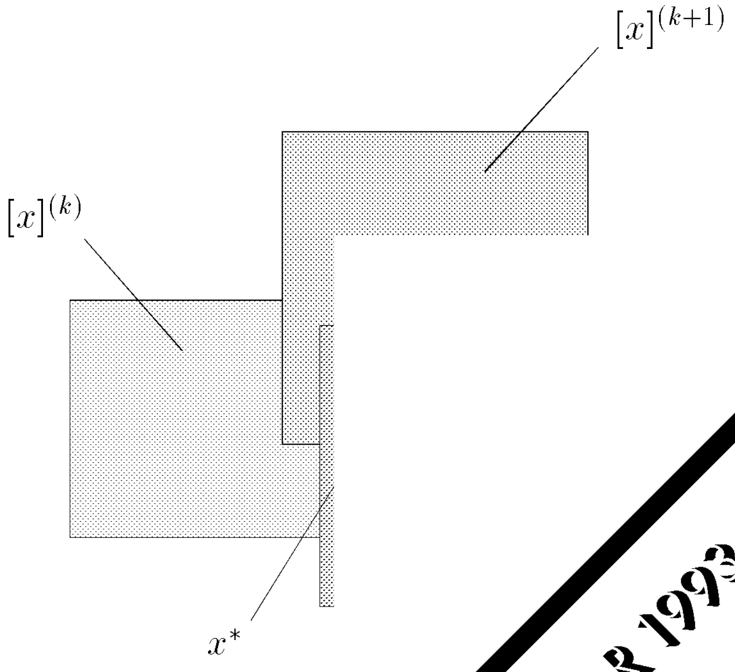
Springer Series in
Computational
Mathematics 21

R. Hammer, M. Hocks, U. Kulisch, D. Ratz

Numerical Toolbox for Verified Computing I

Basic Numerical Problems



TO APPEAR 1993 / EXCERPTS

Springer-Verlag

ISBN 3-540-57118-3 (Heidelberg)
ISBN 0-387-57118-3 (New York)

Preface

As suggested by the title of this book *Numerical Toolbox for Verified Computing*, we present an extensive set of sophisticated tools to solve basic numerical problems with a *verification of the results*. We use the features of the scientific computer language PASCAL–XSC to offer modules that can be combined by the reader to his/her individual needs. Our overriding concern is reliability – the *automatic verification of the result* a computer returns for a given problem. All algorithms we present are influenced by this central concern. We must point out that there is no relationship between our methods of *numerical result verification* and the methods of *program verification* to prove the correctness of an implementation for a given algorithm.

This book is the first to offer a general discussion on

- arithmetic and computational reliability,
- analytical mathematics and verification techniques,
- algorithms, and
- (most importantly) actual implementations in the form of working computer routines.

Our task has been to find the right balance among these ingredients for each topic. For some topics, we have placed a little more emphasis on the algorithms. For other topics, where the mathematical prerequisites are universally held, we have tended towards more in-depth discussion of the nature of the computational algorithms, or towards practical questions of implementation. For all topics, we present examples, exercises, and numerical results demonstrating the application of the routines presented.

The different chapters of this volume require different levels of knowledge in numerical analysis. Most numerical toolboxes have, after all, tools at varying levels of complexity. Chapters 2, 3, 4, 5, 6, and 10 are suitable for an advanced undergraduate course on numerical computation for science or engineering majors. Other chapters range from the level of a graduate course to that of a professional reference. An attractive feature of this approach is that you can use the book at increasing levels of sophistication as your experience grows. Even inexperienced readers can use our most advanced routines as black boxes. Having done so, these readers can go back and learn what secrets are inside.

The central theme in this book is that practical methods of numerical computation can be simultaneously efficient, clever, clear, and (most importantly) reliable.

We firmly reject the alternative viewpoint that such computational methods must necessarily be so obscure and complex as to be useful only in "black box" form where you have to believe in any calculated result.

This book introduces many computational verification techniques. We want to teach you to take apart these black boxes and to put them back together again, modifying them to suit your specific needs. We assume that you are mathematically literate, i.e. that you have the normal mathematical preparation associated with an undergraduate degree in a mathematical, computational, or physical science, or engineering, or economics, or a quantitative social science. We assume that you know how to program a computer and that you have some knowledge of scientific computation, numerical analysis, or numerical methods. We do not assume that you have any prior formal knowledge of numerical verification or any familiarity with interval analysis. The necessary concepts are introduced.

Volume 1 of *Numerical Toolbox for Verified Computing* provides algorithms and programs to solve basic numerical problems using automatic result verification techniques.

Part I contains two introductory chapters on the features of the scientific computer language PASCAL–XSC and on the basics and terminology of interval arithmetic. Within these chapters, the important correlation between the arithmetic capability and computational accuracy and mathematical fixed-point theory is also discussed.

Part II addresses one-dimensional problems: evaluation of polynomials and general arithmetic expressions, nonlinear root-finding, automatic differentiation, and optimization. Even though only one-dimensional problems treated in this part, the verification methods sometimes require multi-dimensional features like vector or matrix operations.

In Part III, we present routines to solve multi-dimensional problems such as linear and nonlinear systems of equations, linear and global optimization, and automatic differentiation for gradients, Hessians, and Jacobians.

Further volumes of *Numerical Toolbox for Verified Computing* are in preparation covering computational methods in the field of linear systems of equations for complex, interval, and complex interval coefficients, sparse linear systems, eigenvalue problems, matrix exponential, quadrature, automatic differentiation for Taylor series, initial value, boundary value and eigenvalue problems of ordinary differential equations, and integral equations. Editions of the program source code of this volume in the C++ computer language are also in preparation.

Some of the subjects that we cover in detail are not usually found in standard numerical analysis texts. Although this book is intended primarily as a reference text for anyone wishing to apply, modify, or develop routines to obtain mathematically certain and reliable results, it could also be used as a textbook for an advanced course in scientific computation with automatic result verification.

