Efficient and Reliable Global Optimization

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of The Ohio State University

By

Anthony P. Leclerc, B.S., M.S.

* * * * *

The Ohio State University

1992

Dissertation Committee:

Ramon E. Moore

Judith D. Gardiner

Wayne E. Carlson

Approved by

Adviser

Department of Computer
and Information Science

To My Parents

Acknowledgements

I offer a sincere thanks to my advisor, Dr. Ramon Moore, for inspiring me, as a young graduate student, with the “interval bug”. I further thank professor Moore for his guidance, encouragement, and enthusiasm during my dissertation research. Your ideas, empathy, and friendship have indeed enriched my life.

For suggestions and comments regarding this document, I thank both Dr. Wayne Carlson and Dr. Judith Gardiner. I thank Eldon Hansen for sharing ideas and proposing problems pertaining to reliable interval global optimization. For input and technical assistance, I am grateful to Manas Mandal, Bob Manson, Steve Romig, Dave Ebert, Richard Fox, and Keith Boyer. For conversation, friendship, and the exchange of ideas, I thank Jeff Ely, a fellow “interval disciple’.

To the CIS department, I offer my gratitude for providing a delightful work environment. The graduate students and the staff have been helpful and encouraging throughout my 6 year graduate program. To those in the main office, Jane Grissom, Jeff Steinbeck, Tom Fletcher, and Marty Marlatt, I say thanks. I offer my thanks to Carl Phillips for being a friend and efficiently managing the CIS network of over 300 workstations necessary for my research.

A deep and sincere thanks to all of my family for their support and love. To my mother, for a lifetime of caring, patience, advice, and encouragement, I express my warmest thanks: I love you. To my father, for “making me think” and consistently being concerned with my total well being, I also express my most sincere thanks. I thank my brothers and sister for their unfailing patience, understanding and interest with regard to my graduate studies.

I offer my sincere appreciation to Clark Hultquist as a friend, tennis partner, fellow graduate student, intellectual, and roommate. To Magoo, I extend my appreciation for your genuine concern and support throughout my undergraduate and graduate endeavors. To T.D. Nicklas, I wish to say thank you for helping me obtain those critical scholarships during my first years as an undergraduate. To Dr. Gary Huckabay, I say thank you for encouraging me and guiding me into mathematics and computer science. To all the rest of my friends from OSU, Columbus, and Lawton, Oklahoma, I say thanks for getting me here. Finally, to Jesus Christ, I say thank you for blessing me with this Ph.D. and many other great and enjoyable things all of my life.

Vita

May 11, 1963	Born—Chalon-Sur-Marne, France
1986	B.S. Computer Science Cameron University Lawton, Oklahoma
1986-1987, 1988-1990	Graduate Teaching Associate Computer Science Department The Ohio State University Columbus, Ohio
1987-1988	Graduate Administrative Associate Computer Science Department The Ohio State University Columbus, Ohio
1988	M.S. Computer Science The Ohio State University Columbus, Ohio
1990-1991	Course Coordinator Computer Science Department The Ohio State University Columbus, Ohio
1992-present	Instructor Computer Science Department The Ohio State University Columbus, Ohio

Publications

R. E. Moore, E. R. Hansen, and A. P. Leclerc. “Rigorous Methods for Parallel Optimizatoin”. *Recent Advances in Global Optimization*, Princeton University Press, December 1991.

A. P. Leclerc. “Newton’s Method”. Technical Report OSU-CISRC-01/91-TR-04, The Ohio State University Computer and Information Science Research Center, January 1991.

Fields of Study

Major Field: Computer and Information Science

Studies in:

Numerical Analysis:	Prof. Ramon Moore
Computer Graphics:	Prof. Ed Tripp
Operating Systems:	Prof. Sandra Mamrak

Table of Contents

List of Tables

LIST OF FIGURES

Efficient and Reliable Global Optimization

By

Anthony P. Leclerc, Ph.D.

The Ohio State University, 1992

Ramon E. Moore, Adviser

For quite some time, it has been held that no numerical algorithm could guarantee having found a global solution to the general nonlinear global optimization problem. The reasoning was:

> The function to be minimized can only be sampled at a finite number of points. Therefore, there is no way of knowing whether the function dips to some smaller value between sampled points.

Although this argument is probably true using evaluations of functions at points, it is not true of methods which can produce asymptotically accurate lower bounds for the range of values of the function over compact sets.

Interval arithmetic, for example, provides asymptotically accurate upper and lower bounds on ranges of values of functions over continua [12, 11, 21, 22, 23, 24, 26, 32]. Coding interval arithmetic in C++, the author has designed an ideal bounding mechanism which is:

1. capable of producing reliable, “tight”, and asymptotically accurate bounds
2. efficient to compute
3. applicable to *any* programmable function
4. easy to generalize and automate
5. convenient for an unsophisticated user to utilize

Using this mechanism, rigorous algorithms are presented which produce a list of “boxes” enclosing the set of all global minimizers and an interval trapping the minimum value.

The most basic algorithm does not even use differentiability. Using interval Newton methods, monotonicity tests, and convexity tests, improvements in efficiency are achieved for differentiable problems. For further improvements in efficiency, the algorithms are parallelized. Numerical examples illustrating the techniques are given.

The parallelization task is accomplished by distributing identical processes over a network of workstations. Each process performs the interval global optimization algorithm but with a different subregion of the initial search space. Communication overhead is minimized in order to maximize the speedup. Issues of distributed initialization, load balancing, and global termination detection are addressed. Finally, an analysis of the speedup is determined.

CHAPTER I

INTRODUCTION

1.1 Description of Global Optimization

Global optimization is concerned with the determination of *global* optima (maxima or minima) of a function. Such problems occur frequently in numerous disciplines which model real world systems. For instance, a bond manager may wish to optimize the geometric mean of several indices, such as average coupon rate, average maturity, average yield to maturity, etc., which measure the performance of a bond portfolio [17]. An engineer, wishing to design an airplane wing consisting of composite materials, would like to determine what angles to bond the various plies of composites in order to maximize the strength of the wing [43]. A chemist desires a "best fit" of a photoelectron spectrum as the sum of Gaussian curves representing individual spectrums of various substances.

1.2 Importance of the Global Optimization Problem

The importance of the global optimization problem almost requires no explanation. For instance, the bond manager, optimizing a bond portfolio, will hopefully reap economic rewards from obtaining an optimal solution to the geometric mean function of the various bond performance indices. The engineer, by optimizing, will conceivably design a lighter, stronger, and safer composite wing. Finally, the chemist, seeking an optimal solution, can decompose a given substance into basic components more accurately.

Claude Jablon, Chief Scientist, Elf-Aquitaine, succinctly emphasizes the importance of global optimization problems [28]:

> Among the various subfields of scientific computing, optimization plays a major role: In a formal way, almost every mathematical problem can be recasted into an optimization problem. On a more practical level, design engineers are continuously looking for the most acceptable trade-offs – thus implicitly or explicitly solving complex optimization problems.

As an area of research, global optimization has become increasingly active. Several new international journals on the subject have recently been published or are being planned [8, 14]. In addition, existing journals are dedicating special issues to the subject of global optimization [9, 10].

1.3 Global Optimization Versus Local Optimization

As its importance alone may suggest, the global optimization problem is an extremely hard problem. Indeed, *no* efficient solutions exist which solve the general global optimization problem. The global optimization problem is much more difficult, for instance, than the related task of *local optimization* in which only the determination of local optima is desired. In local optimization, tests exist to determine if a point is a local optimum (assuming that the function to be optimized is twice differentiable). With such tests, a sequence of points converging to a local optimum can be constructed.

The relative difficulty of solving the global optimization problem on a computer as compared with the local optimization problem stems from the fact that, in general, the objective function may contain numerous local optima with corresponding function values varying significantly. Although an iterative algorithm for finding a local minimum (local method), such as Newton's method, may find one of these local minima efficiently, it is difficult to know whether this local solution is indeed a global solution.

One might suggest finding all of the local minima and choosing the smallest value(s) as the global minimum (minima). In general, however, the number of local minima may be quite large. Furthermore, due to the unpredictable global convergence behavior of local iterative methods (see [18]), one cannot determine whether all the local minima have been identified simply by varying the initial starting guesses.

1.4 Layout of this Dissertation

Chapter II is a short chapter providing a formal definition of the global optimization problem. The solvability of the global optimization is established in terms of an ϵ-global solution. Finally, the complexity of the global optimization is briefly discussed.

