|\cdot|$, and max, since the symbolic derivatives of $|\cdot|$ and max are written in terms of χ. (See formulas 6.11 and 6.16 below, respectively.)

Formulas for $x_p = |x_q|$

If $x \in \mathbb{R}$, then $|x| = \chi(x, -x, x)$. However, $\chi(\boldsymbol{x}, -\boldsymbol{x}, \boldsymbol{x})$ overestimates the range of $|\circ|$ over the interval $\boldsymbol{x}$. For example, the range of $|\circ|$ over $[-1, 2]$ is $[0, 2]$, whereas $\chi([-1, 2], [-2, 1], [-1, 2]) = [-2, 2]$. Hence,it is advantageous to consider $|\circ|$ as a separate operation, with the following computation formulas.

Formula 6.9 *Floating-point evaluation of the derivative (well-known; nothing special is done at the break point)*

$$\frac{d|x_q|}{dx_q} = \begin{cases} -1 & \text{if } x_q < 0; \\ 1 & \text{otherwise.} \end{cases}$$

Formula 6.10 *Interval evaluation*

$$|\boldsymbol{x}| = \begin{cases} [0, \max\{|\underline{x}|, |\overline{x}|\}] & \text{if } 0 \in \boldsymbol{x}; \\ [\min\{|\underline{x}|, |\overline{x}|\}, \max\{|\underline{x}|, |\overline{x}|\}] & \text{otherwise.} \end{cases}$$

Formula 6.11 *Symbolic differentiation*

$$|x_q|' = \chi(x_q, -1, 1){x_q}' = \chi(x_q, -x_q', x_q')$$

Formula 6.12 *Interval evaluation of $d|x|/dx$* (appropriate for a reverse automatic differentiation process)

$$\frac{d|\boldsymbol{x}_q|}{dx_q} = \begin{cases} -1 & \text{if } \boldsymbol{x}_q < 0; \\ 1 & \text{if } \boldsymbol{x}_q > 0; \\ [-1, 1] & \text{otherwise.} \end{cases}$$

Formula 6.13 *Interval evaluation of the slope* $\mathbf{S}(|x_q|, \boldsymbol{X}, \check{\boldsymbol{X}})$ (appropriate for forward automatic differentiation processes)

$$\mathbf{S}(|x_q|), \boldsymbol{X}, \check{\boldsymbol{X}}) = \begin{cases} -\mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } x_q \cup \check{x}_q < 0; \\ \mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } x_q \cup \check{x}_q > 0; \\ \mathbf{S}^{(d)}(|x_q|, x_q, \check{x}_q)\mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{otherwise,} \end{cases} \tag{6.4}$$

where

$$\mathbf{S}^{(d)}(|x|, x, \check{x}) = h(\underline{x}) \cup h(\overline{x}) \quad \text{with} \quad h(x) = \begin{cases} \frac{|x|-|\check{x}|}{x-\check{x}} & \text{for } x \notin \check{x}; \\ [-1, 1] & \text{otherwise.} \end{cases} \tag{6.5}$$

The third branch of Formula 6.13 is an application of Theorem 1.13 on page 41.

Formulas for $x_p = \max\{x_q, x_r\}$

For real values x_q and x_r, $\max\{x_q, x_r\} = \chi(x_r - x_q, x_q, x_r)$, but $\chi(\boldsymbol{x}_r - \boldsymbol{x}_q, \boldsymbol{x}_q, \boldsymbol{x}_r)$ overestimates the range of max for interval values $\boldsymbol{x}_q$ and $\boldsymbol{x}_r$. Formulas appropriate for max follow.

Formula 6.14 *Floating-point evaluation of the derivative (well-known)*

$$\frac{\partial \max\{x_q, x_r\}}{\partial x_q} = \begin{cases} 1 & \text{if } x_q > x_r; \\ 0 & \text{otherwise.} \end{cases}$$

$$\frac{\partial \max\{x_q, x_r\}}{\partial x_r} = \begin{cases} 0 & \text{if } x_q > x_r; \\ 1 & \text{otherwise.} \end{cases}$$

Formula 6.15 *Interval evaluation*

$$\max\{\boldsymbol{x}_q, \boldsymbol{x}_r\} = [\max\{\underline{x}_q, \underline{x}_r\}, \max\{\overline{x}_q, \overline{x}_r\}].$$

Formula 6.16 *Symbolic differentiation*

$$\max'(x_q, x_r) = \chi(x_r - x_q, x_q', x_r').$$

Formula 6.17 *Interval evaluation of* $\partial \max / \partial x_q$ *and* $\partial \max / \partial x_r$ (appropriate in a reverse automatic differentiation process)

$$\frac{\partial \max\{x_q, x_r\}}{\partial x_q} = \begin{cases} 1 & \text{if } x_q > x_r; \\ 0 & \text{if } x_q < x_r; \\ [0,1] & \text{otherwise.} \end{cases}$$

$$\frac{\partial \max\{x_q, x_r\}}{\partial x_r} = \begin{cases} 0 & \text{if } x_q > x_r; \\ 1 & \text{if } x_q < x_r; \\ [0,1] & \text{otherwise.} \end{cases}$$

Formula 6.18 *Interval evaluation of the slope* $\mathbf{S}(\max\{x_q, x_r\}, \boldsymbol{X}, \check{\boldsymbol{X}})$ (appropriate for a forward automatic differentiation process)

$$\mathbf{S}(\max\{x_q, x_r\}, \boldsymbol{X}, \check{\boldsymbol{X}}) = \begin{cases} \mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } x_q \cup \check{x}_q > x_r \cup \check{x}_r; \\ \mathbf{S}(x_r, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{if } x_q \cup \check{x}_q < x_r \cup \check{x}_r; \\ \mathbf{S}(x_q, \boldsymbol{X}, \check{\boldsymbol{X}}) \cup \mathbf{S}(x_r, \boldsymbol{X}, \check{\boldsymbol{X}}) & \text{otherwise.} \end{cases}$$

6.1.3 Exercises

1. Suppose $f, g, h : \mathbb{R}^n \to \mathbb{R}$ and $E, F, G : \mathbb{R}^n \to \mathbb{R}$. If

$$f(X) = \chi(E(X), F(X), G(X)),$$

$g(X) = |E(X)|$, and $h(X) = \max\{E(X), F(X)\}$. Show that, if interval extensions of f, g, and h are obtained with the formulas of this section, then f, g, and h are first-order extensions in the sense of Definition 1.4 on page 14.

2. Let f, g and h be as in Exercise 1. Show that, if derivative and slope extensions of f, g and h are obtained with the formulas of this section, then these formulas lead to Lipschitz matrices in the case of derivative extensions and slope matrices in the case of slope extensions.

6.2 USE IN INTERVAL NEWTON METHODS

The consequences of using the above formulas in the nonlinear equations and global optimization algorithms of Chapters 4 and 5 are explored here.

6.2.1 Objective Function Approximation

It is not hard to show that, computed with the formulas of §6.1.2, interval extensions involving χ, $|\circ|$, and max are first-order extensions (Exercise 1 above). Thus, these interval extensions may be used to prove that no roots exist within a box $\boldsymbol{X}$. Similarly, since the formulas for the derivative extensions, if they lead to finite intervals, lead to Lipschitz matrices or slope matrices, an image of $\boldsymbol{X}$ under any interval Newton method that employed these extensions must contain all roots within $\boldsymbol{X}$ (Exercise 2 above). Hence, the interval Newton method can also be used to prove that no roots exist within $\boldsymbol{X}$.

However, the mean value extension based on the above does not, in general, lead to a second-order interval extension. For example, if $x = [-\epsilon/2, \epsilon]$, and $\check{x} = (-\epsilon/2 + \epsilon)/2 = \epsilon/4$, then Formula 6.12 gives a mean value extension of

$$|[-\epsilon/2, \epsilon]| \subseteq \epsilon/4 + [-1, 1][-3\epsilon/4, 3\epsilon/4] = [-\epsilon/2, \epsilon],$$

an overestimation of $\epsilon/2$ of the range, regardless of ϵ. A somewhat analogous phenomenon, due to the fact that $\mathrm{w}(\boldsymbol{F}'(\boldsymbol{X}))$ does not tend to zero in general, holds for interval Newton methods. This is discussed in the next section.

6.2.2 Convergence Theory

Existence and uniqueness theory based on interval Newton methods was discussed in §1.5.2, while convergence theorems for interval Newton methods appeared as Theorem 1.14 and in Exercise 3 on page 65. Combined, the existence-uniqueness and convergence results give computational analogues of the Kantorovich theorem.

Convergence and existence or uniqueness verification with interval Newton methods have, in the past, been viewed as practical only when there are smooth first derivatives. However, convergence occurs and verification is possible in many cases with the extensions of non-smooth functions described above. Conditions under which convergence occurs are examined in this section.

Interval Newton Methods – General Convergence

Here, general formulas for convergence of interval Newton methods are developed, to later be interpreted for both the smooth and non-smooth cases. The

formulas are based on characterizing and bounding the width norm

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\big)\right\|$$

of the interval Newton image of a box $\boldsymbol{X}$.

If $\boldsymbol{X} \in \mathbb{IR}^n$, $F : \boldsymbol{X} \to \mathbb{R}^n$, and $\check{X} \in \boldsymbol{X}$, then the general interval Newton method will be of the form (1.45), that is,

$$\tilde{\boldsymbol{X}} = \boldsymbol{N}(F;\boldsymbol{X},\check{X}) = \check{X} + \boldsymbol{V}, \tag{1.45}$$

where

$$\Sigma\big(\boldsymbol{A}, -F(\check{X})\big) \subset \boldsymbol{V},$$

where $\boldsymbol{A}$ is either a Lipschitz set (an interval Jacobi matrix $\boldsymbol{F}'(\boldsymbol{X})$) or a slope enclosure $\mathbf{S}(F,\boldsymbol{X},\check{\boldsymbol{X}})$ for F over $\boldsymbol{X}$ (and centered at $\check{\boldsymbol{X}} \ni \check{X}$). Define

$$\Sigma_0 = \Sigma\big(\boldsymbol{A}, -F(\check{X})\big) \qquad \text{and} \qquad \boxbslash\Sigma_0 = \boxbslash\Sigma\big(\boldsymbol{A}, -F(\check{X})\big). \tag{6.6}$$

First note that the width norms obey

$$\|\mathrm{w}(\Sigma_0)\| = \|\mathrm{w}(\boxbslash\Sigma_0)\|, \tag{6.7}$$

since $\boxbslash\Sigma_0$ is the smallest interval vector containing Σ_0 and since $\|\cdot\| = \|\cdot\|_\infty$. Also write

$$\boldsymbol{V} = \boxbslash\Sigma_0 + \boldsymbol{E}, \quad \text{so} \quad \mathrm{w}\big(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\big) = \mathrm{w}(\boldsymbol{V}) = \mathrm{w}\big(\boxbslash\Sigma_0\big) + \mathrm{w}(\boldsymbol{E}). \tag{6.8}$$

The following assumption is non-restrictive.

Assumption 6.1 *There exists a K_{Σ_0}, depending only on the particular interval Newton method, such that*

$$\|\mathrm{w}(\boldsymbol{E})\| = \|\mathrm{w}(\boldsymbol{V}) - \mathrm{w}(\Sigma)\| \le K_{\Sigma_0}\|\mathrm{w}(\Sigma)\|^2.$$

For theory showing that particular interval Newton methods satisfy Assumption 6.1, see [175, §4.2, p. 124 and §4.3.5, p. 138].