We express our appreciation to all our colleagues whose comments on our book were constructive and encouraging, and we thank our students for their help in testing our routines, modules, and programs. We are very grateful to Prof. Dr. George Corliss (Marquette University, Milwaukee, USA) who helped to polish the text and

the contents. His comments and advice based on his numerical and computational experience greatly improved the presentation of our tools for Verified Computing.

Karlsruhe, September 1993 The Authors

The computer programs in this book

are available in several machine-readable formats. To purchase diskettes in IBM-PC compatible format, use the order form at the end of the book. The source code is also available by anonymous ftp from ...

Table of Contents

Chapter 1

Introduction

This is a reference book for numerical methods with automatic result verification. The methods presented here are practical, reliable, and elegant. We provide theory, algorithmic descriptions, and implementations for methods to solve some basic numerical problems in a reliable way. Also, this book can help you learn how to develop such methods and how to proceed for other problems beyond the scope of the book.

We warn of potential computational problems of standard numerical routines, but we do not claim that our advice is infallible! Therefore, we offer our practical judgment on solving many basic numerical problems with automatic result verification. As you gain experience, you will form your own opinion of how reliable our advice is.

We presume that the reader is able to read computer programs in PASCAL and/or PASCAL–XSC (i.e. PASCAL eXtension for Scientific Computation), the language in which all the "tools" in this edition of *Numerical Toolbox for Verified Computing* are implemented. Its wide range of concepts for scientific computation makes PASCAL–XSC especially well suited as a specification language for programming with automatic result verification. We discuss the features of PASCAL–XSC in Chapter 2.

In the following sections, we first give some hints on the methodology and structure of this book. Then we explain the textual and algorithmic notation we use. We also describe the basic concepts and form of our implementations, and we make some comments on the computational environment. We conclude with a section on the motivation and the necessity of methods with numerical result verification.

1.1 Advice for Quick Readers

This book is organized in three parts:

1. the preliminaries,
2. the one-dimensional problems, and
3. the multi-dimensional problems.

The preliminaries give an overview on the extensions of the programming language PASCAL–XSC and on the properties and advantages of interval arithmetic. The reader with little experience with numerical verification and interval techniques should read the section on interval arithmetic with care and interest. Its introductory character shows what arithmetic improvements can do for the reliability of computed results.

We discuss only basic one- and multi-dimensional problems in parts II and III, respectively. We especially direct the attention of readers unfamiliar with numerical result verification to the following chapters:

- Linear Systems of Equations (Chapter 10),
- Evaluation of Polynomials (Chapter 4),
- Automatic Differentiation (Chapter 5), and
- Nonlinear Equations in One Variable (Chapter 6).

Some other chapters, e.g. optimization, require a little more background knowledge on numerical verification and interval techniques. These algorithms sometimes use the basic algorithms mentioned above (e.g. the linear system solver).

1.2 Structure of the Book

In order to make this *Numerical Toolbox* easy and efficient to use, we have chosen some general standards to keep the different chapters of this book uniform. This will help you find the important information on each topic, e.g. the theoretical background or the implementation of a desired algorithm, without necessarily reading the whole chapter.

Each chapter starts with an introduction to describe the numerical and mathematical problem and to motivate a strategy for its solution. A comprehensive discussion of the theoretical basis necessary to solve the problem and to verify the computational result is given in the section "*Theoretical Background*". Under the heading "*Algorithmic Description*", we present the complete algorithm in a very short, clear, and consistent way.

A very important part of each chapter is the section "*Implementation and Examples*" where an actual implementation for the topic discussed in the chapter is presented. As promised in the title, all routines are designed as "tools" that can be combined in several ways to satisfy your needs in solving a specific numerical problem. The routines are built of different modules so they may be integrated in an existing program, e.g. to overcome a numerical trap at a certain point of your own complex problem-solving algorithm. All algorithms presented in this volume are available on a floppy disk and via ftp (see Page vii) to help you use the verification algorithms immediately for testing or for integrating them into existing applications. To demonstrate the effectiveness of each algorithm, we present sample programs that solve practical problems. We point out the superiority of verification methods and precisely defined machine arithmetic over most standard numerical problem-solving strategies.

Most chapters include some "*Exercises*" to encourage you to study the behavior or to improve the algorithms we present. The section "*References and Further*

Reading" at the end of each chapter points out to some similar or more sophisticated and specialized algorithms. The appendix contains basic modules that are used by certain verification algorithms.

1.3 Typography

We use the following type faces for certain words, names, or paragraphs:

italics	emphasizes certain words in the text.
boldface	marks reserved words of PASCAL–XSC (e.g. **begin**, **module**) in the text or in excerpts of programs.
slanted	characterizes predefined identifiers of PASCAL–XSC (e.g. *integer*, *real*) and identifiers from programming examples when they appear in the body of the text.
`typewriter`	distinguishes listings and run-time results of programs, including some kind of pretty-printing to improve the readability (see Section 1.5).

References are always indicated as [*nr*], where the number *nr* corresponds to an entry in the bibliography.

1.4 Algorithmic Notation

The algorithms presented in Parts II and III of this book are specified in a pseudo-code similar to PASCAL code, including mathematical notation (cf. Chapter 3) wherever this is possible. Algorithm 1.1 is a simple example of our algorithmic notation.

Algorithm 1.1: DoSomething (x, y, z, Err) {Procedure}

1. $a := x + y$; $b := x - y$; {Initializations}
2. {Do something special}
   if $(a > 0)$ **and** $(b > 1)$ **then**
   $c := \sqrt{a}$; $d := \ln(a)$;
   $a := -c$; $b := -d$;
   else if $(b = 1)$ **then**
   return Err := "Error number 1 occurred!";
   else
   return Err := "Another error occurred!";
3. **return** $z := a \cdot b \cdot c/d$;

For clarity, the steps of the algorithms are numbered, and the single statements are separated by semicolons. We use indentation and numbering to mark blocks of statements and to avoid "**begin** – **end**" notation, whenever this is possible. Within the algorithmic steps, subalgorithms may be used in the form of procedures or functions

ProcedureName (*ArgumentList*)

Variable := FunctionName (*ArgumentList*).