Chapter III considers methods which are capable of yielding ϵ-global solutions to the global optimization problem. Methods which do not calculate bounds for the range of a function over a compact set are dismissed from further investigation since they are susceptible to the indentation argument mentioned in chapter II. Given a technique for bounding the range of a function, algorithms for reliably solving the global optimization are discussed. Among bounding techniques, interval arithmetic is favored because of its ideal properties, including automatic control of roundoff error. Interval arithmetic is investigated, especially with regard to the author's implementation using C++. Finally, various interval global optimization algorithms are considered.

Chapter IV focuses on the efforts, some of which are based on joint work with R. E. Moore and E. R. Hansen, to make interval global optimization algorithms more efficient. In section 4.2, some simple-looking global optimization problems are considered. These problems are relatively difficult to solve, however, because the functions involved are not differentiable. Therefore, only simple (and relatively inefficient) procedures are applied. In section 4.3 other procedures are discussed which are applicable when the functions are differentiable. Finally, in section 4.4, parallel methods are considered. This section is based entirely on original work of the author, first presented in a talk at Princeton University [25], and expanded here.

Chapter V contains the results of numerical testing. The first problem, the Kowalik problem, was suggested by Eldon Hansen. At the time, the solution to this problem was the only test problem which had eluded Hansen and his interval software. The second problem is a "real world" chemistry problem which the author discovered and researched. This problem was used as the main test of the author's newly parallelized interval software. The photoelectron spectroscopy problem was executed, in parallel, on up to 40 workstations connected on an Ethernet network.

CHAPTER II

THE PROBLEM STATEMENT

2.1 Definition of Global Optimization

Formally, the *global optimization problem* is defined as finding

$$f_* = \min_{x \in X} f(x) \tag{2.1}$$

where $f : R^n \to R^1$ is a continuous real value *objective function* and $X \subset R^n$ is a compact feasible set. X is often succinctly called *the feasible region.* Since minimizing $f(x)$ is equivalent to maximizing $-f(x)$ this definition sufficiently includes the search for global minima as well as global maxima.

For future discussion, the set of all points for which the objective function possesses a global minimum value shall be called X_*. This is the set which contains all points, x_*, such that $f(x_*) = f_*$. This set is often called the set of *global minimizers.*

2.2 Complexity of the Global Optimization Problem

The global optimization problem is indeed hard. Rinnooy Kan and Timmer [16] claim that the global optimization problem as stated in equation 2.1 is unsolvable in a finite number of steps. Their argument is as follows:

> For any continuously differentiable function f, any point $\bar{x}$ and any neighborhood B of $\bar{x}$, there exists a function f' such that $f+f'$ is continuously differentiable, $f+f'$ equals f for all points outside B and the global minimum of $f+f'$ is $\bar{x}$. (($f+f'$) is an indentation of f.) Thus, for any point $\bar{x}$, one cannot guarantee that it is not the global minimum without evaluating the function in at least one point in every neighborhood B of $\bar{x}$. As B can be chosen arbitrarily small, it follows that any method designed to solve the global optimization problem would require an unbounded number of steps.

The *indentation argument* is certainly valid if one wishes to guarantee that an exact point, $\bar{x}$, is a global minimizer. Indeed, should the exact global minimizer be an irrational number, it is obviously impossible, in a finite number of steps, to numerically represent this solution. However, one *can*, in a finite amount of time, guarantee that $f(\bar{x})$ is within ϵ of the global minimum [23]. Such a solution is called a *ϵ-global minimizer*. Formally, an ϵ-global minimizer is defined as a point, $\bar{x}$, where

$$f(\bar{x}) < f_* + \epsilon.$$

While the global optimization problem is, in general, solvable (in the sense that one can come arbitrarily close to the optimum) in a finite amount of time, it is by no means easy to solve. Certain optimization problems have been shown to reside in the complexity class of NP-hard problems (see, for instance, [34, 27]). At present, *all* reliable, universally applicable algorithms to solve the global optimization problem have computational times which increase exponentially with the number of variables.

CHAPTER III

RELATED WORK

3.1 Introduction

This chapter considers methods which are capable of yielding ϵ-global solutions to the global optimization problem. Methods which do not calculate bounds for the range of a function over a compact set are dismissed from further investigation since they are susceptible to the indentation argument mentioned in chapter II. Given a technique for bounding the range of a function, algorithms for reliably solving the global optimization are discussed. Among bounding techniques, interval arithmetic is favored because of its ideal properties, including automatic control of roundoff error. Interval arithmetic is investigated, especially with regard to the author's implementation using C++. Finally, various interval global optimization algorithms are considered.

3.2 Methods to Solve the Global Optimization Problem

Historically, methods to solve the global optimization problem have been classified as either *stochastic* or *deterministic*. Stochastic methods evaluate the objective function, f, at randomly sampled points from the feasible region X (see equation 2.1). Deterministic methods, on the other hand, involve no elements of randomness.

Since this thesis is concerned with *reliable* solutions, all global optimization algorithms will alternatively be partitioned into the two classes: *reliable* and *unreliable*. Clearly all stochastic methods, including simulated annealing, clustering, and random searching, fall into the unreliable category. In fairness, however, efficiency is the strength of such methods. For now, large-scale problems may best be "solved" stochastically.

The class of deterministic algorithms, including branch and bound methods, covering methods, interval methods, tunneling, and enumerating, can be furthered partitioned into two categories:

- methods which compute function values at sampled points (*point methods*) and
- methods which compute function bounds over compact sets (*bounding methods*).

This division further separates reliable methods from unreliable. Point methods are inherently incapable of reliably solving the global optimization problem. On the other hand, bounding methods, if properly implemented and considerate of *roundoff error*, can produce rigorous global optimization solutions.

3.3 Point Methods and Their Weaknesses

Point methods suffer from the indentation argument mentioned in section 2.2. Furthermore, these methods, without additional information about the objective function, cannot even guarantee an ϵ-global minimizer because there is no indication as to the depth of the indentation between sampled points. In addition, since most computations are done with *fixed* precision floating point arithmetic, *even the best* of these methods can miss the detection of a global minimizer positioned between two consecutive machine numbers. The following example given by Moore [23] illustrates this event.

Consider finding the minimum of a function $f(x)$ for $1 \leq x \leq 2$. Suppose a subroutine for $f(x)$ has been written which receives the floating point number, x, and returns a floating point number for $f(x)$. Now, suppose that $f(x)$ is modified by adding to it the new function $g(x)$ defined as follows:

$$g(x) = \begin{cases} 0 & \text{for } x \leq x_1 \text{ or } x \geq x_2 \\ (x - x_1)(x - x_2)10^{20} & \text{for } x_1 < x < x_2 \end{cases}$$

where, say, $x_1 = 1.423561177$ and $x_2 = 1.423561178$.

Any point method limited to sampling values of $f(x)$ in floating point arithmetic with ten decimal digits or fewer of accuracy will find no change in $f(x)$. Thus the indentation or "*blip*" will go undetected.

3.4 Bounding Methods

Using various techniques, all bounding methods attempt to produce at least a lower bound for the range of values of f over a certain compact set. For the "blip" example given in section 3.3, should the input compact set (an interval) say $[x_L, x_U]$, include points in the range $[x_1, x_2]$, then the *lower bounding function*, F_L, would return a value less than or equal to the minimum value of the function at the “blip”.

For instance, if $f(x_1) = f(x_2) = 0$, the automatic bounding technique, *interval arithmetic*, with 10 digits of accuracy, would return the interval result $[-100, 0]$. With more precision and narrower $[x_L, x_U]$, interval arithmetic would converge from the outside upon the minimum value -25.0 at $(x_1+x_2)/2$. The “blip” will not escape detection.

All bounding methods, if properly implemented, are immune to the “blip” problem as discussed in section 3.3. Furthermore, if the lower bounding function is asymptotically accurate (see subsection 3.4.2), as is interval arithmetic, then the bounding method can yield an ϵ-global minimum.

Various techniques have been proposed for constructing lower bounding functions. Most of these techniques operate on compact convex sets which are *hyperrectangles* (n-dimensional rectangles). In addition, many of the techniques require assumptions about the objective function, f.

3.4.1 Some Techniques for Calculating Bounds

The Lipschitzian Approach

The *Lipschitzian* approach [29] is based on the assumption that a Lipschitz constant, L, exists such that

$$|f(x_1) - f(x_2)| \leq L||x_1 - x_2||.$$

If the value of f is known at some point, say, x_1, then a lower bound on the function value for all x between x_1 and x_2 can be determined by $f(x_1) - L||x - x_1||$.

The Linear Lower Bound Approach

The *linear lower bound* approach [2], generates linear lower bounds for a function, f, by decomposing f into simpler functions, such as monotonic convex functions. Once linear lower bounds for each of the simpler functions are found these bounds are recombined to obtain a linear lower bound for the original function. This approach, of course, assumes that f can be decomposed into simpler functions and has, as yet, only been demonstrated for one dimensional global optimization problems.