Combining (6.8) and Assumption 6.1 gives

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\big)\right\| \le \left(1 + K_{\Sigma_0}\left\|\mathrm{w}\big(\boxbslash\Sigma_0\big)\right\|\right)\left\|\mathrm{w}\big(\boxbslash\Sigma_0\big)\right\|, \tag{6.9}$$

so $\|\mathrm{w}\big(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\big)\|$ can be bounded by bounding $\|\mathrm{w}(\Sigma_0)\|$. To bound $\|\mathrm{w}(\Sigma_0)\|$ a sensitivity bound for real linear systems will be used, namely

Theorem 6.1 (A modification of Theorem 2.7.2 in [61]) *Assume*

$$\begin{aligned} AX &= B, & A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^n \text{ and} \\ (A+\Delta A)(X+\Delta X) &= B, & \Delta A \in \mathbb{R}^{n\times n}, \end{aligned}$$

assume $\|\Delta A\| \leq \delta \|A\|$, *and assume* $\delta\kappa(A) = r < 1$, *where* $\kappa(A)$ *is the condition number* $\|A\| \cdot \|A^{-1}\|$. *Then*

$$\begin{aligned} \|\Delta X\| &\leq r\frac{1}{1-r}\|X\| \\ &\leq r\frac{1}{1-r}\|A^{-1}\| \cdot \|B\|. \end{aligned}$$

To use Theorem 6.1, identify:

$$B = -F(\check{X}) \text{ and } A = \check{A} = \mathrm{m}(A), \text{ so } \Delta A = \frac{\mathrm{w}(\boldsymbol{A})}{2},$$

so that

$$\delta = \frac{1}{2}\frac{\|\mathrm{w}(\boldsymbol{A})\|}{\|\check{A}\|} \text{ and } r = \frac{1}{2}\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\check{A}^{-1}\|. \tag{6.10}$$

Assume $X_1 \in \Sigma_0$ and $X_2 \in \Sigma_0$. Then (6.10) and Theorem 6.1 give

$$\|X_1 - X_2\| \leq \|\mathrm{w}(\boldsymbol{A})\| \cdot \| - F(\check{X})\| \left\{ \frac{\|\check{A}^{-1}\|}{2} \cdot \frac{1}{1-r} \right\}. \tag{6.11}$$

Combining (6.11) and (6.9) gives

Lemma 6.2 *Make Assumption 6.1. Also assume*

$$\left(1 + K_{\Sigma_0} \left\|\mathrm{w}\left([\![\Sigma_0\right)\right\|\right) \leq \tilde{K} \tag{6.12}$$

for all possible Σ_0 *under consideration. Then*

$$\left\|\mathrm{w}\left(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\right)\right\| \leq C\|\mathrm{w}(\boldsymbol{A})\| \cdot \|F(\check{X})\|,$$

where

$$C = \tilde{K}\left\{ \frac{1}{2} \cdot \frac{\|\check{A}^{-1}\|}{1 - \frac{1}{2}\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\check{A}^{-1}\|} \right\}. \tag{6.13}$$

The next assumption is necessary for proof that verification is possible.

Assumption 6.2 *Assume there exists an* $X^* \in \boldsymbol{X}$ *with* $F(X^*) = 0$. *Also assume that* $\check{X} = \mathrm{m}(\boldsymbol{X})$.

Now, since $\boldsymbol{A}$ is either a Lipschitz matrix or a slope enclosure centered at $\check{X}$,

$$F(X^*) = 0 = F(\check{X}) + A(X^* - \check{X}) \quad \text{for some } A \in \boldsymbol{A}.$$

Thus,

$$\|F(\check{X})\| \leq \|\boldsymbol{A}\| \frac{\|\mathrm{w}(\boldsymbol{X})\|}{2}. \tag{6.14}$$

Combining (6.14) and Lemma 6.2 thus gives

Theorem 6.3 *Make assumptions 6.1 and 6.2, and let* C *be as in (6.13) of Lemma 6.2. Then*

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F; \boldsymbol{X}, \check{X})\big)\right\| \leq C\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\boldsymbol{A}\| \frac{\|\mathrm{w}(\boldsymbol{X})\|}{2}.$$

Corollary 6.4 *Make assumptions 6.1 and 6.2, and let* C *be as in (6.13) of Lemma 6.2. Then*

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F; \boldsymbol{X}, \check{X})\big)\right\| \leq \|\mathrm{w}(\boldsymbol{X})\|$$

provided

$$C\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\boldsymbol{A}\| < 2 \quad \text{(for convergence)}. \tag{6.15}$$

Furthermore, if we assume $\check{X} \in \boldsymbol{N}(F; \boldsymbol{X}, \check{X})$, *such as when* $F(\check{X}) \approx 0$ *and* ϵ*-inflation is used, then* $\boldsymbol{N}(F; \boldsymbol{X}, \check{X}) \subset \boldsymbol{X}$, *provided*

$$C\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\boldsymbol{A}\| \frac{\|\mathrm{w}(\boldsymbol{X})\|}{\min_{1 \leq i \leq n} \mathrm{w}(\boldsymbol{x}_i)} < 1 \quad \text{(for inclusion)}. \tag{6.16}$$

For smooth problems and Lipschitz or slope matrices formed with the usual computations,

$$\|\mathrm{w}(\boldsymbol{A})\| \leq K_{F'} \|\mathrm{w}(\boldsymbol{X})\|, \tag{6.17}$$

so Condition (6.15) (and the more restrictive Condition 6.16 for the inclusion $\boldsymbol{N}(F; \boldsymbol{X}, \check{X}) \subset \boldsymbol{X}$) hold for sufficiently small $\boldsymbol{X}$ centered about an approximate solution $\check{X} \approx X^*$. Thus, neglecting roundout error, it should always be possible to verify existence or uniqueness with ϵ-inflation. In fact, the following is a direct consequence of Theorem 6.3.

Theorem 6.5 *Make assumptions 6.1 and 6.2, and let C be as in Lemma 6.2. Also assume $\|\mathrm{w}(\boldsymbol{A})\| \le K_{F'}\|\mathrm{w}(\boldsymbol{X})\|$. Then*

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\big)\right\| \le \tilde{C}\|\mathrm{w}(\boldsymbol{X})\|^2,$$

where $\tilde{C} = CK_{F'}\|\boldsymbol{A}\|/2$.

Convergence for Non-Smooth Problems

For non-smooth problems, $\mathrm{w}(\boldsymbol{A}) \not\to 0$ as $\mathrm{w}(\boldsymbol{X}) \to 0$ (i.e. Condition (6.17) does not hold), Condition 6.15 is not in general satisfied, and the interval Newton method does not converge. However, when the non-smoothness does not occur in dominant terms, Condition 6.15 is often still satisfied for small $\mathrm{w}(\boldsymbol{X})$, and the interval Newton method converges linearly:

Theorem 6.6 *Make assumptions 6.1 and 6.2, and let C be as in Lemma 6.2. Also assume that (for small enough $\mathrm{w}(\boldsymbol{X})$)*

$$\|\mathrm{w}(\boldsymbol{A})\| \le \frac{2}{CM_F'}.$$

Then the width of $\boldsymbol{X}$ under iteration of

$$\boldsymbol{X} \leftarrow \boldsymbol{X} \cap \boldsymbol{N}(F;\boldsymbol{X},\check{X})$$

tends to zero linearly, with convergence factor $c = C\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\boldsymbol{A}\|/2$. That is,

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F;\boldsymbol{X},\check{X})\big)\right\| \le c\|\mathrm{w}(\boldsymbol{X})\|.$$

In some cases, Theorem 6.6 does not give reasonably sharp bounds, since the convergence factor c incorporates a worst-case bound $\|X^* - \check{X}\| \le \frac{\mathrm{w}(\boldsymbol{X})}{2}$. Actually, in ϵ-inflation algorithms, $\boldsymbol{X}$ can be *constructed* so

$$\left\|X^* - \check{X}\right\| \le \nu\|\mathrm{w}(\boldsymbol{X})\|^2 \tag{6.18}$$

For example, as explained in §4.2, X^* can be computed with an approximate solver with a stopping tolerance that is proportional to the square of the width of the tolerance of the box constructed around it. In such cases, (6.14) can be replaced by

$$\|F(\check{X})\| \le \|\boldsymbol{A}\|\nu\|\mathrm{w}(\boldsymbol{X})\|^2, \tag{6.19}$$

and Theorem 6.6 may be replaced by

Theorem 6.7 *Make assumptions 6.1 and 6.2, and let C be as in Lemma 6.2. Also assume (6.18) for some $\nu > 0$. Then, for small enough initial $\mathrm{w}(\boldsymbol{X})$, the width of $\boldsymbol{X}$ under iteration of*

$$\boldsymbol{X} \leftarrow \boldsymbol{X} \cap \boldsymbol{N}(F; \boldsymbol{X}, \check{X})$$

tends to zero quadratically. In particular,

$$\left\|\mathrm{w}\big(\boldsymbol{N}(F; \boldsymbol{X}, \check{X})\big)\right\| \leq C\|\mathrm{w}(\boldsymbol{A})\| \cdot \|\boldsymbol{A}\|\nu\|\mathrm{w}(\boldsymbol{X})\|^2,$$

where C is as in (6.13).

Theorem 6.7 follows directly from Lemma 6.2.

Remark 6.1 *Theorem 6.7 implies that, even for non-smooth functions, existence and uniqueness can* always *be verified with ϵ-inflation, provided a sufficiently accurate approximate solution can be obtained.*

Example 6.2 *Consider*

$$f(x) = |x^2 - x| - 2x + 2 = 0. \tag{6.20}$$

This function has both a root and a cusp at $x = 1$, with a left derivative of -3 and a right derivative of -1 at $x = 1$. If $1 \in \boldsymbol{x}$, then a slope enclosure is given by $\mathbf{S}(f, \boldsymbol{x}, x) = [-1, 1](\boldsymbol{x} + x - 1) - 2$.

Consider using the interval Newton method

$$\begin{aligned} \tilde{\boldsymbol{x}} &\leftarrow \check{x}^{(k)} - f(\check{x}^{(k)})/\mathbf{S}(f, \boldsymbol{x}^{(k)}, \check{x}^{(k)}) \\ \boldsymbol{x}^{(k+1)} &\leftarrow \boldsymbol{x}^{(k)} \cap \tilde{\boldsymbol{x}}, \end{aligned}$$

with $\check{x}^{(k)}$ equal to the midpoint of $\boldsymbol{x}^{(k)}$, and $\boldsymbol{x}^{(0)} = [0.7, 1.1]$. The initial slope enclosure is then $\mathbf{S}(f, [0.7, 1.1], 0.9) = [-3, -1]$, $\tilde{\boldsymbol{x}} = .9 - .29/[-3, -1] = [.99\overline{6}, 1.19]$, and $\boldsymbol{x}^{(1)} = [0.99\overline{6}, 1.1]$. Subsequent iterates are given[1] in Table 6.1. Note that on iteration no. 3, existence of a root within $\boldsymbol{x}^{(3)}$ was proven, since $\boldsymbol{x}^{(3)} \subset \mathrm{int}(\boldsymbol{x})^{(2)}$. The last column of Table 6.1 gives ratios of successive widths $\mathrm{w}\big(\boldsymbol{x}^{(k+1)}\big)/\mathrm{w}\big(\boldsymbol{x}^{(k)}\big)$. Thus, the observed convergence is linear, with a convergence factor of roughly 1/3.

[1]The intervals are rounded outward to 17 digits, while the other numbers are floating point approximations.

k	$x^{(k)}$	$\check{x}^{(k)}$	$\mathrm{w}\left(x^{(k)}\right)$	$\frac{w_k}{w_{k-1}}$
0	[0.69999999999999995, 1.1000000000000001]	0.90000000000000	4.0E-1	—
1	[0.99666666666666603, 1.1000000000000001]	1.04833333333333	1.0E-1	0.25
2	[0.99666666666666603, 1.0337233103935069]	1.01519498853009	4.0E-2	0.37
3	[0.99946121840631785, 1.0102869853018799]	1.00487410185410	1.1E-2	0.28
4	[0.99994908830928475, 1.0032654498302074]	1.00160726906975	3.3E-3	0.30
5	[0.99999472584400717, 1.0010732412070054]	1.00053398352551	1.1E-3	0.33
6	[0.99999942598430213, 1.0003561793202454]	1.00017780265227	3.6E-4	0.33
7	[0.99999993663653530, 1.0001185561849595]	1.00005924641075	1.2E-4	0.33
8	[0.99999999297472042, 1.0000394999475355]	1.00001974646113	4.0E-5	0.33
9	[0.99999999921996751, 1.0000131645673783]	1.00000658189367	1.3E-5	0.33
10	[0.99999999991334831, 1.0000043879579972]	1.00000219393567	4.4E-6	0.33

Table 6.1 Iterates of the interval Newton method for $f(x) = |x^2 - x| - 2x + 2$

Now we will use Theorem 6.6 to analyze the convergence for Example 6.2. The limiting value[2] of the slope bound set as $\boldsymbol{x} \to [1,1]$ is $\boldsymbol{A} = [-3,-1]$, and the limiting value of its center is $\check{A} = -2$. From this, we compute limiting values for C as defined in (6.13), (6.12) and Assumption 6.1: Formula 6.12 implies that, for an $\epsilon > 0$ of our choosing and $\mathrm{w}(\mathbb{I}\Sigma_0)$ sufficiently small (i.e. $\mathrm{w}(\boldsymbol{X})$ sufficiently small), $\tilde{K}$ can be taken to be $(1+\epsilon)$. Furthermore, $\mathrm{w}(\boldsymbol{A}) \approx -1 - (-3) = 2$, and $\|\check{A}^{-1}\| \approx |1/(-2)| = \frac{1}{2}$, so

$$C \approx (1+\epsilon) \cdot \frac{1}{2} \cdot \frac{\frac{1}{2}}{1 - \frac{1}{2}(2)\frac{1}{2}} \approx \frac{1}{2}.$$

Also, $\|\boldsymbol{A}\| \approx \max\{|-3|, |-1|\} = 3$, and

$$\mathrm{c} \approx \frac{1}{2} \cdot (2) \cdot \frac{3}{2} = \frac{3}{2}$$

in Theorem 6.6. Thus, Theorem 6.6 is not sharp enough in this case to predict inclusion or convergence. Also, we observe that $|\check{x} - x^*|$ is tending to zero at about the same rate as $\mathrm{w}(\boldsymbol{x})$, so Theorem 6.7 would not give an appropriate conclusion either. However, after iteration no. 2, we observe $|f(x^{(2)})| \leq 0.015$ and $\mathrm{w}\left(x^{(2)}\right) \approx 0.04$. Plugging these values directly into Lemma 6.2 gives

$$\mathrm{w}\left(\boldsymbol{x}^{(3)}\right) \leq \frac{1}{2} \cdot 2 \cdot 0.015 = 0.015 < 0.04/2 = .02,$$

thus proving that verification must occur on this step. (In fact, the new width $\mathrm{w}\left(\boldsymbol{x}^{(3)}\right) \approx 0.01$. The left endpoints of $\boldsymbol{x}^{(k)}$ are converging faster than the right endpoints.)