The statements

return or **return** *ArgumentList*,

cause an immediate return from the current algorithm, where the second form returns new values for the arguments of *ArgumentList*. Loops have three different forms:

for *Index* := *InitialValue* **to** *FinalValue* **do** *Statement(s)*

repeat *Statement(s)* **until** *LogicalExpression*

while *LogicalExpression* **do** *Statement(s)*.

The statements

$\textbf{exit}_{Index\text{-loop}}$, $\textbf{exit}_{while\text{-loop}}$, and $\textbf{exit}_{repeat\text{-loop}}$

terminate a loop. The statement

$\textbf{next}_{Index}$

terminates the current iteration of the loop by transferring control to the end of the loop specified by *Index*. There the next iteration of the loop may be initiated, depending on the nature of the loop, its associated increments and tests. When using indexed expressions in mathematical notation, the loop is characterized by a sequence of indices enclosed in parentheses, e.g. $(k = 1, \ldots, n)$.

Conditional statements are

if *LogicalExpression* **then** *Statement(s)* **else** *Statement(s)*

with optional **else**-branch.

All comments are enclosed in braces.

1.5 Implementation

The authors feel very strongly that you should be able to count on the programs, modules, and code fragments in this book to compile and run as we claim. Therefore, we have compiled and run each of the code fragments (as parts of entire running programs), modules, and programs. Similarly, the results we give were actually produced by the example programs, except that we show both the printed output and the user-supplied input exactly as it will appear on your screen. The code and results have been included directly, without modification, in this manuscript by a pretty-print program, so you are seeing exactly the code that we ran. We include modules or programs in the text in this typographical style:

```
{ Short description of the global routines ... }
module dosome;

const
  NoError      = 0;
  ErrorOne     = 1;
  AnotherError = 2;

global function DoSomethingErrMsg( ErrCode: integer ) : string;
var
  Msg : string;
begin
  case ErrCode of
    NoError     : Msg := '';
    ErrorOne    : Msg := 'Error number 1 occurred';
    AnotherError: Msg := 'Another error occurred';
    else        : Msg := 'Code not defined';
  end;
  if (ErrCode <> NoError) then Msg := 'Error: ' + Msg + '!';
  DoSomethingErrMsg := Msg;
end;

global procedure DoSomething( x, y: real; var z: real; var Err: integer);
var
  a, b, c, d : real;
begin
  a := x + y;  b := x - y;  Err := NoError;      { Initializations }

  if (a > 0) and (b > 1) then                    { Do something special }
    begin
      c := sqrt(a);  d := ln(b);  a := -c;  b := -d;
    end
  else if (b = 1) then
    Err := ErrorOne
  else
    Err := AnotherError;

  if (Err = NoError) then  z := a * b * c / d;   { Else z is undefined }
end;

{ Module initialization }
begin
  { Nothing to initialize }
end.
```

We handle all errors and exceptional cases with a function *DoSomethingErrMsg*. It associates an error message with the error code returned by the routine *DoSomething*.

1.7.4 Scientific Computation with Automatic Result Verification

With these two arithmetic aids (optimal dot product and interval arithmetic), we are able to perform scientific computations with automatic result verification. We will illustrate this by two simple examples.

Example 1.3 We first address the question of whether a given real function defined by an arithmetic expression has zeros in a given interval $[x]$. This question cannot be answered with mathematical certainty if only floating-point arithmetic is available. One can evaluate the function say 1000 times in the interval $[x]$. If all computed function values turn out to be positive, then the probability is high that the function does not have any zeros in $[x]$. However, this conclusion is certainly not reliable. Because of roundoff errors, a positive result could be computed for a negative function value. The function could also descend to a negative value which was not detected due to the choice of the evaluation points. On the other hand, a single interval evaluation of the function may suffice to solve this problem with complete mathematical certainty. If the computed interval result does not contain zero, then the range of values also cannot contain zero, because the computed interval result is a superset of the range. As a consequence, we conclude that the function does not have any zeros in the interval $[x]$. An interval evaluation of the range of the function may fail to validate that the function has no root either because the function really *does* have a root or because of overestimation of the range. In many cases, however, a single interval function evaluation (costing the same as about two floating-point evaluations) provides a guarantee not available from hundreds, thousands, or millions of floating-point evaluations (see Figure 1.1).

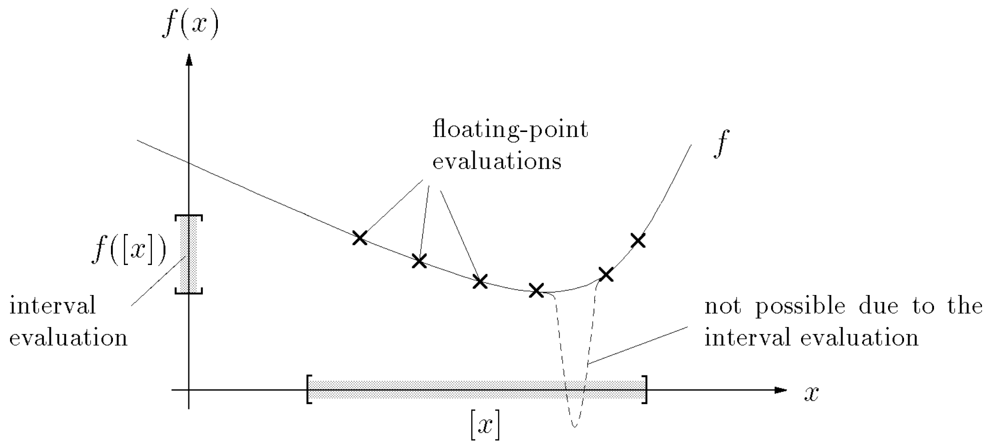


Figure 1.1: A single interval evaluation provides the guarantee that f does not descend to negative values (dashed graph)