The Interval Approach

The *interval* approach automatically calculates lower and upper bounds using the principals of *interval arithmetic.* A "natural interval extension" (see [21]), F, for *any* programmable function, f, can easily be written. Such an interval function operates over hyperrectangles and returns interval results automatically bounding the range of the function over the input set.

3.4.2 A Simple Bounding Algorithm

Given a bounding function, F_L, with the ability to compute a lower bound for the range of f over a set, a simple exhaustive global search algorithm becomes evident.

One of the simplest of the bounding methods partitions the initial compact feasible set X into compact subsets, S_i^X. A lower bound on the function value, $F_L(S_i^X)$, over each subset S_i^X is then calculated. In addition, an upper bound on the global minimum thus far, U_{f_*}, is maintained. Any subset S_i^X where $F_L(S_i^X) > U_{f_*}$ is properly rejected as not containing a global minimum. This process of partitioning, bounding, and possibly rejecting is continued on successively generated subsets until some stopping criteria is met. The union of the remaining unrejected sets will contain the set of all global minimizers of f.

Clearly the effectiveness of this simple bounding method (and all bounding methods) is related to the ability of the lower bounding function to return a "tight" lower bound for the global minimum of the function over a given compact set. If the calculated lower bound is too crude, the ability to reject sets is hampered. On the other hand, if the lower bounding function returns the "tightest" (to the precision of the machine) lower bound for the global minimum of the function over a compact set, then the global optimization problem would be ϵ-solved. Most lower bounding functions, being only a piece of a larger global optimization algorithm, are not so accurate. However, in order to obtain an arbitrarily close global minimum, the lower bounding function should be *asymptotically accurate* in that the lower bound should approach the global minimum of f over the compact set S as the "volume" of S becomes smaller.

3.4.3 The General Bounding Algorithm

All bounding methods, such as branch and bound algorithms [16, 7], covering methods [7], linear lower bound methods [2], Lipschitzian methods [29], bisection methods [7, 30], and interval methods [12, 11, 23, 26, 32, 25], implement the following general algorithm:

1. **partition** the initial search space into smaller subregions,
2. **bound** the function (and possibly its derivatives) over the subregions, and
3. **reject** (by using the bounds calculated in step 2) those subregions which definitely cannot contain a global minimizer.

The union of the remaining *unrejected* subregions will contain all global minimizers.

The general bounding algorithm uses the *exhaustive* principle. It indirectly searches for a global minimum by exhaustively partioning and "cutting away" all of the feasible space, X, which definitely *cannot* contain a global solution.

The Partitioning and Bounding Phases

The manner in which the space is partitioned is mostly a function of the type of compact set over which the bounding function can compute. Each of the partitioned subregions will possess a certain geometry over which the bounding function must be able to compute. For instance, interval arithmetic computes over hyperrectangles. A partitioning step involves dividing a large hyperrectangle into smaller hyperrectangles.

In addition to being able to compute over the partitioned subregions, the geometric shape of the partitioned subregions should be able to *tessellate* (or cover) the entire feasible space, X. If the shapes cannot tessellate the space, then one cannot know, by exhaustively searching, whether a region of space was missed which might have contained a global minimizer.

Hypercircles, for instance, are a poor choice for partitioning a search space. Consider trying to tessellate a fish bowl with marbles. Between the marbles, empty space exists. This gap of space will not be *covered* by the bounding algorithm and therefore a global minimizer may be missed.

It is perhaps possible to use any shape and simply tessellate the space *with overlap*. However, it seems that such a crude collage of the initial search space will produce a less efficient algorithm since there is now a "bleeding" of boundaries between subregions. The ability to reject subregions appears to be less decisive. Furthermore, if one wishes to produce arbitrarily close global solutions, the amount of overlap should asymptotically diminish as the size of the subregion gets smaller.

With these considerations, hyperrectangles are usually the shape of choice by which to partition the feasible space. Hyperrectangles are simple shapes with which one can:

- partition easily
- tessellate the feasible region without overlap
- compute easily

If the feasible space is not itself a hyperrectangle, then an overlapping initial hyperrectangle is used as the initial search space. The equations which are used to characterize the true feasible space (constraint functions) are then additionally used in the rejection phase of the algorithm to eliminate those subregions lying in the initial hyperrectangle, but outside of the feasible region.

The Rejection Phase

The rejection phase of the above algorithm can include any of the following:

- rejection of subregions whose lower bound function value is greater than an upper bound on the global minimum known thus far
- rejection of subregions which are not in the feasible space (i.e. do not meet the constraints)
- if the objective function is differentiable
  - rejection of subregions not on the border of the initial search space for which $0 \notin g(x)$, where $g(x)$ is the *gradient* of the objective function, f
  - rejection of subregions for which the function is not convex anywhere within the subregion

3.5 Ideal Bounding Functions

Since the lower bounding function is at the *core* of bounding algorithms, such a bounding function should ideally possess the following characteristics in descending order of importance:

1. capable of producing reliable, “tight”, and asymptotically accurate bounds
2. efficient to compute
3. applicable to *any* programmable function (i.e. any function for which a computer program can be written)
4. easy to generalize and automate
5. convenient for an unsophisticated user to utilize

Once again, since the aim of this thesis is reliable global optimization, those remaining algorithms which are not capable of generating reliable global optimization solutions will be culled. A final partition separates those methods which use reliable lower bounding functions from those methods which do not.

3.6 Roundoff Error

Regardless of the accuracy of a lower bounding function in theory, such a function must yield *guaranteed* bounds in practice, on a computer. It is for this reason that any reliable lower bounding function must *at least* account for roundoff error.

3.6.1 Example of Roundoff Error

Some might contend that, today, roundoff error is not a critical issue because of the extended precision floating point numbers available on many computers. However, consider the following example of S. M. Rump [33] where the following innocuous-looking function

$$f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y) \tag{3.1}$$

was evaluated at $x = 77617$ and $y = 33096$. The FORTRAN program to evaluate this function at various precisions was compiled and executed on a SPARCstation SLC. The accuracy of the three precisions tested, single, double, and extended, were 6, 14, and 35 decimal digits, respectively. In order to test only the basic arithmetic operations, the powers on x and y were evaluated with multiplications rather than with the built-in library power function. The following results were obtained:

$$\begin{aligned} \text{(single precision)} \quad f &= 6.33825 \times 10^{29} \\ \text{(double precision)} \quad f &= 1.1726039400532 \\ \text{(extended precision)} \quad f &= 1.17260394005317863185883490452018\text{38} \end{aligned}$$

By observing these results, one might conclude that single precision result is wrong. One might further (and erroneously) conclude that the double precision result is accurate since it agrees to 13 digits with the extended precision result. Surprising to many, all three results are wrong even in the first digit! For that matter, the sign itself is incorrect. The exact result, obtained using the variable precision interval arithmetic of VPI [6] with about 40 decimal digits of accuracy, is "trapped" tightly in the following interval:

$$\begin{array}{lll} [& -0.827396059946821368141165095479816292005, & \\ & -0.827396059946821368141165095479816291986 &] \end{array}$$

The point to be made here is two-fold:

1. Roundoff error can seriously compromise the reliability of the results for *any* fixed precision floating point computation.

2. By simply observing the *one number* results at various precisions, *no* indication of the seriousness of the roundoff error is given.

3.6.2 Controlling Roundoff Error

It seems that roundoff error plagues *all* computations performed with fixed precision floating point arithmetic. How can one hope to control this last impediment to reliable global optimization? One tedious effort is to perform floating point error analysis. However, such analyses are extremely cumbersome and usually only applied to simple functions. In addition, floating point error analysis becomes more difficult with iterative algorithms in which roundoff error might be propagated from one iteration to the next.

Another attempt to "estimate" the accuracy of results from a floating point computation by using stochastic approaches such as the CESTAC method (or Permutation-Perturbation method) [40]. Such methods, however, are probabilistic in nature and cannot reliably guarantee the accuracy of the results.

To date, the author knows of only one bounding technique which is capable of computing asymptotically accurate bounds (on a computer) over a hyperrectangle. This technique is *interval arithmetic.*

3.7 Interval Arithmetic

Just *one* evaluation of a function using interval arithmetic provides upper and lower bounds on the range of values of the function over a *set* of values (a continuum) of the arguments. For instance, using interval arithmetic, one can test—on a computer—the truth of a relation such as:

$$^{1}f(x) = f(x_1, x_2, \ldots, x_n) \leq 0 \text{ for } \textit{all} \text{ points } x \in B$$

where B is any given initial hyperrectangle, or "box", defined as the following n-dimensional interval:

$$\begin{aligned} B &= \{x : a_i \leq x_i \leq b_i \text{ for } \textit{all } i = 1, 2, \ldots, n\}. \\ &= ([a_1, b_1], \ldots, [a_n, b_n]). \end{aligned}$$

One can do this for any programmable function using interval software, because a computer can find, simply by evaluating f with the argument B, numbers L and U such that

$$L \leq f(x) \leq U \text{ for } \textit{all } x \in B.$$

If $U \leq 0$, then it follows that $f(x) \leq 0$ for all $x \in B$.