[2]although, in general, successive slope bounds are not contained in previous ones, since the base point $\check{x}^{(k)}$ is changing

7

USE OF INTERMEDIATE QUANTITIES IN THE EXPRESSION VALUES

7.1 THE BASIC APPROACH

Code lists as described in §1.4.4 and §2.2.2 contain complete information about the relationships among the intermediate quantities computed during evaluation of a particular natural interval extension. These relationships can sometimes be used to reduce the amount of overestimation in such interval extensions, or to help in the solution of nonlinear systems of equations.

Example 7.1 (from [112]) *Define* $F(X) : \mathbb{R}^2 \to \mathbb{R}^2$ *by*

$$\begin{array}{rcl} f_1(x_1, x_2) & = & x_1^3 + x_1^2 x_2 + x_2^2 + 1 \\ f_2(x_1, x_2) & = & x_1^3 - 3x_1^2 x_2 + x_2^2 + 1, \end{array} \tag{7.1}$$

with initial box $\boldsymbol{X} = ([-2, 0], [-1, 1])^T$ *and initial guess point* $\check{X} = (-1, 0)^T$.

The system (7.1) has a unique solution within $\boldsymbol{X}$, at the initial guess point $(-1, 0)^T$. However, the system exhibits a substantial amount of dependency (i.e. interrelationships) among the terms, so there is overestimation in interval extensions for the function, slope, and derivative matrices. For instance, a Hansen slope matrix (computed by the `INTOPT_90` routine `DENSE_SLOPE_MATRIX` as on page 96) is

$$\mathbf{S}(F, \boldsymbol{X}, \check{X}) = \begin{pmatrix} [0, 7] & [-1, 5] \\ [0, 7] & [-13, 1] \end{pmatrix}.$$

Therefore, since $\mathbf{S}(F, \boldsymbol{X}, \check{X})$ contains the zero matrix, $Y\mathbf{S}(F, \boldsymbol{X}, \check{X})$ will contain the zero matrix for any preconditioner Y. Thus, since $\|F(\check{X})\|$ is small, the interval Gauss–Seidel method is ineffective.

However, a particular code list decomposition (not necessarily that produced by the package INTLIB_90) corresponds to the expanded system:

	v_1	$=$	x_1,		v_2	$=$	x_2,
(i)	v_3	$=$	v_1^2,	(ii)	v_4	$=$	v_2^2,
(iii)	v_5	$=$	$v_3 v_2$,	(iv)	v_6	$=$	v_1^3,
(v)	v_7	$=$	$v_6 + v_4 + 1$,				
(vi)	$v_7 + v_5$	$=$	0,				
(vii)	$v_7 - 3v_5$	$=$	0.				

(7.2)

The information in the code list, or, equivalently, in systems such as (7.2), could possibly be manipulated symbolically to reduce overestimation in the dependent variables $(v_7 + v_5)$ and $(v_7 - 3v_5)$, but that is outside the scope of this work. Two things can be done numerically when the code list or a system such as (7.2) is available.

First, since each equation in (7.2) contains only a single, invertible operation, each intermediate variable can be solved for each other intermediate variable. This process, called substitution-iteration in our own work, can be continued until all values of the intermediate variables become stationary. For example, in (7.2), the last two equations, corresponding to the original system (7.1), can be used to start the process, once the intermediate quantities are evaluated with forward substitution. In (7.2), an initial forward substitution gives:

$$\boldsymbol{V} = ([-2,0],[-1,1],[0,4],[0,1],[-4,4],[-8,0],[-7,2])^T.$$

If we solve for $\boldsymbol{v}_5$ in the sixth equation, we obtain

$$\tilde{\boldsymbol{v}}_5 = -\boldsymbol{v}_7 = [-2,7],$$

and we may replace $\boldsymbol{v}_5$ by $\boldsymbol{v}_5 \cap \tilde{\boldsymbol{v}}_5 = [-2,4]$. If we then solve for $\boldsymbol{v}_5$ in the seventh equation, we obtain

$$\tilde{\boldsymbol{v}}_5 = \boldsymbol{v}_7/3 = [-\frac{7}{3}, \frac{2}{3}],$$

and we may replace $\boldsymbol{v}_5$ by $\boldsymbol{v}_5 \cap \tilde{\boldsymbol{v}}_5 = [-2, 2/3]$. Finally, we may solve for $\boldsymbol{v}_7$ in the sixth equation, to obtain

$$\tilde{\boldsymbol{v}}_7 = -\boldsymbol{v}_5 = [-\frac{2}{3}, 2],$$

and we may replace $\boldsymbol{v}_7$ by $\boldsymbol{v}_7 \cap \tilde{\boldsymbol{v}}_7 = [-2/3, 2]$. Now, since both v_7 and v_5 have been changed, it may be possible to tighten v_6, v_4, v_3, or v_2, using the fifth or seventh equations in (7.2). For example, $\tilde{\boldsymbol{v}}_2 \leftarrow \boldsymbol{v}_5/\boldsymbol{v}_3$, using the new value

of v_5, may result in $\tilde{v}_2$ different from v_2 (or even disjoint from v_2, in which case the computation would have proven that there are no roots of F in X). However, these substitutions do not result in sharper v_6, v_4, v_3, or v_2, and the substitution-iteration process terminates.

The second numerical process is an interval Newton method, applied not to the original system, but to the expanded system such as (7.2). Combined with optimal preconditioners, such an interval Newton method is more likely to result in sharper bounds on the independent variables than an interval Newton method applied to the original system, such as (7.1). For (7.2), the actual system is

$$F_E(V) = \begin{pmatrix} v_3 - v_1^2 \\ v_4 - v_2^2 \\ v_5 - v_3 v_2 \\ v_6 - v_1^3 \\ v_7 - (v_6 + v_4 + 1) \\ v_7 + v_5 \\ v_7 - 3v_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{7.3}$$

A natural extension of the interval Jacobi matrix will be used, with the interval Gauss–Seidel method preconditioned with C^{W}-preconditioners, and starting with the new vector

$$\boldsymbol{V} = ([-2,0],[-1,1],[0,4],[0,1],[-4,4],[-2,4],[-2/3,2])^T.$$

Performing a preconditioned Gauss–Seidel step for variable 5 gives $v_5 \leftarrow \tilde{v}_5 \subseteq [-1.9 \times 10^{-15}, 1.8 \times 10^{-15}]$, then performing a similar step for variable 6 gives $v_6 \leftarrow \tilde{v}_6 \subseteq [-2,-1]$, then performing a similar step for variable 7 gives $v_6 \leftarrow \tilde{v}_6 \subseteq [-1.3 \times 10^{-15}, 1.4 \times 10^{-15}]$.

The interval Gauss–Seidel and substitution-iteration processes can be mixed to achieve higher efficiency. For example, after v_6 has been recomputed with the interval Gauss–Seidel step, the fourth equation can be used to solve for v_1:

$$v_1 \leftarrow \tilde{v}_1 = \sqrt[3]{v_6} \subseteq [-1.26,-1],$$

then the first equation can be similarly used to solve for v_3:

$$v_3 \leftarrow \tilde{v}_3 = v_1^2 \subseteq [1,1.588],$$

then the third equation can be used to solve for v_2:

$$v_2 \leftarrow \tilde{v}_2 = v_5/v_3 \subseteq [-1.9 \times 10^{-15}, 1.7 \times 10^{-15}].$$

Five more preconditioned Gauss–Seidel steps, mixed with six more steps of the substitution-iteration process, give $v_1 = [1, 1]$ to machine precision. Thus, an enclosure for the actual root $(x_1, x_2)^T = (v - 1, v_2)^T = (1, 0)^T$ has been computed to high precision. See [112] for additional details.

7.2 AN ALTERNATE VIEWPOINT – CONSTRAINT PROPAGATION

A number of researchers have, in a sense, independently discovered the substitution-iteration process proceeding from logic programming [231], e.g. [30, 88, 89, 90, 154, 232]. In fact, logic programming languages such a Prolog, when extended to handle interval data, are designed for this purpose, namely *constraint propagation.* However, programs written in such languages can execute hundreds or thousands of times slower than special-purpose programs for the same process in Fortran 90 or C++ [30]. Nonetheless, a sophisticated theory and successful general algorithm, involving substitution-iteration, interval Newton, and tessellation steps, appears in [232], along with substantial numerical experimentation.

Another research group has produced a special software system UniCalc [12] that solves nonlinear problems (nonlinear equations, optimization, etc.), based primarily on the substitution-iteration process (called *subdefinite calculations* by the group), along with a user-friendly interactive interface.

7.3 APPLICATION TO GLOBAL OPTIMIZATION

Additional opportunities are available when applying the process to global optimization. In particular, relationships among quantities common to both the objective function and gradient appear in a gradient code list. If an upper bound $\overline{\phi}$ for the global optimum is known, then this value can possibly be used to compute sharper inclusions to the gradient, and conversely. Similar computations, implemented in BNR Prolog, were reported in [30].

Example 7.2 *Let* $\phi(X) = x_1^2 - x_1^2 x_2^2 + x_2^2$, *to be optimized over* $\boldsymbol{X} = ([-1, 1], [-1, 1])^T$.