Example 1.4 As a second example, we sketch a method by which one can verify the correctness of the computed solution for a linear system of equations $A \cdot x = b$. First, an approximation for the solution is computed, for example by Gaussian elimination. This approximation is expanded by augmenting it with a small number $\pm\varepsilon$ in each component direction, giving an n-dimensional cube $[x]$ with the computed approximation as its midpoint. This cube is just an interval vector consisting of intervals in each coordinate direction. Now, one transforms the system of equations in an appropriate way into a fixed-point form $x = B \cdot x + c$, where one naturally makes use of the fact that one has already computed a presumed approximate solution. For example, one might get B and c from the residual form of $A \cdot x = b$ as discussed in Chapter 10. Then the image $[y] := B \cdot [x] + c$ is computed. If $[y]$ is contained in $[x]$, which can be verified simply by comparison of endpoints, then $[y]$ contains a fixed-point of the equation $x = B \cdot x + c$ by Brouwer's fixed-point theorem (cf. Theorem 3.1). If $[y]$ is strictly contained in $[x]$, so that the endpoints do not touch, then the fixed-point is unique. That is, the computer program has verified computationally that the original matrix A of the system of equations is nonsingular. By iteration using the optimal dot product, the fixed-point can then be determined to arbitrary accuracy in practice. This example illustrates that validation ($Ax = b$ has a unique solution in $[y]$) is distinct from achieving high accuracy (iterating $[x] := B \cdot [x] + c$ until the width of the interval components of $[x]$ are small). In this way, the user obtains a mathematically certain and reliable result.

Examples 1.3 and 1.4 of numerical computation with automatic result verification described here leave open the question of what to do if the verification step fails. For rootfinding (Example 1.3), solving linear systems (Example 1.4), and many other problems, refined methods have been developed which allow valid assertions. A great advantage of automatic result verification is that the computer itself can quickly establish that the computation performed has not led to a correct and usable result. In this case, the program can choose an alternative algorithm or repeat the computation using higher precision.

Similar techniques of automatic result verification can be applied to many other algebraic problem areas, such as the solution of nonlinear systems of equations, the computation of zeros, the calculation of eigenvalues and eigenvectors of matrices, optimization problems, etc. In particular, valid and highly accurate evaluation of arithmetic expressions and functions on the computer is included. These routines also work for problems with uncertain or interval data.

The technique of enclosing function values with high accuracy is an essential aid for scientific computation. Newton's method for the computation of the roots of a system of equations is frequently unstable in a neighborhood of a root, for example when a root results from the mutual cancellation of positive and negative terms. In the immediate vicinity of a root, these terms are only approximately equal and cancellation occurs in ordinary floating-point arithmetic. However, in the case of insufficiently accurate function evaluation, Newton's method can easily overshoot its goal, possibly into the domain of attraction of another root. An iteration process which converges to a solution x in the real numbers thus may not converge to the value x in floating-point arithmetic. The iteration can swing over to an entirely different root. A concluding verification step is thus also indicated for iteration processes.

3.8 Principles of Numerical Verification

The automatic verification of numerical results is based on two major prerequisites:

1. the theory of interval arithmetic,
2. appropriate algorithms.

The easiest technique for computing verified numerical results is to replace any real or complex operation by its interval equivalent and then to perform the computations using interval arithmetic. This procedure leads to reliable, verified results. However, the diameter of the computed enclosures may be so wide as to be practically useless. We need more sophisticated methods which combine the benefits of interval arithmetic with a mechanism for refining already computed, but rough, enclosures.

In developing algorithms for validated enclosures, we must be careful what we compute an enclosure *of.* For example, if we take a program for the approximate solution of initial value problems in ordinary differential equations by a Runge-Kutta method and replace all floating-point computations by computations with floating-point intervals, we get an enclosure, as suggested in the preceding paragraph. However, we get an enclosure not of the solution to the differential equation, but of the Runge-Kutta approximation. We have enclosed the roundoff errors but not the truncation errors. An algorithm must enclose all sources of error that come into the computations, including conversion errors discussed in Section 3.7, in order to achieve validated enclosures of the desired answer.

A simple mechanism for the verified solution of point problems, i.e. problems with non-interval input data, is the principle of *iterative refinement.* After computing a first approximation, its error is enclosed using machine interval arithmetic. If the diameter of the error interval is less than a desired accuracy, then a verified enclosure of the solution is given by the sum of the approximation and the enclosure of its error. Otherwise, the approximation may be refined by adding the midpoint of the error interval and repeating the process. See also Stetter [82]. We use iterative refinement methods in this book for the accurate evaluation of polynomials (Chapter 4) and arithmetic expressions (Chapter 8).

Many algorithms for numerical verification are based on the application of well-known fixed-point theorems with respect to interval sets. As an example, we cite the following theorem (cf. [64]).

Theorem 3.1 (Brouwer's Fixed-Point Theorem) *Let $f : I\!R^n \to I\!R^n$ be a continuous mapping and $X \subseteq I\!R^n$ be a closed, convex, and bounded set. If $f(X) \subseteq X$, then f has at least one fixed-point x^* in X.*

Let $X = [x] \in IR^n$ be a machine interval vector. As a box in the n-dimensional space, $[x]$ satisfies the conditions of Brouwer's Fixed-Point Theorem. Suppose we can find a box with $f([x]) \subseteq [x]$. Then $[x]$ is proved to be an enclosure of at least one fixed-point x^* of f. The assertion remains valid if we replace f by its floating-point interval evaluation $f_\diamond$ because $f_\diamond([x]) \subseteq [x]$ implies $f([x]) \subseteq [x]$, since $f_\diamond([x])$ is a superset of $f([x])$.