[1]Such a relation is important in inequality constrained global optimization problems

The bounds L and U are generally not sharp (unless B is degenerate) even if infinite precision interval arithmetic is used. With fixed precision, these bounds are slightly less sharp because of rounding errors. In practice, interval arithmetic is done with *outward rounding* (the left end-point towards minus infinity and the right end-point towards plus infinity) so that the bounds are always correct although possibly not sharp.

Any given region can be covered by a set of boxes which can be subdivided so that each sub-box is as small as one likes. Because of this, there are two facts which make it possible to compute arbitrarily sharp bounds on the ranges of values of functions:

1. As a sub-box Y gets smaller, the over-estimation of $F(Y)$ diminishes and converges towards the actual range of values limited only by the number of digits carried (i.e. asymptotically accurate) [23, 31].

2. As Y gets smaller, if the number of digits carried is increased, one can come arbitrarily close to the exact range of values $F(Y)$ [23].

The range of values of f over a box B is contained in the union of the ranges of values of f over a covering of B by sub-boxes.

3.7.1 A Brief History of Interval Arithmetic

The earliest reported independent investigations into interval arithmetic were in papers by P. S. Dwyer [5], M. Warmus [42], T. Sunaga [38], and R. E. Moore [19]. However, it was the papers of Moore, especially his early monograph introducing interval arithmetic [20] in 1966, which actively commenced work in interval arithmetic. More than two dozen books and over 1000 journal articles and reports concerning interval analysis have been written since his monograph (see [12]). It is for this reason that R. E. Moore is know as the “father” of interval arithmetic.

3.7.2 Machine Interval Arithmetic

An interval is a closed bounded set of real numbers

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

Arithmetic with intervals is simply *arithmetic with inequalities*. For instance, if $a \leq x \leq b$ and $c \leq y \leq d$, then $a + c \leq x + y \leq b + d$. Thus, the addition of two intervals is defined by:

$$[a, b] + [c, d] \quad = \quad [a + c, b + d]$$

Likewise, using the properties of inequalities, the four basic interval operations are defined as follows:

$$\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d] \\
[a, b] - [c, d] &= [a - d, b - c] \\
[a, b] \times [c, d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\
[a, b] \div [c, d] &= [a, b] \times [1/d, 1/c] \text{ if } 0 \notin [c, d]
\end{aligned}$$

The implementation of interval arithmetic on a computer is easy. Since the endpoints a and b of a given interval $[a, b]$ may not be machine representable numbers, a is rounded to the largest machine number, say a_m, which is less than or equal to a, and b is rounded to the smallest machine number, say b_m, which is greater than or equal to b. This outward-rounded *machine interval*, $[a_m, b_m]$, contains $[a, b]$. Simply put, $[a, b] \subseteq [a_m, b_m]$. The basic principle of interval arithmetic is preserved in that the exact result is contained in the corresponding known machine interval, with roundoff error controlled.

Interval arithmetic is *inclusion monotonic.* This means that if

$$S_1 \subseteq X_1 \text{ and } S_2 \subseteq X_2$$

then

$$\begin{aligned} S_1 + S_2 &\subseteq X_1 + X_2, \\ S_1 - S_2 &\subseteq X_1 - X_2, \\ S_1 \times S_2 &\subseteq X_1 \times X_2, \\ S_1 \div S_2 &\subseteq X_1 \div X_2. \end{aligned}$$

By finite induction and the transitivity of the partial order relation, $\subseteq$, it follows that rational interval functions are inclusion monotonic, as are natural interval extensions of *all the standard functions used in computing* [21]. Furthermore, machine interval arithmetic with outward rounding is also inclusion monotonic. Given an expression in real variables, one can replace the real variables by corresponding interval variables and replace the real arithmetic operations and functions by the corresponding interval arithmetic operations and functions to obtain an interval function which is an inclusion monotonic *natural interval extension* of the real function.

Today, performing interval arithmetic on a computer with *outward rounding* is easy to achieve. The IEEE standard for binary floating point arithmetic specifies that the ability to round up or down as desired be available in the arithmetic hardware or software. The author has written a fixed precision interval arithmetic package in C and C++ (see [37]) which has been ported to the following systems:

- HP300
- SUN3 and SUN4
- IBM-PC

3.7.3 An Example of Machine Interval Arithmetic

Consider the following interval computation of:

$$x(u,t) = \frac{u^2 t}{u^2 + t^2 + 1}$$

where $u = [.1, .3]$ and $t = [.2, .6]$. Evaluating each sub-expression, one obtains:

$$\begin{aligned}
u^2 &= [.01, .09] \\
t^2 &= [.04, .36] \\
u^2 t &= [.002, .054] \\
u^2 + t^2 + 1 &= [1.05, 1.45] \\
\frac{u^2 t}{u^2 + t^2 + 1} &= \frac{[.002, .054]}{[1.05, 1.45]} = \left[\frac{.002}{1.45}, \frac{.054}{1.05}\right] \subseteq [.00137, .05143]
\end{aligned}$$

3.7.4 Interval Arithmetic Compared with the Lipschitzian Approach

For comparison, the above computation will be performed using a different bounding technique, namely the Lipschitzian approach. This approach requires center points, u_c and t_c, for u and t respectively. In addition, values for the "half-widths" of u and t are needed. These half-width values will be called du and dt, respectively.

Since $u \in [.1, .3]$, $u_c = .2$, $du = .1$ and since $t \in [.2, .6]$, $t_c = .4$, $dt = .2$. Using the Lipschitzian approach, one obtains

$$|x(u,t) - x(.2,.4)| \leq Bx_u|u - .2| + Bx_t|t - .4| \leq Bx_u(.1) + Bx_t(.2)$$

where Bx_u is an upper bound for $|x_u|$ over the box $u \in [.1, .3]$, $t \in [.2, .6]$ and Bx_t is an upper bound for $|x_t|$ over this same box. Here x_u and x_t are the partials of $x(u,t)$ with respect to u and t. Now

$$\begin{aligned} x_u &= \frac{(u^2+t^2+1)(u^2t)' - (u^2+t^2+1)'(u^2t)}{(u^2+t^2+1)^2} \\ &= \frac{(u^2+t^2+1)(2ut) - (2u)(u^2t)}{(u^2+t^2+1)^2} \\ &= \frac{2ut(t^2+1)}{(u^2+t^2+1)^2} \end{aligned}$$

$$x_t = \frac{(u^2+t^2+1)(u^2) - (2t)(u^2t)}{(u^2+t^2+1)^2} = \frac{u^2(u^2-t^2+1)}{(u^2+t^2+1)^2}$$

So

$$|x_u| = \frac{2|u|\,|t|(t^2+1)}{(u^2+t^2+1)^2} \leq \frac{2(.3)(.6)(.6^2+1)}{((.1)^2+(.2)^2+1)^2} = \frac{(.36)(1.36)}{(1.05)^2} \leq .44409 = B_{x_u}$$

and

$$|x_t| = \frac{u^2|u^2 - t^2 + 1|}{(u^2 + t^2 + 1)^2} \leq \frac{u^2(u^2 + t^2 + 1)}{(u^2 + t^2 + 1)^2} = \frac{u^2}{u^2 + t^2 + 1} \leq \frac{.09}{1.05} \leq .08571 = B_{x_t}$$

Thus

$$\begin{aligned} |x(u,t) - x(.2,.4)| &\leq (.44409)(.1) + (.08571)(.2) = .06155 \\ &= \text{half-width of the bounding box for } x \end{aligned}$$

and the width $= 2 * .06155 = .12310$. The resulting bounding interval is $x(u_c, t_c) \pm .06155 \approx .01333 \pm .06155 = [-.04822, .07488]$.

In this example, the interval calculation

- requires less mathematical sophistication from the user (the user need not know any calculus, but merely be able to perform simple interval calculations),
- assumes less about the nature of the function (i.e. the existence of bounded partial derivatives for the functions need not be assumed), and
- results in a "tighter" bounding box.