The global optimum of ϕ over $\boldsymbol{X}$ is 0, and the unique optimizer is $(0,0)^T$. If ϕ is programmed as

```
T(1) = X(1)**2
T(2) = X(2)**2
PHI(1) = T(1) - T(1)*T(2) + T(2)
```

then INTLIB_90 produces[1] the following code list.

	v_1	=	x_1		v_2	=	x_2				
(i)	v_3	=	v_1^2	(ii)	v_4	=	v_2^2	(iii)	v_5	=	$v_4 v_3$
(iv)	v_6	=	$v_3 - v_5$	(v)	v_7	=	$v_6 + v_9$	(vi)	v_8	=	$2v_1$
(vii)	v_9	=	$v_8 v_4$	(viii)	v_{10}	=	$v_8 - v_9$	(ix)	v_{11}	=	$2v_2$
(x)	v_{12}	=	$v_3 v_{11}$	(xi)	v_{13}	=	$-v_{12}$	(xii)	v_{14}	=	$v_{13} + v_{11}$
	ϕ	=	v_7		$\frac{\partial \phi}{\partial x_1}$	=	v_{10}		$\frac{\partial \phi}{\partial x_2}$	=	v_{14}

Suppose we know a sharp upper bound $\overline{\phi} = 0$ on the global optimum, and suppose the sub-box $\boldsymbol{X} = ([.5,1],[-1,-.5])$ is to be processed. An initial evaluation of the code list (i.e. a *forward substitution*) gives:

v_1	=	$[.5,1]$	v_2	=	$[-1,-.5]$			
v_3	=	$[.25,1]$	v_4	=	$[.25,1]$	v_5	=	$[.0625,1]$
v_6	=	$[-.75,.9375]$	v_7	=	$[-.5,1.9375]$	v_8	=	$[1,2]$
v_9	=	$[.25,2]$	v_{10}	=	$[-1,1.75]$	v_{11}	=	$[-2,-1]$
v_{12}	=	$[-2,-.25]$	v_{13}	=	$[.25,2]$	v_{14}	=	$[-1.75,1]$
ϕ	=	$[-.5,1.9375]$	$\frac{\partial \phi}{\partial x_1}$	=	$[-1,1.75]$	$\frac{\partial \phi}{\partial x_2}$	=	$[-1.75,1]$

Thus, since $0 \in \phi(\boldsymbol{X})$ and $0 \in \nabla\phi(\boldsymbol{X})$, the box cannot be rejected just from the computed values. However, we may set $v_7 = 0$, and solve for v_9 in (v):

$$\tilde{v}_9 = v_7 - v_6 = [-.9375,.75]; \quad v_9 \leftarrow v_9 \cap \tilde{v}_9 = [.25,.75].$$

We may now solve for v_{10} in (viii), obtaining

$$v_{10} \leftarrow \tilde{v}_{10} = [1,2] - [.25,.75] = [.25,1.75].$$

Since $0 \notin v_{10}$ (with $v_{10} = \partial\phi/\partial x_2$), $\boldsymbol{X}$ cannot contain both the global optimum and a critical point of ϕ. Thus, the box $([.5,1],[-1,-.5])^T$ may be rejected[2].

[1] with NAG Fortran 90 compiler, version 2.1. The actual code list may vary with the compiler.

[2] if the boundary elements of X corresponding to parts of the boundary of the original box in Example 7.2 are considered separately

A similar technique can be used with the equality constraints, to provide narrower intervals for the bound constraints or to reject sub-boxes. Our package INTOPT_90 provides this facility within the module COMPONENT_SOLVE.

7.4 EFFICIENCY AND PRACTICALITY

In a sense, solving for one variable in one equation represents a single operation (one of the four arithmetic operations or evaluation of a standard function). However, easy implementations involve significant overhead, and it is not always advantageous to carry the substitution-iteration process until no variables are changed. In general, a mix of substitution-iteration, interval Newton iteration (such as interval Gauss–Seidel steps), and generalized bisection, with good heuristics to choose when to do each, should be most appropriate.

The work [128] contains a heuristic for deciding for which coordinates to do an interval Gauss–Seidel step for the expanded system. The assumption was that the substitution-iteration process is inexpensive relative to a Gauss–Seidel step.

In [30], upper bounds on the objective function were used, within a modified Moore–Skelboe algorithm (page 173) implemented in BNR Prolog, to reject boxes by contradiction. (That is, $\tilde{\boldsymbol{v}}_i$ were computed through the substitution-iteration process, i.e. through constraint propagation, such that $\tilde{\boldsymbol{v}}_i \cap \boldsymbol{v}_i = \emptyset$.) The constraint propagation process was shown to greatly increase the efficiency of the Moore–Skelboe algorithm.

The substitution-iteration process was discovered independently by Dallwig, Neumaier, and Schichl [47]. There, experiments that hint at the practicality of the method are presented.

However, our own research, i.e. in [128, 222] and unpublished, is less conclusive. Differences may be due to differences in the programming environments. For example, certain operations are much faster in Fortran than in Prolog, while others are not. Also, algorithmic details, such as where second order interval extensions are used, and how (or if) interval Newton methods are used, affect the conclusions. Along these lines, the work [232] shows much promise.

The substitution-iteration process could possibly be more effective if, not the entire code list, but only selected relationships, were used to solve for one

variable in terms of the others. Not only can those operations in the code list (i.e. the operations in Tables 2.6) be used, but any user-defined relation is possible. Relations relating more complicated subexpressions that are shared among various components of the gradient (or among various equations in a nonlinear system) should be particularly effective, while redundant relations can be omitted. Although, at the time of this book, we have set up some machinery for experimentation with this (see below), we have not yet had extensive experience with it.

Another improvement in efficiency is to use an efficient sparse linear programming problem solver to produce preconditioners for the expanded system. In [222] and elsewhere, empirical results reveal that Gauss–Seidel steps on the expanded system result in less boxes considered than working with the original system of equations, when there is much interval dependency in the original system. Despite this, the total execution time could still be larger with substitution-iteration. However, these experiments were not done with linear programming problem solvers that made optimal use of the sparsity and structure associated with LP-preconditioners.

7.5 PROVIDED SOFTWARE

The following building blocks are available in INTOPT_90 for the substitution-iteration process.

`SUBROUTINE FORWARD_SUBSTITUTION(X,XX)` returns a set of interval values of the intermediate quantities in the expressions to evaluate the function (the variables v_1 through v_{14} of Example 7.2) in `XX`, given the intervals corresponding to the independent variables (i.e. x_1 and x_2 in Example 7.2) in `X`. The number of intermediate variables in the code list, i.e. the dimension of `XX`, is given in the variable `NVAR` in the module `CODELIST_VARIABLES`.

`FUNCTION COMPONENT_SOLVE` solves for the `L`-th variable in the `J`-th equation. See the source code for details.

`SUBROUTINE SUBSIT(X, CHANGED_COORDINATES, FLAG)` performs the substitution-iteration process, starting with those equations containing the variables indicated in the logical array `CHANGED_COORDINATES`.

`SUBROUTINE SUBSIT_DENSE_OPT` performs the substitution-iteration process on a gradient code list, checking whether a contradiction occurs based on the current best approximation for an optimum.

Additionally, there is support for user-defined operations, including the modules `SUBSIT_OPERATIONS`, `SUBSIT_PARAMETERS`, and `USER_SUPPORT`, as well as

the routine USER_SUBSIT and the user-supplied routine USER_COMPONENT_SOLVE. The a standard sparse matrix indexing scheme is used to describe the user-supplied equations and the variables therein.

As of the writing of this book, the user-defined substitution-iteration routines have not been thoroughly packaged and documented. The author should be contacted at rbk@usl.edu if details concerning these routines are desired.

7.6 EXERCISES

1. Compute the interval Jacobi matrix of the system (7.1), evaluated at $\boldsymbol{X} = ([-2, 0], [-1, 1])^T$.

2. Use the INTOPT_90 routine DENSE_GAUSS_SEIDEL_STEP to check the preconditioned interval Newton method computations starting on page 229.

3. Use INTLIB_90 to generate a code list corresponding to the system (7.1). Is the code list the same as that represented by (7.2)? Would the difference impact the substitution-iteration process?

4. Use the INTOPT_90 routines GAUSS_SEIDEL_STEP and SUBSIT to perform a computation similar to that on page 229, but for the code list generated in Problem 3.

5. Repeat the computations in the preceding exercise, but using a Hansen slope matrix, rather than an interval Jacobi matrix.

REFERENCES

[1] O. Aberth. *Precise Numerical Analysis.* Wm. C. Brown, Dubuque, Iowa, 1988.

[2] O. Aberth. Computation of topological degree using interval arithmetic, and applications. *Math. Comp.*, 62(205):171–178, January 1994.

[3] O. Aberth and M. Schaefer. Precise computation using range arithmetic, via C++. *ACM Trans. Math. Software*, 18(4):481–491, December 1992.

[4] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook - Complete ANSI/ISO Reference.* McGraw-Hill, New York, 1992.

[5] Y. Akyildiz and M. I. Suwaiyel. No pathologies for interval Newton's method. *Interval Computations*, 1993(1):60–72, 1993.

[6] G. Alefeld. Bounding the slope of polynomial operators and some applications. *Computing*, 26:227–237, 1980.

[7] G. Alefeld. Inclusion methods for systems of nonlinear equations – the interval Newton method and modifications. In J. Herzberger, editor, *Topics in Validated Computations*, pages 7–26, Amsterdam, 1994. Elsevier Science Publishers.

[8] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[9] P. Alexandroff and H. Hopf. *Topologie.* Chelsea, 1935.

[10] E. Allgower and K. Georg. *Numerical Continuation Methods: An Introduction.* Springer-Verlag, New York, 1990.

[11] N. Apostolatos, U. Kulisch, Krawczyk R., B. Lortz, K. Nickel, and H.-W. Wippermann. The algorithmic language triplex-ALGOL-60. *Numer. Math.*, 11:175–180, 1968.

[39] G. F. Corliss. Guaranteed error bounds for ordinary differential equations. In W. A. Light and M. Marletta, editors, *Theory of Numerics in Ordinary and Partial Differential Equations*, Advances in Numerical Analysis, vol. IV, pages 1–75, London, 1995. Oxford University Press.

[40] G. F. Corliss and L. B. Rall. Computing the range of derivatives. In E. Kaucher, S. M. Markov, and G. Mayer, editors, *Computer Arithmetic, Scientific Computation, and Mathematical Modelling*, volume 12 of *IMACS Annals on Computing and Applied Math.*, pages 195–212, Basel, 1991. J. C. Baltzer AG.

[41] H. Cornelius and R. Lohner. Computing the range of values of real functions with accuracy higher than second order. *Computing*, 33(3):331–347, 1984.

[42] F. Crary. The AUGMENT precompiler. Technical Report 1470, MRC, University of Wisconsin, Madison, 1976.

[43] J. Cronin. *Fixed Points and Topological Degree in Nonlinear Analysis.* American Mathematical Society, Providence, RI, 1964.

[44] A. E. Csallner and T. Csendes. Convergence speed of interval methods for global optimization and the joint effects of algorithmic modifications, 1995. Talk given at SCAN'95, Wuppertal, Germany, Sept. 26 – 29, 1995.

[45] T. Csendes. Nonlinear parameter estimation by global optimization – efficiency and reliability. *Acta Cybernetica*, 8(4):361–370, 1988.

[46] T. Csendes and D. Ratz. Subdivision direction selection in interval methods for global optimization, 1994. Accepted for publication in *SIAM J. Numer. Anal.*

[47] S. Dallwig, A. Neumaier, and H. Schichl. GLOPT – a program for constrained global optimization, 1996. Preprint, Institut für Mathematik, Universität Wien, Strudlhofgasse 4, A-1090 Wien, Austria.

[48] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Least Squares.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[49] L. C. W. Dixon and G. P. Szegö. The global optimization problem: An introduction. In L. C. W. Dixon and G. P. Szegö, editors, *Towards Global Optimization 2*, pages 1–15, Amsterdam, Netherlands, 1978. North-Holland.

[50] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Comm. ACM*, 30(5):403–407, May 1987.

[51] K. Du. *Cluster Problem in Global Optimization using Interval Arithmetic.* PhD thesis, University of Southwestern Louisiana, 1994.

[52] K. Du and R. B. Kearfott. The cluster problem in global optimization: The univariate case. *Computing (Suppl.)*, 9:117–127, 1992.

[53] J. S. Ely. The VPI software package for variable precision interval arithmetic. *Interval Computations*, 1993(2):135–153, 1993.

[54] W. Enger. Interval ray tracing – a divide and conquer strategy for realistic computer graphics. *The Visual Computer*, 9:91–104, 1992.

[55] J. Eriksson. *Parallel Global Optimization using Interval Analysis.* PhD thesis, University of Umeå, Institute of Information Processing, 1991.

[56] J Eriksson and P. Lindstrœm. A parallel interval method implementation for global optimization using dynamic load balancing. *Reliable Computing*, 1(1):77–92, 1995.

[57] C. Falco-Korn, S. Gutzwiller, S. Künig, and Ch. Ullrich. Modula-SC, motivation, language definition and implementation. In E. Kaucher, S. Markov, and G. Mayer, editors, *Computer Arithmetic – Scientific Computation and Mathematical Modelling*, IMACS Annals on Computing and Applied Math. 12, pages 161–180, Basel, 1992. J. C. Baltzer AG.

[58] C. A. Floudas and P. M. Pardalos. *A Collection of Test Problems for Constrained Global Optimization Algorithms.* Lecture Notes in Computer Science no. 455. Springer-Verlag, New York, 1990.

[59] J. Gargantini and P. Henrici. Circular arithmetic and the determination of polynomial zeros. *Numer. Math.*, 18:305–320, 1972.

[60] P. E. Gill, W. Murray, and M. Wright. *Practical Optimization.* Academic Press, New York, 1981.