Viewing the Brouwer Fixed-Point Theorem in an interval context motivates a template for the design of algorithms that compute a verified solution of numerical problems. First, find a fixed-point form $x = f(x)$ equivalent to the original problem. As an example, we refer to Newton's method for problems of finding a zero. Replace the generating function of the right-hand side by its floating-point interval extension $f_\diamond$. Start the following iteration scheme with some approximate solution $[x]^{(0)}$

$$[x]^{(k+1)} = f_\diamond([x]^{(k)}) \qquad \text{for } k = 0, 1, 2, \ldots \tag{3.1}$$

Stop the iteration if $[x]^{(k+1)} \subseteq [x]^{(k)}$ for some $k \geq 0$. If the iteration succeeded, then we have proven in the mathematical sense that the original problem has at least one solution x^* contained in $[x]^{(k)}$.

We distinguish *a priori* and *a posteriori* methods for the starting approximation of (3.25). For an *a priori* method, the starting approximation already includes the fixed-point desired. For this case, the iteration scheme (3.25) may be modified by intersecting successive iterates, that is,

$$[x]^{(k+1)} = f_\diamond([x]^{(k)}) \cap [x]^{(k)} \qquad \text{for } k = 0, 1, 2, \ldots$$

The iteration is halted if there are two successive iterates of same value or if a maximum number of iterations is exceeded. Figure 3.10 gives an illustration of how an *a priori* interval iteration works. Examples for *a priori* methods are found in Chapters 6 and 7.

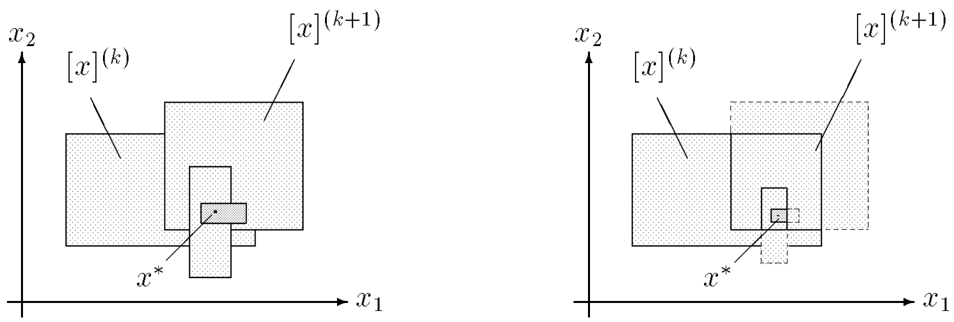


Figure 3.10: *A priori* method without (left picture) and with intersection

The starting approximation for an *a posteriori* method does not necessarily have to contain a desired fixed-point. Here, the hope is that successive iterates come closer and closer to a fixed-point and finally will enclose it. Of course, the better the starting approximation is, the faster the iteration will converge. However, in practice the iterates come closer and closer to the fixed-point, but they rarely catch it. A simple trick saves the method. Before starting a new iteration step, the actual iterate is slightly enlarged by means of the ε-inflation, as defined in Section 3.6. Thus, for an *a posteriori* method, the iteration scheme (3.25) is modified to

$$\left.\begin{aligned} [x]^{(k)} &= [x]^{(k)} \bowtie \varepsilon \\ [x]^{(k+1)} &= f_\diamond([x]^{(k)}) \end{aligned}\right\} \quad \text{for } k = 0, 1, 2, \ldots$$

Figure 3.11 illustrates the effects of an ε-inflation on an *a posteriori* method. Examples of *a posteriori* methods appear in Chapters 10 and 11.

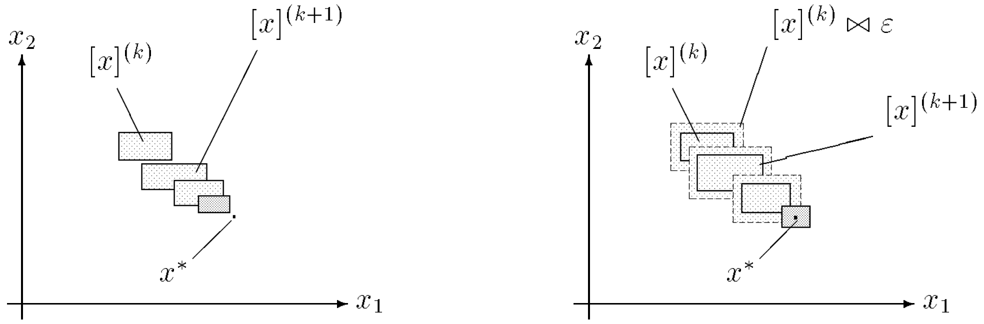


Figure 3.11: *A posteriori* method without (left picture) and with ε-inflation

Fixed-point methods may even be modified for some problems to prove the uniqueness of a fixed-point. For example, in Chapter 10 the uniqueness of the solution of a system of linear equations is proved by simply changing the stopping criterion of the fixed-point iteration to $[x]^{(k+1)} \overset{\circ}{\subset} [x]^{(k)}$.

So far, we have introduced the mathematical basics for verified numerical computing. We have defined the basic terms of real, complex, and extended interval arithmetics and have discussed aspects of an implementation on a computer. We also have discussed the problem of data conversion. Finally, we have introduced the principles of numerical verification by iterative refinement, *a priori*, and *a posteriori* methods. All these terms are used in the following chapters to describe algorithms that compute verified solutions of problems from numerical analysis.

Chapter 7

Global Optimization

We want to find the global minimum in an interval $[x]$ of a function f that may have many local minima. We want to compute the minimum value of f and the point(s) at which the minimum value is attained. This is a very difficult problem for classical methods because narrow, deep valleys may escape detection. In contrast, the interval method presented here evaluates f on a continuum of points, including those points that are not finitely representable, so valleys, no matter how narrow, are recognized with certainty. Further, interval techniques often can reject large regions in which the optimum can be guaranteed not to lie, so they can be faster overall than classical methods for many problems.