The author does not assert that the interval calculation always results in a tighter bound, but the other two advantages hold regardless of the function. It would seem that the only reason for possibly choosing the Lipschitzian approach would be if it produced a consistently tighter bounding box than the interval approach. In fact, given the considerable burden of asking the user to supply bounding functions for the rates, one should ultimately require the Lipschitzian approach to produce such a tighter bounding box that algorithms would accrue a non-trivial speedup to make it worth the bother.

Even though $[.00137, .05143]$ overestimates the actual range of values, it is *certainly* known that $\min x(u, t) \geq .00137$ and that $x(u, t)$ has no zeroes for any $u \in [.1, .3]$ and $t = [.2, .6]$. The Lipschitzian result, although accurate, was not sharp enough for us to make these two conclusions.

3.7.5 Interval Arithmetic in C++

In section 3.5 the characteristics of an ideal bounding function were mentioned in descending order of importance. From the example in subsection 3.7.3 interval arithmetic has been shown to be

- capable of producing reliable, "tight", and asymptotically accurate bounds, and
- efficient to compute.

Finally, using the language C++, interval arithmetic is additionally

- easy to generalize and automate, and
- convenient for an unsophisticated user to utilize.

Compare the floating point code in appendix A.1 with that of the interval code in appendix A.2. The only difference is the inclusion of "interval.h" instead of "math.h" and the variable declaration type of "interval" rather than "double". Yet, the interval code in appendix A.2 is performing the more complicated task of interval arithmetic and interval I/O.

The reason that the interval code is as simple to read and write as the floating point code is that the type *interval* has been defined as an *object class* within C++. In addition, the semantics of the operators, $+, -, *$, and / have been *overloaded* to perform the appropriate interval operations when expressions involving interval variables are evaluated. The input operator, “>>”, and the output operator, “<<” have also been overloaded so that the complicated task of interval I/O, which requires converting between base 10 and base 2 with correct outward rounding, is transparent to the user.

Appendix B shows the complete underlying *interval* class for all the C++ interval computations used in this thesis. The interval class declaration for the overloaded interval operation of division, for instance, can be found in appendix B.1. The corresponding code for such a declaration is listed in appendix B.2. The overloaded I/O routines, “>>” and “<<”, are also listed.

In appendix B.2 one can find the routines “rounddown()” and “roundup()” which are the essential outward rounding routines used in all the interval arithmetic functions and operators. The particular definition of these routines is tailored for a SUN4 workstation. The most complicated routines are the overloaded operators for division and multiplication, the non-monotonic sine and cosine routines, and the I/O routines, “>>” and “<<”.

3.8 Interval Optimization with No Rejection Tests

Moore [20] was possibly the first person to use interval arithmetic as a tool to compute the range of a function over a hyperrectangle. Clearly, if one can find the range of values of a function over a given set, then the maximum and minimum over the given set has been found. For this reason, Moore is credited with developing and implementing the first interval global optimization algorithms.

As Ratschek and Rokne [32] describe, Skelboe [35] later combined Moore's "range finding" interval methodology with the *branch and bound* principle. This principle has two characteristics:

- A uniform search of the feasible space is not performed. Instead, certain subregions (or *branches*) are searched sooner and more in depth than other subregions.
- A lower *bound* over a subregion must be computable.

The Moore-Skelboe algorithm first partitions the initial search space, X, into smaller subregions, S_i^X. The search for the global minimum, f_*, is performed by preferring those subregions, S_i^X, for which the lower bound on the function value, $F_L(S_i^X)$, is least. It is hoped that these selected boxes are more likely to contain a global minimizer, x_*.

The selection of the subregion, S_i^X, where $F_L(S_i^X)$ is the least can be quickly determined by using and maintaining a priority queue (or list). In a queue with m hyperrectangles, the subregion with the smallest $F_L(S_i^X)$, say S_m^X, will be at the front. The value of $F_L(S_m^X)$ is a *lower* bound on the value of the global minimum thus far and shall be called L_{f_*}.

The Moore-Skelboe algorithm does *not* employ any tests to eliminate those subregions which definitely do not contain any global minimizers. Such tests are called *rejection tests.* The algorithm simply continues until a suitable stopping criteria has been met. For instance, one may continue until either

$$\text{width}(F(S_m^X)) < \epsilon_{f1} \tag{3.2}$$

or

$$U_{f_*} - L_{f_*} < \epsilon_{f2} \tag{3.3}$$

where the upper bound on the global minimum, U_{f_*} is determined as the minimum of the upper bound on the function value, $F_U(S_i^X)$, over each subset S_i^X.

When using condition 3.2, the value, $\text{width}(F(S_m^X))$, is a bound for the absolute error when f_* is approximated by any point $x \in S_m^X$. If condition 3.3 is met, a bound for the absolute error, $|U_{f_*} - f_*|$ or $|L_{f_*} - f_*|$, is given by $U_{f_*} - L_{f_*}$. Relative error bounds for f_* can also be determined for both conditions 3.2 and 3.3. These bounds are, respectively,

$$\frac{\text{width}(F(S_m^X))}{\min(|F_L(S_m^X)|, |F_U(S_m^X)|)} \text{ if } 0 \in F(S_m^X)$$

and

$$\frac{U_{f_*} - L_{f_*}}{\min(|L_{f_*}|, |U_{f_*}|)} \text{ if } 0 \in [L_{f_*}, U_{f_*}]$$

Since, due to roundoff error, conditions 3.2 and 3.3 may not be met, additional stopping criteria should be integrated in order to prevent the possibility of an infinite loop. Two possible extra safety conditions are:

$$n \geq n_0$$

or

$$\mathrm{width}(S_m^X) < \epsilon$$

where n is the number of iterations of the algorithm performed and n_0 is the prescribed maximum number of iterations to perform.

3.9 Interval Optimization with a Midpoint Rejection Test

Two weaknesses with the Moore-Skelboe algorithm are as follows:

- No stopping criteria exist which are based on the "closeness" of an approximated global minimizer to a true global minimizer. In other words, when the Moore-Skelboe algorithm converges to a solution, S_m^X, one does not have a good measure of how close $x \in S_m^X$ is to a true global minimizer, x_*.
- No guarantee exists that the set of *all* global minimizers has been found. In general, there may be many global minimizers scattered about the initial search space.

An improvement to the Moore-Skelboe algorithm was made by Ichida and Fujii [15]. Their method incorporated a test which allowed one to determine if a particular subregion possibly contained a global minimizer. If it was determined that a particular subregion definitely did *not* contain a global minimizer, then the subregion was wisely rejected from further consideration.

The rejection test Ichida and Fujii employed is called the midpoint test. This test discards all subregions, S_i^X, where:

$$F_L(S_i^X) > U_{f_*}.$$

Here, U_{f_*} is an upper bound on the global minimum thus far determined by evaluating and maintaining the function at the *midpoint* of each hyperrectangle, S_i^X, and choosing the smallest value. Note that this definition of U_{f_*} is different from that of the one in the Moore-Skelboe algorithm.

The Ichida-Fujii algorithm produces a list of hyperrectangles. Furthermore, each hyperrectangle can contain a global minimizer. At any stage of the algorithm, the union of each S_i^X yields a proper superset, M, of X_*. No global minimizers have been lost.

A stopping criterion based on the error estimation of the approximated global minimizers to the true global minimizers now exists. The additional possible stopping criterion is as follows:

$$\max_{i=1,\ldots,n} \text{width}(S_i^X) < \epsilon_x \tag{3.4}$$

Using only stopping condition 3.4, one can now estimate the accuracy of an approximate global minimizer. Given $x \in S_i^X$, then

$$||x - x_*||_\infty \leq \max_{y \in S_i^X} ||x - y||_\infty$$

In addition, if it is known that each hyperrectangle, S_i^X, on the list contains a global minimizer, x_*, then one can more tightly bound the error as

$$||x - x_*||_\infty \leq \max_{i=1,\ldots,n} \text{width}(S_i^X)$$

3.10 Interval Optimization with Uniform Subdivisions

Ideally one would like a global optimization algorithm which, given enough precision, would always converge to the true set of global minimizers, X_*, as the number of iterations, n, approached infinity. The weakness of the Ichida-Fujii algorithm is that it can, in rare cases, converge to only a superset of X_*, and not upon X_* itself as $n \to \infty$. This exception occurs if there exists a subregion, S_k^X, *not* at the front of the queue in which

$$f_* = F_L(S_k^X).$$

This subregion will very possibly never move to the front of the queue, and therefore never be further partitioned. Furthermore, S_k^X will never be rejected because the condition $F_L(S_k^X) > U_{f_*}$ will never be true. Since M is the union of each S_i^X, M will never equal X_*.