[61] G. H. Golub and C. F. Van Loan. *Matrix Computations (Second Edition).* Johns Hopkins University Press, Baltimore, MD, 1989.

[62] R. T. Gregory and D. L Karney. *A Collection of Matrices for Testing Computational Algorithms.* Wiley, New York, 1969.

[63] A. Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24(3):20–21, May 1991.

[64] A. Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24(4):8–24, July 1991.

[65] A. Griewank and G. F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Philadelphia, 1991. SIAM.

[66] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markowitz rule. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135, Philadelphia, 1991. SIAM.

[67] A. Griewank and Ph. L. Toint. On the unconstrained optimization of partially separable functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312, New York, 1982. Academic Press.

[68] E. Grosse and J. J. Dongarra. Distribution of mathematical software via electronic mail. *SIGNUM Newsletter*, 20(3):45–47, July 1985.

[69] G. D. Hager. Solving large systems of nonlinear constraints with application to data modeling. *Interval Computations*, 1993(2):169–200, 1993.

[70] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *Numerical Toolbox for Verified Computing I.* Springer-Verlag, New York, 1993.

[71] R. Hammer, M. Neaga, and D. Ratz. PASCAL-XSC, New concepts for scientific computation and numerical data processing. In E. Adams and U. Kulisch, editors, *Scientific Computing with Automatic Result Verification*, pages 15–44, New York, etc., 1993. Academic Press.

[72] E. R. Hansen. Interval arithmetic in matrix computations, part 1. *SIAM J. Numer. Anal.*, 2:308–320, 1965.

[73] E. R. Hansen. Interval forms of Newton's method. *Computing*, 20:153–163, 1978.

[74] E. R. Hansen. Global optimization using interval analysis: The one-dimensional case. *J. Optim. Theory Appl.*, 29(3):331–44, 1979.

[75] E. R. Hansen. Global optimization using interval analysis: The multidimensional case. *Numer. Math.*, 34(3):247–270, 1980.

[76] E. R. Hansen. Bounding the solution of interval linear equations. *SIAM J. Numer. Anal.*, 29(5):1493–1503, October 1992.

[77] E. R. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, Inc., New York, 1992.

[78] E. R. Hansen and R. I. Greenberg. An interval Newton method. *Appl. Math. Comput.*, 12:89–98, 1983.

[79] E. R. Hansen and G. W. Walster. Bounds for Lagrange multipliers and optimal points. *Comput. Math. Appl.*, 25(10):59, 1993.

[80] D. Harper, C. Wooff, and D. Hodgkinson. *A Guide to Computer Algebra Systems.* Wiley, New York, 1991.

[81] E. Heinz. An elementary analytic theory of the degree of mapping in n-dimensional space. *Journal of Mathematics and Mechanics*, 8(2):231–247, 1959.

[82] J. Herzberger. ALGOL-60 procedures evaluating standard functions in interval analysis. *Computing*, 5(4):377–384, 1970.

[83] J. Herzberger, editor. *Topics in Validated Computations*, Studies in Computational Mathematics, Amsterdam, 1994. Elsevier Science Publishers.

[84] R. Horst and M. Pardalos. *Handbook of Global Optimization.* Kluwer, Dordrecht, Netherlands, 1995.

[85] C. Hu. *Optimal Preconditioners for the Interval Newton Method.* PhD thesis, University of Southwestern Louisiana, 1990.

[86] C. Hu and R. B. Kearfott. On bounding the range of some elementary functions in FORTRAN 77. *Interval Computations*, 1993(3):29–39, 1993.

[87] D. Husung. Precompiler for scientific computation (TPX). Technical Report 91.1, Inst. for Comp. Sci. III, Technical University Hamburg–Harburg, 1989.

[88] E. Hyvönen and S. De Pascale. Interval constraint satisfaction tool InC++: A local interval arithmetic library. Technical report, VTT, Tech. Research Center of Finland, 1994.

[89] E. Hyvönen and S. De Pascale. Interval computations on the spreadsheet. In R. B. Kearfott and V. Kreinovich, editors, *Applications of Interval Computations*, Applied Optimization, pages 169–209, Dordrecht, Netherlands, 1996. Kluwer.

[90] E. Hyvönen and S. De Pascale. Shared computations for efficient interval function evaluation. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, Mathematical Research, volume 90, pages 38–44, Berlin, 1996. Akademie Verlag.

[91] K. Ichida and Y. Fujii. An interval arithmetic method for global optimization. *Computing*, 23(1):85–97, 1979.

[92] M. Iri. Simultaneous computation of functions, partial derivatives and estimates of rounding errors – complexity and practicality –. *Japan Journal of Applied Mathematics*, 1(2):223–252, December 1984.

[93] M. Iri and K. Kubota. Methods of fast automatic differentiation and applications. Technical Report RMI 87-02, University of Tokyo, July 1987.

[94] C.-H. Jan. *Expression Parsing and Rigorous Computation of Bounds on All Solutions to Practical Nonlinear Systems.* PhD thesis, University of Southwestern Louisiana, May 1992.

[95] P. Jansen and P. Weidner. High-accuracy arithmetic software – some tests of the ACRITH problem solving routines. *ACM Trans. Math. Software*, 12(1):62–71, 1986.

[96] C. Jansson. A global minimization method: The one-dimensional case. Technical Report 91.2, Technical University Hamburg-Harburg, November 1991.

[97] C. Jansson. A global optimization method using interval arithmetic. In L. Atanassova and J. Herzberger, editors, *Computer Arithmetic and Enclosure Methods*, pages 259–268, Amsterdam, Netherlands, 1992. North-Holland.

[98] C. Jansson. On self-validating methods for optimization problems. In J. Herzberger, editor, *Topics in Validated Computations*, pages 381–439, Amsterdam, Netherlands, 1994. North-Holland.

[99] C. Jansson and O. Knüppel. Numerical results for a self-validating global optimization method. Technical Report 94.1, Technical University Hamburg–Harburg, February 1994.

[100] J. P. Jeter and B. D. Shriver. Variable precision and interval arithmetic: A portable enhancement to FORTRAN. *Adv. Eng. Software*, 6(1):45–50, 1984.

[101] H.-P. Jüllig. Algorithmen mit Ergebnisverification mit C++/2.0. Technical report, Technical University Hamburg-Harburg, 1991.

[102] W. M. Kahan. A more complete interval arithmetic. In *Lecture Notes for a Summer Course at the University of Michigan*, 1968.

[103] E. Kaucher, U. Kulisch, and Ch. Ullrich, editors. *Computer Arithmetic – Scientific Computation and Programming Languages*, Stuttgart, 1987. Teubner.

[104] E. W. Kaucher and W. L. Miranker. *Self-Validating Numerics for Function Space Problems.* Academic Press, Orlando, 1984.

[105] R. B. Kearfott. *Computing the Degree of Maps and a Generalized Method of Bisection.* PhD thesis, University of Utah, Department of Mathematics, 1977.

[106] R. B. Kearfott. An efficient degree-computation method for a generalized method of bisection. *Numer. Math.*, 32:109–127, 1979.

[107] R. B. Kearfott. Abstract generalized bisection and a cost bound. *Math. Comp.*, 49(179):187–202, July 1987.

[108] R. B. Kearfott. Some tests of generalized bisection. *ACM Trans. Math. Software*, 13(3):197–220, September 1987.

[109] R. B. Kearfott. Interval arithmetic techniques in the computational solution of nonlinear systems: Introduction, examples, and comparisons. In E. L. Allgower and K. Georg, editors, *Computational Solution of Nonlinear Systems of Equations (Lectures in Applied Mathematics, volume 26)*, pages 337–358, Providence, RI, 1990. American Mathematical Society.

[110] R. B. Kearfott. Interval Newton / generalized bisection when there are singularities near roots. *Annals of Operations Research*, 25:181–196, 1990.

[111] R. B. Kearfott. Preconditioners for the interval Gauss–Seidel method. *SIAM J. Numer. Anal.*, 27(3):804–822, June 1990.

[112] R. B. Kearfott. Decomposition of arithmetic expressions to improve the behavior of interval iteration for nonlinear systems. *Computing*, 47(2):169–191, 1991.

[113] R. B. Kearfott. An interval branch and bound algorithm for bound constrained optimization problems. *Journal of Global Optimization*, 2:259–280, 1992.

[114] R. B. Kearfott. Empirical evaluation of innovations in interval branch and bound algorithms for nonlinear algebraic systems, 1994. Accepted for publication in *SIAM J. Sci. Comput.*

[115] R. B. Kearfott. On proving existence of feasible points in equality constrained optimization problems, 1994. Preprint, Department of Mathematics, Univ. of Southwestern Louisiana, U.S.L. Box 4-1010, Lafayette, La 70504.

[116] R. B. Kearfott. On verifying feasibility in equality constrained optimization problems. Technical report, University of Southwestern Louisiana, 1994.

[117] R. B. Kearfott. A Fortran 90 environment for research and prototyping of enclosure algorithms for nonlinear equations and global optimization. *ACM Trans. Math. Software*, 21(1):63–78, March 1995.

[118] R. B. Kearfott. Interval extensions of non-smooth functions for global optimization and nonlinear systems solvers, 1995. Accepted for publication in *Computing*.

[119] R. B. Kearfott. Automatic differentiation of conditional branches in an operator overloading context. In George Corliss, editor, *Proceedings of the Second International Workshop on Computational Differentiation*, Philadelphia, 1996. SIAM.

[120] R. B. Kearfott. A review of techniques in the verified solution of constrained global optimization problems. In R. B. Kearfott and V. Kreinovich, editors, *Applications of Interval Computations*, pages 23–60, Dordrecht, Netherlands, 1996. Kluwer.

[121] R. B. Kearfott. Test results for an interval branch and bound algorithm for equality-constrained optimization. In C. Floudas and P. M. Pardalos, editors, *State of the Art in Global Optimization: Computational Methods and Applications*, pages 181–200, Dordrecht, Netherlands, 1996. Kluwer.

[122] R. B. Kearfott. Treating non-smooth functions as smooth functions in global optimization and nonlinear systems solvers. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, Mathematical Research, volume 90, pages 160–172, Berlin, 1996. Akademie Verlag.

[123] R. B. Kearfott, M. Dawande, K.-S. Du, and C.-Y. Hu. Algorithm 737: INTLIB, a portable FORTRAN 77 interval standard function library. *ACM Trans. Math. Software*, 20(4):447–459, December 1994.

[124] R. B. Kearfott, M. Dawande, Du K.-S., and C. Hu. INTLIB: A portable FORTRAN 77 elementary function library. *Interval Computations*, 3(5):96–105, 1992.

[125] R. B. Kearfott and K. Du. The cluster problem in multivariate global optimization. *Journal of Global Optimization*, 5:253–265, 1994.

[126] R. B. Kearfott, C. Hu, and M. Novoa. A review of preconditioners for the interval Gauss–Seidel method. *Interval Computations*, 1(1):59–85, 1991.

[127] R. B. Kearfott and M. Novoa. Algorithm 681: INTBIS, a portable interval Newton/bisection package. *ACM Trans. Math. Software*, 16(2):152–157, June 1990.

[128] R. B. Kearfott and X. Shi. Optimal preconditioners for the interval Gauss–Seidel method. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, Mathematical Research, volume 90, pages 173–178, Berlin, 1996. Akademie Verlag.

[129] R. B. Kearfott and Z. Xing. Rigorous computation of surface patch intersection curves, 1993. Preprint, Department of Mathematics, Univ. of Southwestern Louisiana, U.S.L. Box 4-1010, Lafayette, La 70504.

[130] R. B. Kearfott and Z. Xing. An interval step control for continuation methods. *SIAM J. Numer. Anal.*, 31(3):892–914, June 1994.

[131] G. Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Software*, 6(2):150–165, 1980.

[132] J. B. Keiper. Interval arithmetic in Mathematica. *Interval Computations*, 1993(3):76–87, 1993.

[133] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and Ch. Ullrich. *PASCAL-XSC: A PASCAL Extension for Scientific Computation.* Springer-Verlag, New York, 1991.

[134] R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo, and M. Rauch. *C-XSC; A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, New York, 1993.

[135] O. Knüppel. PROFIL/BIAS — A fast interval library. *Computing*, 53:277–287, 1994.

[136] J. Kok. The embedding of accurate arithmetic in Ada. In P. J. L. Wallis, editor, *Improving Floating Point Programming*, pages 99–120, New York, 1990. Wiley.