In classical numerical analysis, global optimization methods proceed by iteration starting from some approximate trial points. Classical optimization methods sample the objective function at only a finite number of points and cannot guarantee that the function does not have some unexpectedly small values in between these trial points.

In this chapter, we consider an algorithm based on the method of Hansen [25] for computing all solutions of the global unconstrained optimization problem. The algorithm computes enclosures for all global minimizers in a given interval and for the global minimum value of a twice continuously differentiable function.

7.1 Theoretical Background

We address the problem of finding all solutions x^* of

$$\min_{x \in [x]} f(x) \tag{7.1}$$

for a twice continuously differentiable function $f : I\!R \to I\!R$ and $[x] \in II\!R$. We wish to find the set of all global minimizers x^* and the global minimum value $f^* = f(x^*)$. The existence of a global minimizer is assured since f is a continuous function on a compact set, but need not be unique. In fact, the minimizers may include continua of points.

Our algorithm for problem (7.1) is based on the method of Hansen (see [25] or [28]) and the modifications of Ratz (see [73]). Starting from the initial interval $[x]$, our algorithm subdivides $[x]$ and stores the subintervals $[y] \subset [x]$ in a list L. Subintervals which are guaranteed not to contain a global minimizer of f are discarded from that list, while the other subintervals get subdivided again until the desired accuracy (relative diameter) of the intervals in the list is achieved. The key to the speed of this method is its use of several tests to discard subintervals in which no minimizer may occur:

- midpoint test,
- monotonicity test,
- concavity test, and
- interval Newton step.

In the following sections, we consider each of these methods in detail. We use the notation $\underline{f_y}$ as abbreviation for the lower interval bound of the interval function evaluation $[f_y] := f_{[]}([y])$ for $[y] \in I\!R$.

7.1.1 Midpoint Test

If we are able to determine an upper bound $\widetilde{f}$ for the global minimum value f^*, then we can delete all subintervals $[y]$ for which

$$\underline{f_y} > \widetilde{f} \geq f^*. \tag{7.2}$$

The midpoint test first determines or improves such an upper bound for f^*. Initially, let $\widetilde{f} = +\infty$. We choose an interval $[y]$ out of the list L which satisfies $\underline{f_y} \leq \underline{f_z}$ for all intervals $[z]$ in the list L. That is, $[y]$ has the smallest lower bound for the range of f. Hence, it is a likely candidate to contain a minimizer. Let $c = m([y])$ (or any other point in $[y]$), and compute $\widetilde{f} = \min\{f(c), \widetilde{f}\}$. Such an upper bound can also be computed on a computer when rounding errors occur: we compute $f_\diamond(c)$ and use the upper bound of the resulting interval as the possibly new value $\widetilde{f}$.

Now, with a possibly improved (decreased) value of $\widetilde{f}$, we can discard all intervals $[z]$ from the list L for which $\widetilde{f} < \underline{f_z}$. The midpoint test is relatively inexpensive, and it often allows us to discard from consideration large portions of the original interval $[x]$. Figure 7.1 illustrates this procedure, which deletes the intervals ...

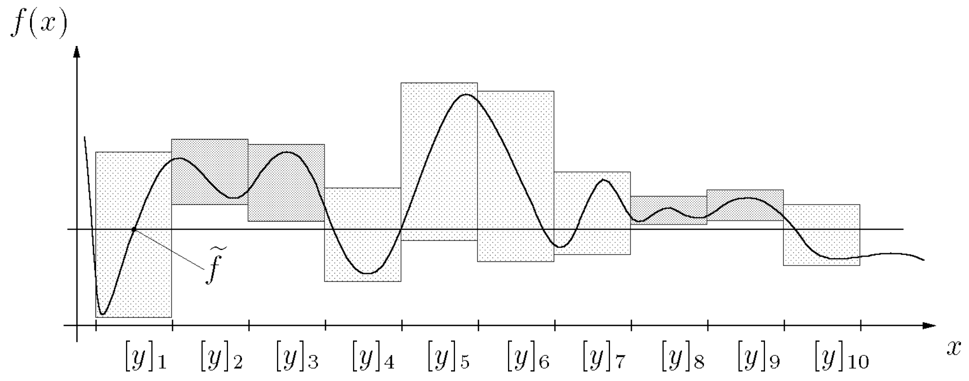


Figure 7.1: Midpoint test

Chapter 10

Linear Systems of Equations

Finding the solution of a linear system of equations is one of the basic problems in numerical algebra. We will develop a verification algorithm for square systems with full matrix based on a Newton-like method for an equivalent fixed-point problem.

10.1 Theoretical Background

10.1.1 A Newton-like Method

Let $Ax = b$ be a real system of equations with $A \in I\!R^{n \times n}$ and $b, x \in I\!R^n$. Finding a solution of the system $Ax = b$ is equivalent to finding a zero of $f(x) = Ax - b$. Hence, Newton's method gives the following fixed-point iteration scheme

$$x^{(k+1)} \;=\; x^{(k)} - A^{-1}(Ax^{(k)} - b), \quad k = 0, 1, \dots \;. \tag{10.1}$$

Here, $x^{(0)}$ is some arbitrary starting value. In general, the inverse of A is not exactly known. Thus instead of (10.1), we use the Newton-like iteration

$$x^{(k+1)} \;=\; x^{(k)} - R(Ax^{(k)} - b), \quad k = 0, 1, \dots, \tag{10.2}$$

where $R \approx A^{-1}$ is an approximate inverse of A.