It was Hansen who first suggested a reordering of the list in order to guarantee that with enough precision and iterations, M can be made to converge arbitrarily tightly upon X_*. He orders the list either with respect to the widths of the hyperrectangles or to the *age* (how long a particular hyperrectangle has been in the list) of the hyperrectangles. Either variant produces a uniform subdivision of all the hyperrectangles which have not been rejected.

With enough precision, Hansen's algorithm always converges to the true set of global minimizers, X_*, as the number of iterations, n, approaches infinity (see [26]). Efficiency aside, this is *the* ideal global optimization algorithm. Unlike all point methods, which are inherently incapable of ever completely solving the general global optimization problem, Hansen's algorithm is capable of solving *any* programmable global optimization problem given enough resources.

CHAPTER IV

TOWARDS EFFICIENT INTERVAL GLOBAL OPTIMIZATION

4.1 Introduction

This chapter focuses on the efforts, some of which are based on joint work with R. E. Moore and E. R. Hansen, to make interval global optimization algorithms more efficient. In section 4.2, some simple-looking global optimization problems are considered. These problems are relatively difficult to solve, however, because the functions involved are not differentiable. Therefore, only simple (and relatively inefficient) procedures are applied. In section 4.3 other procedures are discussed which are applicable when the functions are differentiable. Finally, in section 4.4, parallel methods are considered. This section is based entirely on original work of the author, first presented in a talk at Princeton University [25], and expanded here.

4.2 A Basic Interval Global Optimization Algorithm for Programmable Functions

In this section, a simple algorithm is presented for optimization problems of the form:

$$\min_{x \in B} f(x)$$

subject to

$$p_i(x) \le 0, i = 1, \ldots, k. \tag{4.1}$$

Here B is any given initial hyperrectangle, or "box", defined as the following n-dimensional interval:

$$\begin{aligned} B &= \{x : a_i \le x_i \le b_i \text{ for } \textit{all } i = 1, 2, \ldots, n\}. \\ &= ([a_1, b_1], \ldots, [a_n, b_n]). \end{aligned}$$

An arbitrarily tight bound is sought, within the box B, for the set X_* of all global minimizers x_* lying in the feasible region defined by the inequality constraints (4.1). Likewise, an arbitrarily tight bound on the global minimum value, f_*, of the given objective function, f, is also desired. That is, a bound for X_* and f_* is sought such that

$$f(x_*) = f_* \text{ and } f_* \le f(x) \text{ for all } x \in B \text{ satisfying (4.1).}$$

The functions f, p_1, ..., p_k are assumed to be programmable, but not necessarily differentiable throughout this section. For extensions of the algorithm to unbounded feasible regions and to problems involving equality constraints, see [32].

The basic algorithm will find a list of small boxes whose union *contains* the set X_* of all feasible global minimizers x_*. The algorithm terminates when the maximum box *width* (defined as the maximum edge length over all the coordinate directions) of all boxes in the list is less than a prescribed tolerance ϵ_x. The algorithm also finds lower and upper bounds on the minimum value $f_* = f(x_*)$.

The algorithm proceeds by *rejecting* parts of the initial box B which *cannot* contain a global minimizer, leaving a list of sub-boxes (of B) whose union still contains the set of all global minimizers of $f(x)$.

The methods for rejecting parts of a box and for finding lower and upper bounds on the minimum value of $f(x)$ for feasible points x are now described.

4.2.1 The Feasibility Rejection Test

Let X be a sub-box of B. A point in B can be represented as a degenerate box. One can evaluate (in interval arithmetic) the constraint functions $p_1(X), \ldots, p_k(X)$ and make the following test. X is said to be *certainly feasible* if $p_i(X) \leq 0$ for all $i = 1, \ldots, k$. If X is certainly feasible, then every point $x \in X$ is feasible. This is guaranteed despite the presence of rounding errors because of the outward rounding used in the computer implementation of interval arithmetic.

- X is *certainly infeasible* (i.e. it contains *no* feasible points) if, for some $i = 1, \ldots, k$, $p_i(X) > 0$. X may be rejected from further consideration.

4.2.2 The Midpoint Rejection Test

Let mX be the midpoint (or any other point) of a sub-box X of B. If mX is certainly feasible (see (4.2.1)), then the objective function is evaluated at mX to obtain an interval $f(mX) = [L_{F_{mX}}, U_{F_{mX}}]$. It is certainly the case that $U_{F_{mX}} \geq f_*$, that is, $U_{F_{mX}}$ is an upper bound on the minimum value of $f(x)$ over the feasible region. If f is evaluated over another sub-box Y of B yielding $f(Y) = [L_{F_Y}, U_{F_Y}]$, then the following test can be made. Let U_{F_*} be the smallest $U_{F_{mX}}$ yet found.

- If $L_{F_Y} > U_{F_*}$ then Y cannot contain a feasible global minimizer in B, so Y can be rejected from further consideration.

Using these tests, a very simple algorithm for global optimization with inequality constraints can be formulated. The algorithm is valid whether or not the functions involved are differentiable.

The list of boxes in the algorithm to follow is technically a *queue*. Elements are added to the end of the queue and removed from the front of the queue. Every time a box is removed , it is bisected along the coordinate direction of maximum width. Each half is then tested. If it cannot be rejected, then it is placed at the end of the queue.

As a result, the widest box remaining is always the first one. If it is narrower than ϵ_x then so are all the rest. At any stage of the process, the remaining boxes on the list contain all the feasible global minimum points.

4.2.3 The Algorithm

```
1. Input the initial box, $B$, and box width tolerance, $\epsilon_x$

2. Add $(B, L_{F_B})$ to the queue and update $U_{F_*}$

3.

   loop:

       Remove the first box, $X$, on the queue

       Bisect $X = X_1 \cup X_2$ along the coordinate direction of maximum width

       Reject $X_1$ or queue $(X_1, L_{F_{X_1}})$ at end of the queue

           If $X_1$ is queued, then update $U_{F_*}$ if $mX_1$ is feasible

       Reject $X_2$ or queue $(X_2, L_{F_{X_2}})$ at end of the queue

           If $X_2$ is queued, then update $U_{F_*}$ if $mX_2$ is feasible

       If the first box on the queue has width $< \epsilon_x$, then go to step 4

       Otherwise go to the beginning of step 3

4.

   Main program end:

       Graph (optionally print) the remaining boxes on the queue

       Print $L_{F_*} = \min L_{F_X}$ over all the boxes $X$ on the queue

       Print the current $U_{F_*}$
```

Some remarks are in order. Lots of other information can, of course, be displayed on the screen as the computation proceeds. For instance, the bisection and rejection steps can be graphically displayed as they take place along any two coordinate directions that one wishes. The number of function evaluations and the total elapsed real time during the computation can also be monitored.

Upon termination, $L_{F_*} \le f_* \le U_{F_*}$. The union of boxes in the list will certainly contain all the feasible global minimizers of $f \in B$. U_{F_*} is "updated" by replacing it with any smaller $U_{F_{mX}}$ found (see "midpoint test" (4.2.2)).

Suppose that a sub-box Y of B contains a local (but non-global) minimizer. Y is rejected if $L_{F_Y} > U_{F_*}$ (see "midpoint test" (4.2.2)). The smaller Y is, the closer the lower bound L_{F_Y} is to being exact. Therefore, the smaller ϵ_x is, the more likely that a local minimum will be rejected. Compare Figures 1 and 2.

4.2.4 An Illustration of the Algorithm

As an example of how the algorithm works, consider the following non-differentiable optimization problem [1] in which the following function is to be minimized:

$$f(x) = (|x_1^2 + x_2^2 - 1| + .001)|x_1^2 + x_2^2 - 0.25|$$

subject to

$$\begin{aligned} p(x) \quad &= \max\{ \quad (1 - \max\{|x_1|/0.6, |x_2|/0.25\}), \\ &\qquad\qquad (1 - \max\{|x_1|/0.25, |x_2 - 0.4|/0.3\})\} \le 0 \end{aligned}$$

and x in the initial box $B = ([-1.2, 1.2], [-1.2, 1.2])$, (that is: $-1.2 \le x_1 \le 1.2$ and $-1.2 \le x_2 \le 1.2$).

[1] This problem was suggested by Devin Moore.

The results obtained depend on a number of factors, including the number of digits carried in the arithmetic and the final box-width tolerance, ϵ_x.

Using interval arithmetic with outward rounding at the 11th decimal digit, the following results (carried out on an Apple MacIntosh with all the programming written in MS-BASIC) were obtained.

For $\epsilon_x = 0.2$, there were 74 boxes in the final list (see Figure 1). For $\epsilon_x = 0.1$, there were 28 boxes in the final list (see Figure 2).