[137] C. F. Korn and Ch. Ullrich. Extending LINPACK by verification routines for linear systems. *Mathematics and Computers in Simulation*, 39(1–2):21–37, 1995.

[138] W. Krämer. *Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen.* PhD thesis, Universität Karlsruhe, 1987.

[139] W. Krämer. Multiple precision computations with result verification. In E. Adams and U. Kulisch, editors, *Scientific Computing with Automatic Result Verification*, Mathematics in Science and Engineering, volume 189, pages 325–256, New York, 1993. Academic Press.

[140] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlershranken. *Computing*, 4:187–201, 1969.

[141] R. Krawczyk and A. Neumaier. Interval slopes for rational functions and associated centered forms. *SIAM J. Numer. Anal.*, 22(3):604–616, June 1985.

[142] R. Krawczyk and K. Nickel. The centered form in interval arithmetics: Quadratic convergence and inclusion isotonicity. *Computing*, 28(2):117–137, 1982.

[143] V. N. Krishchuk, N. M. Vasilega, and G. L. Kozina. Interval operations and functions library for FORTRAN 77 programming system and its practice using. *Interval Computations*, 4(6):2–8, 1992.

[144] B. P. Kristinsdottir, Z. B. Zabinsky, T. Csendes, and M. E. Tuttle. Methodologies for tolerance intervals. *Interval Computations*, 1993(3):133–147, 1993.

[145] U. W. Kulisch. A new arithmetic for scientific computation. In U. W. Kulisch and W. L. Miranker, editors, *A New Approach to Scientific Computation*, Notes and Reports in Comp. Sci. and Applied Math., pages 1–26, New York, 1983. Academic Press.

[146] U. W. Kulisch and W. L. Miranker, editors. *A New Approach to Scientific Computation*, Notes and Reports in Comp. Sci. and Applied Math., New York, 1983. Academic Press.

[147] U. W. Kulisch and W. L. Miranker. The arithmetic of the digital computer: A new approach. *SIAM Rev.*, 28(1):1–40, 1986.

[148] L. Kupriyanova. Inner estimation of the united solution set of interval algebraic systems. *Reliable Computing*, 1:15–32, 1995.

[149] S. E. Laveuve. Definition einer Kahan-Arithmetik und ihre Implementierung. In K. Nickel, editor, *Interval Mathematics*, Lecture Notes in Computer Science 25, pages 236–245, New York, 1975. Springer-Verlag.

[150] C. Lawo. C-XSC – a programming environment for verified scientific computing and numerical data processing. In E. Adams and U. Kulisch, editors, *Scientific Computing with Automatic Result Verification*, pages 71–86, New York, etc., 1993. Academic Press.

[151] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Algorithm 539 basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5(3):308–325, September 1979.

[152] A. Leclerc. Parallel interval global optimization in C++. *Interval Computations*, 1993(3):148–163, 1993.

[153] N. G. Lloyd. *Degree Theory.* Cambridge University Press, Cambridge, England, 1978.

[154] W. A. Lodwick. Constraint propagation, relational arithmetic in AI systems and mathematical programs. *Ann. Oper. Res.*, 21(1-4):143–148, 1989.

[155] W. Luther and W. Otten. Computation of standard interval functions in multiple-precision interval arithmetic. Technical Report SM-DU-233, Uni. Duisburg Gesamthochschule, 1993.

[156] M. A. MacCallum. *Algebraic Computing with REDUCE.* Oxford University Press, London, 1992.

[157] S. Markov. On an interval arithmetic and its applications. In *Proceedings of the 5th Symposium on Computer Arithmetic. IEEE. Univ. Michigan. 1981. Ann Arbor, MI, USA. 18-19 May 1981.*, 1981.

[158] S. M. Markov. Some applications of extended interval arithmetic to interval iterations. *Computing (Suppl.)*, 2:69–84, 1980.

[159] G. Mayer. Epsilon–inflation in verification algorithms. *J. Comput. Appl. Math.*, 60:147–169, 1994.

[160] M. Metzger. FORTRAN-SC, a FORTRAN extension for engineering / scientific computation with access to ACRITH: Demonstration. In R. E. Moore, editor, *Reliability in Computing*, Perspectives in Computing, pages 63–80, New York, 1988. Academic Press.

[161] C. Miranda. Un' osservatione su un teorema di Brouwer. *Bol. Un. Mat. Ital., Series 2*, 2:5–7, 1940.

[162] V. Mladenov. An improved interval method for solving nonlinear systems of monotone functions. In S. M. Markov, editor, *Mathematical Modelling and Scientific Computing*, pages 23–26, Sofia, 1993. DATECS Publishing.

[163] R. E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing.* PhD thesis, Stanford University, October 1962.

[164] R. E. Moore. A test for existence of solutions to nonlinear systems. *SIAM J. Numer. Anal.*, 14(4):611–615, September 1977.

[165] R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[166] R. E. Moore, E. Hansen, and A. Leclerc. Rigorous methods for parallel global optimization. In A. Floudas and P. Pardalos, editors, *Recent Advances in Global Optimization*, pages 321–342, Princeton, N.J., 1992. Princeton Univ. Press.

[167] R. E. Moore and S. T. Jones. Safe starting regions for iterative methods. *SIAM J. Numer. Anal.*, 14(6):1051–1065, December 1977.

[168] R. E. Moore and H. Ratschek. Inclusion functions and global optimization II. *Math. Prog.*, 41(3):341–356, September 1988.

[169] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. User guide for MINPACK-1. Technical Report ANL-80-74, Argonne National Laboratories, 1980.

[170] J. J. Moré and S. J. Wright. *Optimization Software Guide.* Frontiers in Applied Mathematics 14. SIAM, Philadelphia, 1993.

[171] A. P. Morgan. A method for computing all solutions to systems of polynomial equations. *ACM Trans. Math. Software*, 9(1):1–17, 1983.

[172] A. P. Morgan and A. J. Sommese. Computing all solutions to polynomial systems using homotopy continuation. *Appl. Math. Comput.*, 24(2):115–138, 1987.

[173] S. P. Mudur and P. A. Koparkar. Interval methods for processing geometric objects. *IEEE Comput. Graphics and Appl.*, 4(2):7–17, February 1984.

[174] B. A. Murtagh and M. A. Saunders. Minos 5.1 user's guide. Technical Report SOL 83-20R, Dept. Operations Res., Stanford University, 1987.

[175] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, England, 1990.

[176] A. Neumaier. A compact input format for nonlinear optimization problems, 1993. Preprint, Institut für Mathematik, Universität Wien, Strudhofgasse 4, A-1050 Wien, Austria.

[177] A. Neumaier. Second-order sufficient optimality conditions for local and global nonlinear programming, 1994. Accepted for publication in *Journal of Global Optimization.*

[178] A. Neumaier and Z. Shen. The Krawczyk operator and Kantorovich's theorem. *Mathematical Analysis and Applications*, 149(2):437–443, July 1990.

[179] S. Ning. A report on code lists for higher-order derivatives summary of a fall, 1993 individual study course. Technical Report Dept. Math., Univ. of Southwestern La., University of Southwestern Louisiana, 1993.

[180] Manuel III Novoa. Theory of preconditioners for the interval Gauss–Seidel method and existence / uniqueness theory with interval Newton methods, 1993. Preprint, Department of Mathematics, University of Southwestern Louisiana, U.S.L. Box 4-1010, Lafayette, La 70504.

[181] W. Oettli. On the solution set of a linear system with inaccurate coefficients. *SIAM J. Numer. Anal.*, 2:115–118, 1965.

[182] P. M. Pardalos and J. B. Rosen. *Constrained Global Optimization: Algorithms and Applications.* Lecture Notes in Computer Science no. 268. Springer-Verlag, New York, 1987.

[183] P. M. Pardalos and S. A. Vavasis. Quadratic programming with one negative eigenvalue is NP-hard. *Journal of Global Optimization*, 1(1):15–22, 1992.

[184] G. Peterson. *Object-Oriented Computing, Volume 1: Concepts.* IEEE Computer Society Press, Washington, D.C., 1987.

[185] L. Qiao. Basic interval arithmetic subroutines library in the C language. Technical Report 335, Math./Stat./Comp. Sci., Marquette University, November 1990.

[186] L. B. Rall. A comparison of the existence theorems of Kantorovich and Moore. *SIAM J. Numer. Anal.*, 17(1):148–161, 1980.

[187] L. B. Rall. *Automatic Differentiation: Techniques and Applications.* Lecture Notes in Computer Science no. 120. Springer, Berlin, New York, etc., 1981.

[188] L. B. Rall. An introduction to the scientific computing language Pascal-SC. *Computers and Mathematics with Applications*, 14(1):53–59, 1987.

[189] R. H. Rand. *Computer Algebra in Applied Mathematics: An Introduction to MACSYMA.* Research notes in mathematics. Pitman Advanced Publishing Program, Boston, London, Melbourne, 1984.

[190] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions.* Horwood, Chichester, England, 1984.

[191] H. Ratschek and J. Rokne. *New Computer Methods for Global Optimization.* Wiley, New York, 1988.

[192] H. Ratschek and J. Rokne. Formulas for the width of interval products. *Reliable Computing*, 1(1):9–14, 1995.

[193] H. Ratschek and R. L. Voller. Global optimization over unbounded domains. *SIAM J. Control Optim.*, 28(3):528–539, 1990.

[194] D. Ratz. *Automatische Ergebnisverifikation bei globalen Optimierungsproblemen.* PhD thesis, Universität Karlsruhe, 1992.

[195] D. Ratz. An inclusion algorithm for global optimization in a portable PASCAL-XSC implementation. In L. Atanassova and J. Herzberger, editors, *Computer Arithmetic and Enclosure Methods*, pages 329–338, Amsterdam, Netherlands, 1992. North-Holland.

[196] D. Ratz. Box-splitting strategies for the interval Gauss–Seidel step in a global optimization method. *Computing*, 53:337–354, 1994.

[197] D. Ratz. On branching rules in second-order branch-and-bound methods for global optimization. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, Mathematical Research, volume 90, pages 221–227, Berlin, 1996. Akademie Verlag.

[198] D. Ratz. On extended interval arithmetic and inclusion isotonicity, 1996. Preprint, Institut f. Angew. Mathematik, Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany.

[199] D. Ratz and T. Csendes. On the selection of subdivision directions in interval branch-and-bound methods for global optimization. *Journal of Global Optimization*, 7:183–207, 1995.

[200] W. C. Rheinboldt and J. M. Ortega. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, 1970.

[201] J. Rohn. Solving systems of linear interval equations. In R. E. Moore, editor, *Reliability in Computing*, Perspectives in Computing, pages 171–182, New York, 1988. Academic Press.

[202] J. Rohn. New condition numbers for matrices and linear systems. *Computing*, 41(1–2):167–169, 1989.

[203] J. Rohn. NP-hardness results for linear algebraic problems with interval data, 1994. Preprint, Faculty of Math. and Physics, Charles University, Malostranske nam. 25, 11800 Praha 1, Czech Republic.

[204] J. Rohn and V. Kreinovich. Computing exact componentwise bounds on solutions of linear systems with interval data is NP-hard. *SIAM J. Matrix Anal. Appl.*, 16(2), April 1995.

[205] S. M. Rump. *Kleine Fehlerschranken bei Matrixproblemen.* PhD thesis, Universität Karlsruhe, 1980.

[206] S. M. Rump. ACRITH – high accuracy arithmetic subroutine library. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, New York, 1985. Springer-Verlag.

[207] S. M. Rump. On the solution of interval linear systems. *Computing*, 47:337–353, 1992.

[208] S. M. Rump. Verification methods for dense and sparse systems of equations. In J. Herzberger, editor, *Topics in Validated Computations*, pages 63–135, Amsterdam, 1994. Elsevier Science Publishers.

[209] S. M. Rump. Bounds for the componentwise distance to the nearest singular matrix. Technical Report 95.3, Technical University Hamburg-Harburg, 1995.

[210] S. M. Rump. Expansion and estimation of the range of nonlinear functions, 1995. Preprint, Informatik III – Programmiersprachen und Algorithmen, Technische Universität Hamburg, Eissendorfer Str. 38, 2100 Hamburg 90, Germany.

[211] S. M. Rump. New results on validation algorithms for large systems of equations, 1995. Talk given at SCAN'95, Wuppertal, Germany, Sept. 26 – 29, 1995.

[212] C. A. Schnepper. *Large Grained Parallelism in Equation-Based Flowsheeting Using Interval Newton / Generalized Bisection Techniques.* PhD thesis, University of Illinois, Urbana, 1992.