Let us now replace the real iterates $x^{(k)}$ by interval vectors $[x]^{(k)} \in I\,I\!R^n$. If there exists an index k with $[x]^{(k+1)} \subseteq [x]^{(k)}$, then, by Brouwer's fixed-point theorem (see page 51), Equation 10.2 has at least one fixed-point $x \in [x]^{(k)}$. Supposed R is regular. Then this fixed-point is also a solution of $Ax = b$. However, if we consider the diameter of $[x]^{(k+1)}$, we get $d([x]^{(k+1)}) = d([x]^{(k)}) + d(R(A[x]^{(k)} - b)) \geq d([x]^{(k)})$. Thus in general, the subset relation will not be satisfied. For this reason, we modify the right-hand side of (10.2) to

$$x^{(k+1)} \;=\; Rb + (I - RA)x^{(k)}, \quad k = 0, 1, \dots, \tag{10.3}$$

where I denotes the $n \times n$ identity matrix. Rump [76] proved for Equation (10.3) that if there exists an index k with $[x]^{(k+1)} \overset{\circ}{\subset} [x]^{(k)}$, then the matrices R and A are regular, and there is a unique solution x of the system $Ax = b$ with $x \in [x]^{(k+1)}$. Moreover this result is valid for any matrix R. Note that the $\overset{\circ}{\subset}$ operator denotes the inclusion in the interior as defined in Section 3.1.

10.1.2 The Residual Iteration Scheme

It is a well-know numerical principle that an approximate solution $\widetilde{x}$ of $Ax = b$ may be improved by solving the system $Ay = d$, where $d = b - A\widetilde{x}$ is the residual of $A\widetilde{x}$. Since $y = A^{-1}(b - A\widetilde{x}) = x - \widetilde{x}$, the exact solution of $Ax = b$ is given by $x = \widetilde{x} + y$. Applying Equation (10.3) to the residual system leads to

$$y^{(k+1)} = \underbrace{R(b - A\widetilde{x})}_{=:\, z} + \underbrace{(I - RA)}_{=:\, C} y^{(k)}, \quad k = 0, 1, \dots . \tag{10.4}$$

According to Rump's results mentioned above, the residual equation $Ay = d$ has a unique solution $y \in [y]^{(k+1)}$ if we succeed in finding an index k satisfying $[y]^{(k+1)} \overset{\circ}{\subset} [y]^{(k)}$ for the corresponding interval iteration scheme. Moreover, since $y = x - \widetilde{x} \in [y]^{(k+1)}$, we then have a verified enclosure of the unique solution of $Ax = b$ given by $\widetilde{x} + [y]^{(k+1)}$.

These results remain valid if we replace the exact expressions for z and C in (10.4) by interval extensions. However, to avoid overestimation effects, it is highly recommended to evaluate $b - A\widetilde{x}$ and $I - RA$ without any intermediate rounding.

10.1.3 How to Compute the Approximate Inverse

We recall that the results summarized above are valid even for an arbitrary matrix R. However, it is a non-surprising empirical fact that the better R approximates the inverse of A, the faster the contained-in-the-interior relation for two successive iterates will be satisfied. We favor Crout's algorithm (cf. [83]) with partial pivoting for matrix inversion.

Let $A' = PA$ be the matrix which comes from A by row interchanges so that a factorization $A' = LU$ with a lower-triangular matrix L and an upper-triangular matrix U exists. If we normalize the diagonal entries of L to unity, and if we save the indices for the pivotal rows in a vector p, initialized by $p = (1, 2, \dots, n)^{\mathrm{T}}$, the decomposition procedure is as follows:

$$\left.\begin{array}{l}
v_k = a_{ki} - \displaystyle\sum_{j=1}^{i-1} l_{kj} a_{ji}, \quad k = i, \dots, n \\
|v_j| = \displaystyle\max_{i \le k \le n} |v_k| \\
\text{if } j \ne i\text{:} \quad p_i \leftrightarrow p_j, v_i \leftrightarrow v_j, a_i \leftrightarrow a_j, l_i \leftrightarrow l_j, u_i \leftrightarrow u_j \\
u_{ii} = v_i \\
\left.\begin{array}{l}
u_{ik} = a_{ik} - \displaystyle\sum_{j=1}^{i-1} l_{ij} u_{jk} \\
l_{ki} = \dfrac{v_k}{v_i}
\end{array}\right\} k = i+1, \dots, n
\end{array}\right\} i = 1, \dots, n. \tag{10.5}$$

Here, the double arrow ($\leftrightarrow$) indicates that the values of the specified elements or rows are swapped.

... more sophisticated inflation strategy. It is an empirical fact that the inner inclusion is satisfied nearly always after a few steps or never. Thus in Algorithm 10.4, the iteration is stopped after $p_{\max}$ steps.

Algorithm 10.1: VerificationStep $([x], [z], [C], \mathit{IsVerified})$ {Procedure}

1. {Initialization}
   $\varepsilon := 1000$; $p_{\max} := 3$; $p := 0$; $[x]^{(0)} := [z]$;
2. **repeat**
   $[x]^{(p)} := [x]^{(p)} \bowtie \varepsilon$; {$\varepsilon$-Inflation}
   $[x]^{(p+1)} := \Diamond([z] + [C] \cdot [x]^{(p)})$;
   $\mathit{IsVerified} := ([x]^{(p+1)} \overset{\circ}{\subset} [x]^{(p)})$;
   $p := p + 1$;
   until $\mathit{IsVerified}$ **or** $(p \geq p_{\max})$;
   $[x] := [x]^{(p)}$;
3. {Return enclosure $[x]$ and flag $\mathit{IsVerified}$}
   return $[x]$, $\mathit{IsVerified}$;

We now give the complete algorithm based on the Algorithms 10.2–10.4 for computing a verified enclosure of the solution of a linear system of equations. The procedure fails if the computation of an approximate inverse R fails or if the inclusion in the interior can not be established. A condition number estimate is returned by the variable *Cond*. We need a narrow enclosure $[z]$ of $z = R(b - A\widetilde{x})$ to start the verification step. Thus in Step 3, we first compute the residual $d = \square(b - A\widetilde{x})$. If the computation was afflicted with a roundoff error, an enclosure of this error is given by $[d] = \Diamond(b - A\widetilde{x} - d)$. So, we have $b - A\widetilde{x} \in d + [d]$ and $z \in \Diamond(Rd + R[d])$. In Step 4, we remark that $\widetilde{x}$ is an exact, but not necessarily unique, solution of the system if $[z] = 0$.