For this problem there are three disconnected continua of global optimizers consisting of arcs, on the circle $x_1^2 + x_2^2 = 0.25$, which lie outside the union of two rectangles cutting through the circle. The entire unit circle, $x_1^2 + x_2^2 = 1$, consists of local minimizers. By merely reducing the final box width tolerance, ϵ_x, the local (but not the global) minimizers are eliminated. Compare Figures 1 and 2.

It is clear that standard (non-interval) optimization methods using only function evaluations at sample points will not be able to solve problems of this type in the same sense as is done here. The interval "solution" consists of a list of boxes whose union contains the set of all global minimizers. Other points are also contained, of course; but one can come as close as one pleases to the actual set of minimizers by doing enough computing and by carrying enough digits in the interval arithmetic. See Figure 3.

As alternative stopping criteria one might use, for instance:

- Stop if $U_{F_*} - L_{F_*} < \epsilon_f$, or
- Stop if $U_{F_*} - L_{F_*} < \epsilon_f$ *and* the maximum box width $< \epsilon_x$.

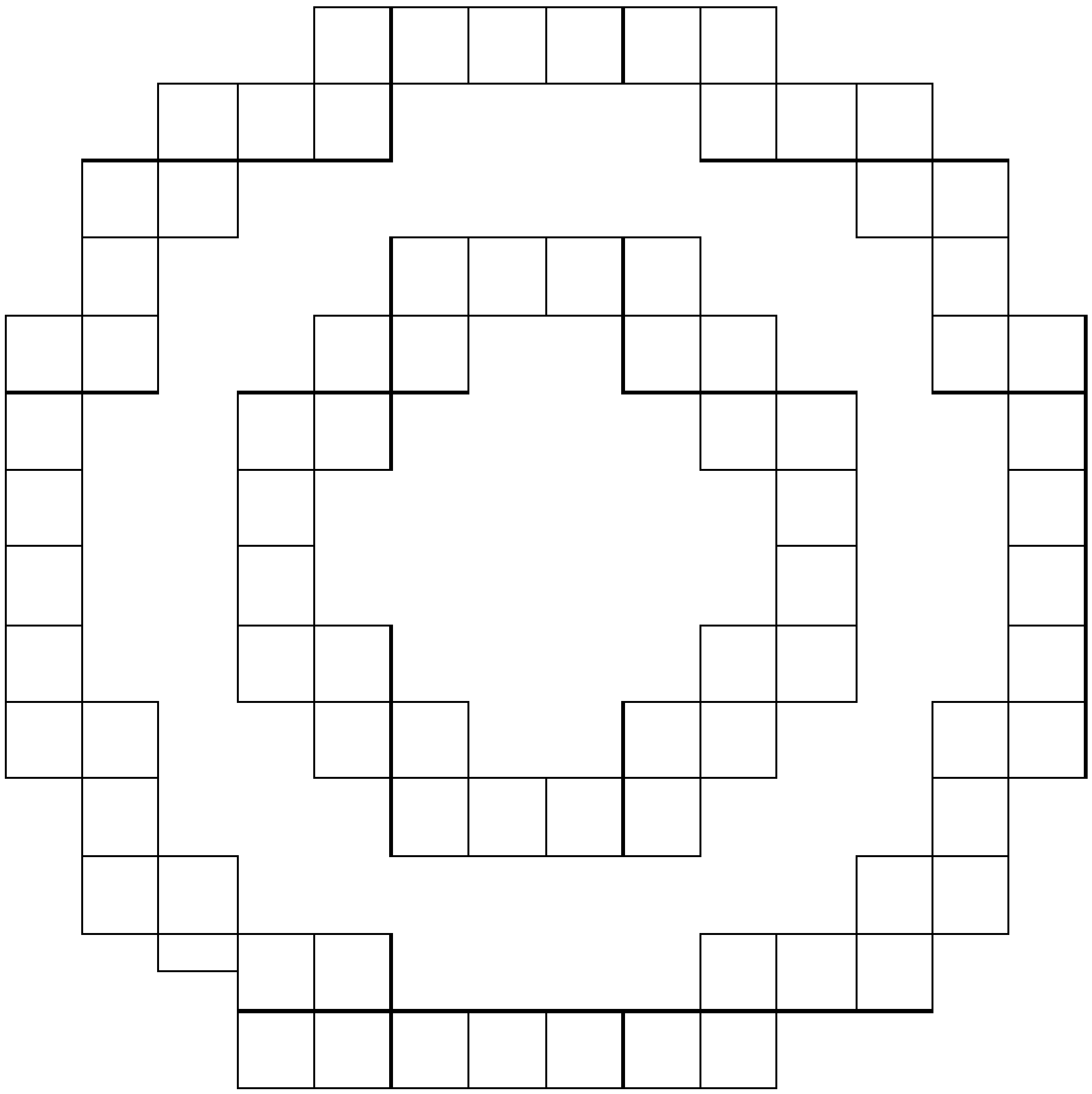

Figure 1: **Smiley Face with** $\epsilon_x = 0.2$

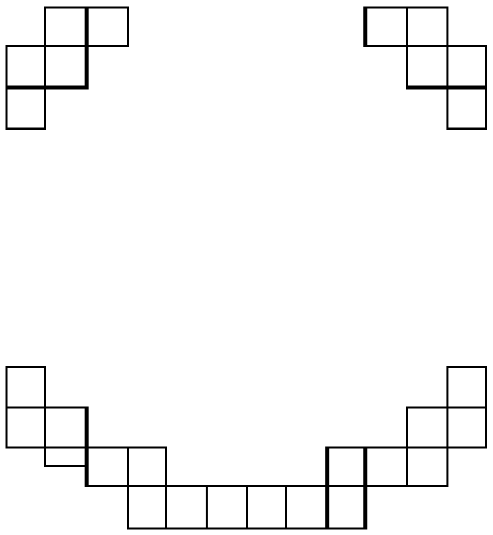

Figure 2: **Smiley Face with** $\epsilon_x = 0.1$

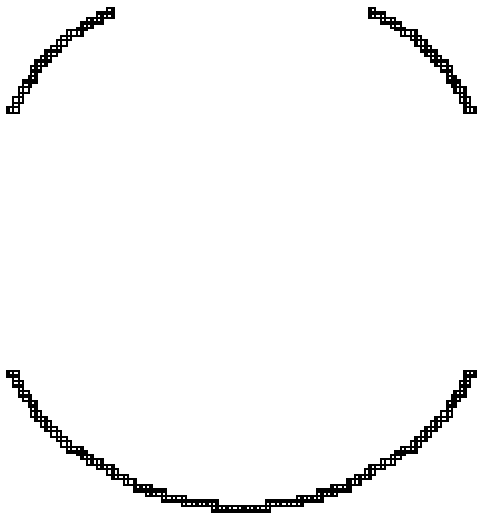

Figure 3: **Smiley Face with** $\epsilon_x = 0.0125$

However, it seems necessary to scan the existing list for the minimum lower bound L_{F_X} (see 4.2.3) each time such a test is applied, so that this would require some additional time.

For a discussion of the "complexity" of the type of algorithm presented in this section, see [32]. An accurate a-priori estimate of computing time seems difficult, even for the two-dimensional example given above, as Table 1 shows. The initial box B, the stopping tolerance ϵ_x on the maximum width of boxes in the final list, a count of the number of objective function evaluations n and the actual computing time, t, in seconds (in the slow, interpretive language MS-BASIC on the MacIntosh) is listed. In each case, the final list of boxes contained the global minimizers in B, but not the local minimizers on the unit circle.

Table 1: Timing Results for the Smiley Face Problem

B	ϵ_x	n	t
$([-1.2, 1.2], [-1.2, 1.2])$	0.1	898	337
$([-1.2, 1.2], [-1.2, 1.2])$	0.05	1143	445
$([0, 1.1], [0, 1.1])$	0.025	444	183
$([0, 1], [0, 1])$	0.025	943	346
$([0, 1], [0, 1])$	0.0125	1137	417
$([0, 2], [0, 2])$	0.025	1020	392
$([0, 10], [0, 10])$	0.025	629	308
$([0, 100], [0, 100])$	0.025	546	201
$([0, 10^6], [0, 10^6])$	0.025	798	298

An explanation of these timing results is difficult. They depend on the details of the rejection process. In particular, note that the cases involving the initial box $([0, 1], [0, 1])$, in the example at hand, take longer than some of the much larger initial boxes.

It is likely that this happened, for this example, because the unit circle of local minimizers is tangent to the edges of the initial box $([0,1],[0,1])$. A similar thing happens for the initial box $([0,2],[0,2])$ after the first couple of bisections. This type of curious behavior has been observed by the authors for other examples using quite different interval algorithms.