[213] C. A. Schnepper and M. A. Stadtherr. Application of a parallel interval Newton / generalized bisection algorithm to equation-based chemical process flowsheeting. *Interval Computations*, 1993(4):40–64, 1993.

[214] T. W. Sederberg and S. R. Parry. Comparison of three curve intersection algorithms. *Comput. Aided Des.*, 18(1):58–63, 1986.

[215] S. P. Shary. Solving the tolerance problem for interval linear systems. *Interval Computations*, 1994(2), 1994.

[216] S. P. Shary. On optimal solution of interval linear equations. *SIAM J. Numer. Anal.*, 32(2):610–630, April 1995.

[217] S. P. Shary. Algebraic approach to the interval linear static identification, tolerance and control problems. *Reliable Computing*, 2(1):3–34, 1996.

[218] Z. Shen, A. Neumaier, and M. C. Eiermann. Solving minimax problems by interval methods. *BIT*, 30:742–751, 1990.

[219] Z. Shen and M. A. Wolfe. On interval enclosures using slope arithmetic. Technical report, Dept. Math., Univ. of St. Andrews, Scotland, September 1989.

[220] Z. Shen and M. A. Wolfe. A note on the comparison of the Kantorovich and Moore theorems. *Nonlinear Analysis*, 15(3):229–232, 1990.

[221] E. C. Sherbrooke and N. M. Patrikalakis. Computation of the solutions of nonlinear polynomial systems. *Computer Aided Geometric Design*, 10:379–405, 1993.

[222] X. Shi. *Intermediate Expression Preconditioning and Verification for Rigorous Solution of Nonlinear Systems.* PhD thesis, University of Southwestern Louisiana, Department of Mathematics, August 1995.

[223] D. Shiriaev. *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment.* PhD thesis, University of Karlsruhe, 1993.

[224] S. Skelboe. Computation of rational interval functions. *BIT*, 14:87–95, 1974.

[225] B. Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical Report R-80-1002, Univ. of Illinois at Urbana–Chambaign, January 1980.

[226] F. Stenger. An algorithm for the topological degree of a mapping in $\mathbb{R}^n$. *Numer. Math.*, 25:23–38, 1976.

[227] Stevenson, D., chairman, Floating-Point Working Group, Microprocessor Standards Subcommittee. IEEE standard for binary floating point arithmetic (IEEE/ANSI 754-1985). Technical report, IEEE, 1985.

[228] M. Stynes. *An Algorithm for the Numerical Calculation of the Degree of a Mapping.* PhD thesis, Oregon State University, Department of Mathematics, Corvallis, Oregon, 1977.

[229] Symbolic Computation Group Staff. *Maple, Version 4.2.1.* Brooks/Cole, Monterey, California, 1990.

[230] University of Minnesota Computer Center. *M77 Reference Manual: 1977 Standards Version Edition 1.* University of Minnesota, 1983.

[231] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[232] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. Technical Report CS-95-01, Dept. of Comp. Sci., Brown University, 1995.

[233] R. J. Van Iwaarden. *An Improved Unconstrained Global Optimization Algorithm.* PhD thesis, University of Colorado at Denver, 1996.

[234] M. N. Vrahatis. Solving systems of nonlinear equations using the nonzero value of the topological degree. *ACM Trans. Math. Software*, 14(4):312–336, December 1988.

[235] G. W. Walster, E. R. Hansen, and S. Sengupta. Test results for a global optimization algorithm. In P. T. Boggs, R. H. Byrd, and R. B. Schnabel, editors, *Numerical Optimization 1984*, pages 272–287, Philadelphia, 1985. SIAM.

[236] W. V. Walter. FORTRAN-SC, a Fortran extension for engineering / scientific computation with access to ACRITH: Language description. In R. E. Moore, editor, *Reliability in Computing*, Perspectives in Computing, pages 43–62, New York, 1988. Academic Press.

[237] W. V. Walter. ACRITH-XSC: A Fortran-like language for verified scientific computing. In E. Adams and U. Kulisch, editors, *Scientific Computing with Automatic Result Verification*, pages 45–70, New York, etc., 1993. Academic Press.

[238] W. V. Walter. FORTRAN-XSC: A portable Fortran 90 module library for accurate and reliable scientific computing. *Computing (Suppl.)*, 9:265–286, 1993.

[239] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, August 1964.

[240] A. S. Wexler. Automatic evaluation of derivatives. *Appl. Math. Comput.*, 24:19–26, 1987.

[241] J. H. Wilkinson. Modern error analysis. *SIAM Rev.*, 13(4):548–568, 1971.

[242] M. A. Wolfe. An interval algorithm for constrained global optimization. *J. Comput. Appl. Math.*, 50:605–612, 1994.

[243] J. Wolff von Gudenberg. Programming language support for scientific computation. *Interval Computations*, 1992(4):116–126, 1992.

[244] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer (Second Edition).* Addison-Wesley, Reading, MA, 1991.

[245] Zh. Xing. *Rigorous Step Control for Continuation.* PhD thesis, University of Southwestern Louisiana, 1993.

[246] A. G. Yakovlev. Multiaspectness and localization. *Interval Computations*, 1993(4):195–209, 1993.

[247] J. M. Yohe. Software for interval arithmetic: A reasonably portable package. *ACM Trans. Math. Software*, 5(1):50–53, March 1979.

INDEX

Accurate dot product, 8, 103–104, 106–107
ACRITH, 104–105
ACRITH-XSC, 102, 104–105
Ada, 44, 105, 107
Algol, 102
Algol-60, 103
Anonymous FTP, xv, 71, 76, 107, 111, 139, 166
Approximate optimizer, 175
Augment precompiler, 103
Automatic differentiation, 29, 36–37, 48, 94–95, 209
  forward mode, 37–38, 48
  reverse mode, 38, 48, 96
Automatic verification, 24
BIAS, 107–108
BIBINS, 107
Binary code list, 85
Bisection, 122, 148, 173–175
BLAS, 72, 107
BNR Prolog, 230, 232
Bound constraints, 170, 172, 175, 178, 180–182, 185, 187, 190–198, 201, 204, 206
Box complementation, 145–147, 154, 157
Box, 1, 6, 95–96
Box-splitting strategies, 174–175
Branch and bound, 145, 171, 175
Branch function, 87, 211
Brouwer degree, 66–67, 111
Brouwer fixed point theorem, 18, 60–61
C, 106
C++, 44, 103, 105–107, 109, 111, 176, 230
  Borland, version 4, 106
C-LP-DENSE, 140
C-preconditioner, 124–125, 128
C-XSC, 8, 106
  free version via FTP, 106
Cancellation subtraction, 5
Case statement, 48
  Fortran-90, 48
CDC mainframes, 104
CDLINEQ, 86
CDLLHS, 45, 83, 86
CDLVAR, 45, 83, 86–87
Centered form, 16
Certainly feasible, 179
CHI function, 87, 211
Circular arithmetic, 10
CM-LP-preconditioner, 136
Code list, 43–45, 47–48, 50, 78, 83, 85, 87–89, 91–101, 159, 161, 199–201, 227–228, 231–234
  ASCII, 85
  binary, 85
  derivative, 91, 93
  first derivative, 94, 99
  gradient, 94–95, 99, 101, 199, 201–202, 207, 230, 233
  second derivative, 93
CODELIST-VARIABLES, module, 95
Common subexpressions, 101
Complementation, box, 145–147, 154, 157
Complete pivoting, 189

Complex interval arithmetic, 10
  circular, 10
  rectangular, 10
Complex interval dependency, 10
COMPONENT-SOLVE, 232
Computational cost, 97
Computational fixed point theory, 59
Concavity test, 172
Configuration file, 83
Constraint logic programming, 109–110
Constraint processing, 110
Constraint propagation, 110, 129, 230, 232
Constraint satisfaction, 109
Constraints, xi, 99, 169–170, 177–180, 185, 203, 206
  bound, 170, 172, 178, 180–182, 185, 187, 190–198, 201, 204, 206
  equality, 170, 177, 179–180, 185, 188, 195, 200
  inequality, 169, 177–180
Continuation method, 191
Controlled solution set, 19
Convergence theory, interval Newton methods, 210, 219
Conversion error, 85
Convex hull, 6
Convexity, 173
CP/M, 104
Cray machines, 78
Degeneracy, 69
DENSE-GAUSS-SEIDEL-STEP, 234
DENSE-JACOBI-MATRIX, subroutine, 96
DENSE-NLE-BOX-LIST, 165–166
DENSE-SLOPE-MATRIX, 96, 227
Dependency, 4, 233
Dependent variable, 93–94
Derivative code list, 91–95, 212
Derivative tensor, 94
Diameter, relative, 146
Differentiation arithmetic, 37, 41
Differentiation, 83
  automatic, 29, 36–38
  numerical, 36
  symbolic, 36, 43, 91
Directed rounding, 3, 7, 107, 115–116, 120
  simulated, 8, 71–72
Distributed-memory, 176
DOS, 104, 109
Dot product, accurate, 8, 103–107
E-preconditioner, 124–128
Eigenvalues, 173
Einzelschrittverfahren, 21
Epsilon-inflation, 143, 145, 147, 150–151, 176, 204, 207, 209
Equality constraints, 170, 178–180, 185, 188, 195, 200
  feasibility of, 178
  proving feasibility, 179
Excel, 103, 109
Existence verification, 18, 25, 59–61, 63, 68, 126, 128, 143, 150–151, 171–172, 174, 176, 219
Expanded system, 39, 109, 228–229, 232
Expression swell, 38
Extended interval arithmetic, 3, 9, 175
Extended interval, 9
F-2, subroutine, 96
F-POINT, subroutine, 96
Feasibility, proving, 172–173, 178–179, 195
First derivative code list, 94, 99
First order interval extension, 14
Fixed point iteration, 24, 56
Fixed point theory, 50–51, 60–61

classical, 51
computational, 56, 59
interval, 50
FORTRAN IV, 103–104
FORTRAN-66, 103
FORTRAN-77, 8, 11, 13, 44, 71–73, 76, 101, 104, 110–112, 175
Fortran-90, 7, xiii–xiv, 44–47, 50, 71, 76, 78–81, 83, 86–87, 97, 104–106, 109–110, 167, 230–232
FORTRAN-SC, 8, 102, 104
Fortran-XSC, 106
Forward mode, automatic differentiation, 37–38, 48
Forward substitution, 231
FORWARD-SUBSTITUTION, 233
Frechét arithmetic, 94
Fritz John equations, 100, 170, 172, 179, 194–196, 204, 208
FTP, xv, 71, 76, 106–107, 111, 139, 166
F, subroutine, 95
Gap, 127
Gauss–Seidel method, interval, 21–22, 58–59, 61, 113–114, 117, 119, 121–124, 126, 141, 143, 148, 150, 175, 179, 204, 227, 229, 232, 189, 191
GAUSS-SEIDEL-STEP, 234
Gaussian elimination, complete pivoting, 189
Gaussian elimination, interval, 21, 51, 58, 113–114, 118–119, 121–123, 141, 143, 179, 196
Generalized bisection, 69, 122, 141, 145, 148, 154, 157, 175–176, 232
coordinate selection strategies, 157
Generic functions, 47
Gradient code list, 94–95, 99, 101, 199, 201–202, 207, 230, 233
Gradient
reduced, 101
GRADIENT-CODELIST, module, 91, 94
Hansen slope, 33, 35, 96, 227, 234
Hansen's algorithm, 174
Hessian, 94, 99
reduced, 101
Horner's method, 4, 17
Hull
interval, 20
solution set, 20
Hypercube, 176
IBM 360, 103, 105
IBM 370, 104
IBM PC, 107, 111
Idealized interval arithmetic, 3
IEEE binary floating point standard, 7
Ill-conditioning, 150
InC++, 107, 109
Inclusion isotonic, 17
Inclusion monotonic, 17, 35
Inequality constraints, 169, 177–180
feasibility of, 178
proving feasibility, 179
Infeasibility, proving, 172–173, 178–180, 194
INFFOR, 112
INITIALIZE-CODELIST
module, 87
Inner estimates, 9, 20
INTCOM, 111
Interior, 6
Intermediate variable, 91, 94
Internal representation, 43
Interpreter, 48
Interval arithmetic, 3
complex, 10