Algorithm 10.2: LinSolve $(A, b, [x], \mathit{Cond}, \mathit{Err})$ {Procedure}

1. {Computation of an approximate solution}
   MatInv (A, R, Err);
   if $(\mathit{Err} \neq$ "No error"$)$ **then**
   return $\mathit{Err} :=$ "Matrix is probably singular!"
   $\mathit{Cond} := \|A\|_\infty \cdot \|R\|_\infty$;
2. {Real residual iteration for an approximate solution}
   $k_{\max} := 10$; $k := 0$; $\widetilde{x}^{(0)} := R \cdot b$;
   repeat
   $d := \square(b - A \cdot \widetilde{x}^{(k)})$;
   $\widetilde{x}^{(k+1)} := \square(\widetilde{x}^{(k)} + R \cdot d)$;
   CheckForZeros $(\widetilde{x}^{(k)}, \widetilde{x}^{(k+1)})$;
   $\mathit{Success} :=$ Accurate $(\widetilde{x}^{(k)}, \widetilde{x}^{(k+1)})$;
   $k := k + 1$;

until *Success* **or** $(k \geq k_{\max})$;
$\widetilde{x} := \widetilde{x}^{(k)}$;

3. {Computation of enclosures $[C]$ and $[z]$ for $C = I - RA$ and $z = R(b - A\widetilde{x})$}
$[C] := \Diamond(I - R \cdot A)$;
$d := \Box(b - A \cdot \widetilde{x})$;
$[d] := \Diamond(b - A \cdot \widetilde{x} - d)$;
$[z] := \Diamond(R \cdot d + R \cdot [d])$;

4. {Verification step}
if $([z] = 0)$ **then**
$[x] := \widetilde{x}$; {Exact solution}
else
VerificationStep $([x], [z], [C], \mathit{IsVerified})$;
if (**not** *IsVerified*) **then**
Err := "Verification failed, the system is probably ill-conditioned!";
return *Cond*, *Err*;
else
$[x] := \widetilde{x} + [x]$; {Approximation plus correction interval}

5. {Return: Enclosure, condition number estimate, and error code}
return $[x]$, *Cond*, *Err*;

10.3 Implementation and Examples

10.3.1 PASCAL–XSC Program Code

We give the listings of the modules *matinv* for matrix inversion and *linsys* for solving a linear systems of equations. We emphasize that those parts of the algorithms of the previous section which are marked by $\Box(\ldots)$ and $\Diamond(\ldots)$ are implemented using accurate expressions. Error codes are passed by parameters of type *integer*. The error code 0 means that no error occurred.

10.3.1.1 Module matinv

The procedure *MatInv* is an implementation of Algorithm 10.1. It takes into account that the index range of the matrix parameters may start with lower indices different from one. The procedure checks whether the system is square and whether the input and the output matrices have the same shape. The function *MatInvErrMsg* may be used to get an explicit error message. ...

... of the parameters are compatible. The local procedures *CheckForZeros*, *Accurate*, and *VerificationStep* are implementations of the Algorithms 10.2–10.4. The module provides two global procedures *LinSolve* one with and the other without computing a condition number estimate for the input matrix. Both procedures call *LinSolveMain* with an appropriate list of parameters. The global function *LinSolveErrMsg* gives an explicit error message.

```
{----------------------------------------------------------------------------}
{ Purpose: Computation of a verified solution of a square linear system of   }
{    equations A*x = b with full real matrix A and real right-hand side b.   }
{ Method: Transformation of A*x = b to fixed-point form and applying an      }
{    interval residual iteration.                                            }
{ Global functions and procedures:                                           }
{   procedure LinSolve(...)      : To get a verified enclosure of the        }
{                                  solution, two versions                    }
{   function LinSolveErrMsg(...) : To get an error message text              }
{----------------------------------------------------------------------------}
module linsys;      { Linear system solving }
                    { ---    ---            }
use
  mv_ari,           { Matrix/vector arithmetic          }
  mvi_ari,          { Matrix/vector interval arithmetic }
  matinv;           { Matrix inversion                  }

{----------------------------------------------------------------------------}
{ Error codes used in this module.                                           }
{----------------------------------------------------------------------------}
const
  NoError      = 0;   { No error occurred.                         }
  NotSquare    = 1;   { System to be solved is not square.         }
  DimensionErr = 2;   { Dimensions of A*x = b are not compatible. }
  InvFailed    = 3;   { System is probably singular.               }
  VerifFailed  = 4;   { Verification failed, system is probably   }
                      { ill-conditioned.                           }

{----------------------------------------------------------------------------}
{ Error messages depending on the error code.                                }
{----------------------------------------------------------------------------}
global function LinSolveErrMsg ( Err : integer ) : string;
var
  Msg : string;
begin
  case Err of
    NoError     : Msg := '';
    NotSquare   : Msg := 'System to be solved is not square';
    DimensionErr: Msg := 'Dimensions of A*x = b are not compatible';
    InvFailed   : Msg := 'System is probably singular';
    VerifFailed : Msg := 'Verification failed, system is probably ' +
                         'ill-conditioned';
    else        : Msg := 'Code not defined';
  end;
  if (Err <> NoError) then Msg := 'Error: ' + Msg + '!';
  LinSolveErrMsg := Msg;
end;

{----------------------------------------------------------------------------}
{ Computes the absolute value of a real vector component by component.       }
{----------------------------------------------------------------------------}
function abs( v : rvector ) : rvector[lb(v)..ub(v)];
var
  ...
```