directed rounding, 7
extended, 3, 9
idealized, 3
Kahan, 3, 8
Kahan–Novoa–Ratz, 54
operational definitions, 3
ordinary interval division, 3
overestimation, 4, 8
complex, 10
real, 3
rounded, 7
software packages, 8
subdistributivity, 4
Interval constraint propagation, 109
Interval data type, 78
Interval dependency, 4, 233
complex, 10
Interval enclosures, 3
Interval extension, 11, 209
centered form, 16
first order, 14
inclusion isotonic, 17
inclusion monotonic, 17
mean value, 15, 96
natural, 11–13, 15–17, 26–27, 35–36, 56, 96, 142, 149, 227
non-smooth, 209
order, 14, 177
properties, 13
second order, 14, 147, 232
united, 6
Interval Gauss algorithm, 21
Interval Gauss–Seidel method, 21–22, 58–59, 61, 63, 113–114, 117, 119, 121–124, 126, 141, 143, 148, 150, 175, 179, 189, 191, 204, 227, 229, 232
Interval Gaussian elimination, 21, 51, 58, 63, 113–114, 118–119, 121–123, 141, 143, 179, 196
Interval hull, 20
Interval Jacobi matrix, 27, 121–122, 229, 234
Interval linear systems, 18, 113
preconditioning, 21
Interval Newton methods, 18, 25–26, 32–33, 50–51, 55, 59–60, 63, 69, 108, 114–115, 117, 120–121, 150, 157, 174–176, 185, 187–192, 194–196, 198–199, 209–210, 215, 219, 229, 232, 234
convergence theory, 210, 219
multivariate, 55–56
quadratic convergence, 59
univariate, 51–52
Interval, 3
dependency, 4
complex, 10
thin, 6
INTERVAL-ARITHMETIC, module, 78, 80, 105
INTLIB, 8, 44, 71–74, 76, 78–79, 81, 107, 166
INTLIB-90, xiv, 45, 78, 89, 91–93, 95, 101–103, 105–106, 110, 150, 166, 211, 228, 231, 234
INTLIB-ARITHMETIC, module, 107
INTNEWT.CFG, 199
INTOPT-90, xiv, 78, 94, 140, 145–146, 153, 159, 162, 165–167, 170, 199, 201, 205, 207, 227, 232–234
Inverse midpoint preconditioner, 114–115, 118–122, 141–142
optimality properties, 118
IVL, 80
Jacobi matrix, 27, 66–67, 94, 96, 99, 188, 196
ill-conditioned, 150

interval, 27, 119, 121–122, 229, 234
singular, 150
Kahan arithmetic, 53
Kahan–Novoa–Ratz arithmetic, 3, 8–9, 54, 115, 124, 127, 136, 175
Kantorovich theorem, 60
Karlsruhe University, 105
Kaucher arithmetic, 9
Krawczyk method, 21, 24, 51, 56–58, 60, 113, 116, 119, 141–142
Krawczyk operator, 56
Lagrange multiplier, 195–196, 198
LANCELOT, 36, 88, 205
LANCELOT-OPT, 204–205, 207–208
Left-optimal preconditioner, 125–126, 138
Linear algebra, numerical, 107
Linear programming, 185
Linear systems of equations, interval, 18, 113
Linear systems of equations, interval
solution set, 19
Linked list, 165
Lipschitz matrix, 26–27, 33, 60, 62–64, 210, 212, 219
Lipschitz set, 220
Load balancing, 176
Local optimization, 175
Logic programming languages, 109–110
Logic programming, 110, 230
Logical-valued operators, 80
LP-preconditioners, 136, 142, 175, 233
CM, 136
M77 compiler, 104
Machine interval arithmetic, 7
MACSYMA, 36, 95
Magnitude, 5
Magnitude-optimal LP preconditioner, 136
Magnitude-optimal preconditioner, 125–126, 128–129, 134–135, 137, 141, 143, 189
MAKE-GRADIENT, 201
Maple, 36, 95, 103, 108
Markov arithmetic, 9
Mathematica, 36, 95, 103, 108
Mathematics Research Center, 104
Maximum smear, 153, 157, 175
Mean value extension, 15, 56, 96, 219
Midpoint test, 172–175, 204
Mignitude, 5
Mignitude-optimal LP-preconditioner, 137
Mignitude-optimal preconditioner, 127
Mignitude-optimal splitting preconditioner, 127–129, 136, 138–139, 141, 143, 167, 194
Minimum smear, 153
Miranda's theorem, 60–61
Modula-2, 107
Modula-SC, 107
Monotonicity test, 172–173, 175, 205
Moore–Skelboe algorithm, 110, 173–174, 232
Multivariate interval Newton methods, 55–56
NAG, 97, 231
Natural interval extension, 11–13, 15–17, 26–27, 35–36, 56, 96, 142, 149, 227
NEAREST, Fortran-90, 7
NETLIB, 76
Newton methods, interval, 18, 25–26, 32–33, 50–51, 55,

59–60, 69, 108, 120–121, 185, 187–192, 194–195, 198–199, 229, 232, 234
Non-convexity, 173
Non-smooth functions, interval extensions of, 209
NP-completeness, 180
Numerical differentiation, 36
Operator overloading, 43, 83, 103–104
Optimal LP-preconditioners, 124
Optimal preconditioners, 114, 129, 175, 179, 191, 229
  width-, 123
OPTTBND.CFG, 199, 204–205
Order, interval extension, 14, 177
Ordinary interval division, 3
Outer estimates, 9, 20
Outward rounding, 7
Overestimation, 88, 157, 189
Overhead, 97
OVERLOAD.CFG, 85, 159, 161, 199
OVERLOAD, module, 83
Parallelization, 176
Parameter-fitting, 176
Parametrized systems, 154
Partial separability, 88
Pascal-SC, 8, 104–106
Pascal-XSC, 8, 106, 174
Peeling, viii–ix, 181, 183–184, 198
Penalty function, 179
PL/I, 104, 112
POINT-JACOBI-MATRIX, subroutine, 96
Portability, 105
Positive definite, 173
PRECISION BASIC, 111
Preconditioner
  width-optimal, 143
Preconditioning, 21, 55, 58, 113
  inverse midpoint, 114–115, 118–122, 141–142
  left-optimal, 125–126, 138
  magnitude-optimal, 125–126, 128–129, 134–135, 137, 141, 143, 189
  mignitude-optimal splitting, 127–129, 136, 138, 141, 143, 167, 194
  mignitude-optimal, 127
  optimal C, 125
  optimal LP, 124, 128, 233
    theory, 138
  optimal, 114, 134, 136–138, 179
    C, 124–125
    E, 124–128
    LP, 134, 233
    S, 124
    width-, 123
  right-optimal, 125–126, 138
  splitting, 167
  width-optimal, 125–126, 128–129, 133, 137, 140–141, 143, 150, 167, 189, 199, 229
PROFIL, 107
Prolog, 103, 110, 230, 232
  BNR, 230, 232
Proving feasibility, 178–179, 195
  equality constraints, 178–179
  inequality constraints, 178–179
Quadratic convergence, 52, 59, 65, 172, 174
Quadratic programming, 180
Range arithmetic, 110
RATFOR, 104
Rational approximations, 13
Real interval arithmetic, 3
Reduce, 36, 95
Reduced gradient, 101, 172, 180, 182
Reduced Hessian matrix, 101
Regular, 19, 62

Regularity, 18
  strong, 119
Relative diameter, 146
Relative width, 146
Reverse mode, automatic differentiation, 38, 48, 94, 96
Right-optimal preconditioner, 125–126, 138
RNDOUT, subroutine, 73
ROOTS-DELETE, 159–160, 162, 165–167
ROOTSDL.CFG, 160–161, 166
Rounded interval arithmetic, 7
Roundout error, 7, 17, 42
RUN-GLOBAL-OPTIMIZATION, 198–202, 204–205, 207–208
RUN-ROOTS-DELETE, 159–160, 162, 166, 198–199
Russian Institute of Artificial Intelligence, 109
S-preconditioner, 124
SC languages, 8, 44, 102, 104
Second derivative code list, 93–94
Second order interval extension, 26, 147, 232
SEPAFOR, 103
Separable, 88
SIMINI, subroutine, 76
Simulated directed rounding, 8, 71–72
Single-step method, 21
Singularity, 66, 69
Slack variables, 178
Slope arithmetic, 41
Slope matrix, 26–28, 30, 33, 35, 41–42, 51, 58, 60, 63–66, 96, 219
Slopes, 27
  multivariate, 29
  univariate, 27
Smear, 153, 175
Smear, maximum, 157, 175
Software packages, 8
Solution hull, 20
Solution set
  interval linear systems, 19
Splitting preconditioners, 136, 167
Splitting strategies, 174
Spreadsheet, 109
Standard function, 11–15, 17–18, 37, 41, 44, 50, 71, 74, 103, 105
Strongly regular, 119, 141
Subdefinite calculations, 109, 230
Subdistributivity, 4
SUBSIT, subroutine, 233–234
Substitution-iteration, 101, 109, 205, 228–230, 232–234
Sun, 107
Symbolic differentiation, 36, 43, 91, 94–95
Symbolic manipulation packages, 36, 108–109
  MACSYMA, 95
  Maple, 95, 103, 108
  Mathematica, 95, 103, 108
  Reduce, 95
Symmetric interval, 115
Syntax rules, 86
Taylor arithmetic, 107
Tessellation, 145, 172, 175, 178, 205, 230
Thin interval, 6
Tolerable solution set, 19
Topological degree, 66–67, 69, 111
Tree, 182
Triplex-Algol, 103
Trisection, 176
ULP, 8, 106
Unconstrained optimization, 174
UniCalc, 103, 109, 230
Uniqueness verification, 18, 25, 59–60, 63–66, 126, 128, 143,

150–151, 171–172, 174, 176, 195, 219
United extension, 6
United solution set, 19
  inner estimates, 20
  outer estimates, 20
Univac, 103
Unix, 167
Variable precision, 103, 106, 110–111
Verification
  existence, 18, 25, 59, 61, 63, 68, 126, 128, 150–151, 219
  uniqueness, 150
Width, 6
  relative, 146
Width-optimal LP-preconditioner, 134, 138
Width-optimal preconditioner, 123, 125–126, 128–129, 133, 137, 140–141, 143, 150, 167, 189, 199, 229
Winding number, 67
World Wide Web, 71, 74
XSC languages, 105
  C-XSC, 106
  Fortran-XSC, 106
  free version of C-XSC via FTP, 106
  Pascal-XSC, 106

Nonconvex Optimization and Its Applications

1. D.-Z. Du and J. Sun (eds.): *Advances in Optimization and Approximation.* 1994. ISBN 0-7923-2785-3
2. R. Horst and P.M. Pardalos (eds.): *Handbook of Global Optimization.* 1995 ISBN 0-7923-3120-6
3. R. Horst, P.M. Pardalos and N.V. Thoai: *Introduction to Global Optimization* 1995 ISBN 0-7923-3556-2; Pb 0-7923-3557-0
4. D.-Z. Du and P.M. Pardalos (eds.): *Minimax and Applications.* 1995 ISBN 0-7923-3615-1
5. P.M. Pardalos, Y. Siskos and C. Zopounidis (eds.): *Advances in Multicriteria Analysis.* 1995 ISBN 0-7923-3671-2
6. J.D. Pintér: *Global Optimization in Action.* Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications. 1996 ISBN 0-7923-3757-3
7. C.A. Floudas and P.M. Pardalos (eds.): *State of the Art in Global Optimization.* Computational Methods and Applications. 1996 ISBN 0-7923-3838-3
8. J.L. Higle and S. Sen: *Stochastic Decomposition.* A Statistical Method for Large Scale Stochastic Linear Programming. 1996 ISBN 0-7923-3840-5
9. I.E. Grossmann (ed.): *Global Optimization in Engineering Design.* 1996 ISBN 0-7923-3881-2
10. V.F. Dem'yanov, G.E. Stavroulakis, L.N. Polyakova and P.D. Panagiotopoulos: *Quasidifferentiability and Nonsmooth Modelling in Mechanics, Engineering and Economics.* 1996 ISBN 0-7923-4093-0
11. B. Mirkin: *Mathematical Classification and Clustering.* 1996 ISBN 0-7923-4159-7
12. B. Roy: *Multicriteria Methodology for Decision Aiding.* 1996 ISBN 0-7923-4166-X
13. R.B. Kearfott: *Rigorous Global Search: Continuous Problems.* 1996 ISBN 0-7923-4238-0

KLUWER ACADEMIC PUBLISHERS – DORDRECHT / BOSTON / LONDON