It is remarkable that the much larger initial boxes $([0,100],[0,100])$ and even $([0,10^6],[0,10^6])$ run faster than $([0,1],[0,1])$. This is because, upon bisection, these boxes never get cut down to any sub-box with edges tangent to the circle of local minimizers. Thus, an accurate a-priori complexity analysis for this algorithm must take into account the geometry of the solution set relative to the boundary of the feasible region. If this is not impossible, it is perhaps a research problem for the future.

More to the point, many improvements in efficiency are possible, particularly for differentiable problems, as is shown in the next section.

4.3 Differentiable Optimization Problems

When the objective function and the constraint functions are differentiable, one can use more efficient methods such as those discussed in this section.

If the objective function is differentiable, one can make use of local monotonicity. One can also make more efficient use of the upper bound U_{F_*} on the globally minimum value f_* of $f(x)$. When the objective function is twice differentiable, one can make use of local convexity, and one can use interval Newton methods. See subsection 4.3.4.

If the constraints are also continuously differentiable, then one can apply an interval Newton method to solve the Kuhn-Tucker or the John conditions which must be satisfied at a solution point.

For simplicity in discussing the procedures of this section, it will be assumed that the constraint functions $p_i(x)$ are continuously differentiable and the objective function $f(x)$ is twice continuously differentiable.

4.3.1 The Monotonicity Test

Consider the case in which a box B is certainly strictly feasible. Suppose the gradient g is evaluated over a sub-box X of B. If $0 \notin g_i(X)$ for some $i = 1, \ldots, n$, then the gradient is not zero in X. Therefore the global minimum cannot occur in X, and X can be rejected.

4.3.2 A Nonconvexity Test

Again consider a certainly strictly feasible box B and consider a sub-box X of B. If a global solution point, x_* occurs in B, then f must be convex in some neighborhood of x_*. In other words, the Hessian of $f(x)$ must be non-negative definite (positive semi-definite) at x_*. If it can be shown that the Hessian is *not* positive semi-definite anywhere in X, then X can be rejected.

The diagonal elements of the Hessian are given by

$$h_{ii}(x) = \partial^2 f(x)/\partial x^2, i = 1, \ldots, n.$$

One necessary condition for the Hessian to be positive semi-definite is that its diagonal elements be non-negative. Suppose h_{ii} is evaluated over X, using interval arithmetic, and that $h_{ii}(X) < 0$ for some i. X can be rejected. Other necessary conditions for non-convexity could be checked, but here only this simplest one is used.

4.3.3 Using the Upper Bound

In the basic algorithm described in section 4.2, the use of an upper bound U_{F_*} on the globally minimum value f_* of $f(x)$ was considered. It was pointed out that a box X could be rejected if $f(X) > U_{F_*}$. This bound can be used in a more sophisticated way. Even if $f(X) \not> U_{F_*}$, it may still be possible to reject a sub-box X' of X for which $f(X') > U_{F_*}$.

f can be expanded in the form

$$f(y) \in f(x) + (y - x)^T g(X).$$

See, for example [21]. If

$$f(x) + (y - x)^T g(X) > U_{F_*} \tag{4.2}$$

for all $y \in X'$, then $f(y) > U_{F_*}$ for all $y \in X'$; therefore, X' can be rejected.

One can solve the inequality (4.2) for y to determine such a sub-box X'. See [11] for details. One can also use a second order expansion of f in the same way. Again, see [11] for details.

4.3.4 An Interval Newton Method

The most effective procedure for use in solving differentiable global optimization problems is the interval Newton method.

Consider a vector function $g(x)$ which might be the gradient of f. Suppose one seeks the zero(s) of g in a box X. Let J denote the Jacobian of g and let x be a point in X. For any point y in X, the expansion

$$g(y) \in g(x) + J(X)(y - x)$$

holds [21]. If y is a zero of g , then y is in the solution set of

$$g(x) + J(X)(y - x) = 0. \tag{4.3}$$

One can find a box Y containing the set of solution points y of (4.3). X is then replaced by $X \cap Y$. Note that this intersection could be empty. If so, X is simply rejected. See [13] for details.

If X is certainly strictly feasible, then any minimum of $f(x)$ in X is at a stationary point. Therefore, $g(x) = 0$ at such a point, where $g(x)$ is the gradient of $f(x)$. In such a case, the interval Newton method to solve $g(x) = 0$ in X can be applied.

If X is not certainly strictly feasible, then a global minimum may not occur where $g(x) = 0$. However, it will occur where the John or Kuhn-Tucker conditions are satisfied. In this case, one can apply the interval Newton method to solve the set of equations expressing the John or Kuhn-Tucker conditions.

For the problem given by (4.1), these conditions are

$$g(x) + \sum_i u_i \nabla p_i(x) = 0$$

$$u_i p_i(x) = 0, i = 1, \ldots, k$$

where u_i is a Lagrange multiplier. See, for example [32].

4.4 Parallelization

A basic interval global optimization algorithm for possibly non-differentiable functions was given in section 4.2. In section 4.3 a list of more "powerful" box and sub-box deletion tests were given which could be incorporated when the objective function and the constraint functions were differentiable.

One can further improve the efficiency of the algorithm given in section 4.2, or augmented variations of this algorithm which use the tests given in section 4.3, by parallelizing. The parallelization of a sequential algorithm is not an obvious or even unique task for most problems. In particular this is true for the global optimization algorithms mentioned in sections 4.2 and 4.3. For what follows, a differentiation among the possible variations of these algorithms will not be made. The interval global optimization algorithms to be parallelized will simply be referred to, collectively, as "the algorithm".

In order to better qualify the parallelization intent, the terms *fine grain parallelism* and *coarse grain parallelism* will be used. Fine grain parallelism refers to the parallelism in an algorithm at the instruction level. For example, consider the addition of two intervals, $A = [al, ar]$ and $B = [bl, br]$ to obtain $C = [cl, cr] = [al + bl, ar + br]$. One could implement this simultaneously on two processors, one of which calculates $cl = al + bl$ while the other performs $cr = ar + br$.

The *speedup* of this parallel interval addition over the sequential interval addition is at most 2. Furthermore, this maximum speedup can only be realized in hardware, or perhaps on *tightly-coupled* parallel machines such as the Butterfly, Hypercube, or Transputer network, in which communication or shared-memory overhead is minimal. In a similar vein, the other basic operations $(-, *, /)$ could also be fine grain parallelized.

If arbitrary precision intervals are considered, such as those used in VPI [6], one can gain much greater speedups over the sequentially coded version by implementing the basic arithmetic operations as vector operations on a vector processor such as the CRAY Y-MP 8/64. In this thesis, however, fixed precision intervals are used. Furthermore, speedups on the order of N, where N is the number of processors, are desired.

Since in the Computer Science Department of The Ohio State University there are more than 300 SPARC station SLCs on a distributed Ethernet network available for the author to use, fine grain parallelism is not considered in this thesis, but instead *coarse grain parallelism* is investigated in which a different *partition* of the initial input data is *mapped* onto various processors.

4.4.1 One Possible Parallel Implementation

One possible parallel implementation would designate one processor as the queue processor, P_Q. P_Q would be responsible for maintaining the *global queue.* All the other processors would queue/dequeue boxes by consulting P_Q and then perform the rest of the algorithm in parallel.

Whenever an improved upper bound U_{F_*} on the global minimum (see *midpoint test* in section 4.2) is discovered by any processor, this *new* U_{F_*} would be broadcast to all other processors so that all could make sharper midpoint deletion tests. Note that at any given instant in time, some of the processors might be performing a midpoint test with an *older* U_{F_*}. This does not affect the correctness of the algorithm, but rather in the worst case, the box for which the test is being applied will not be deleted, but instead bisected. When each bisected half later comes to the foreground of the queue, the *newer* U_{F_*} will be available.

This parallel version possesses at least two short-comings. First, as the total number of processors involved in the computation increases, so does the demand for services from P_Q. Quickly P_Q will become a *bottleneck* with all other processors waiting unreasonable amounts of time in order to access the global queue. The second handicap is that the size of the global queue is limited to the amount of memory available on a single processor. Based upon considerable experience with the sequential algorithm, this is too constraining for real world problems. Therefore, a coarse grain distributed parallel global optimization algorithm in which each processor maintains its own *local queue* is considered.

4.4.2 The Distributed Parallel Algorithm

The distributed parallel algorithm has three main steps:

1. Initialize/Startup all processors
2. Perform "the algorithm" in parallel
    - Dynamic partitioning and load balancing
    - Broadcasting *new* U_{F_*}
3. Terminate all processors
    - Detect global termination
    - Compute final solution list

The steps are defined and discussed in the succeeding sections. Before doing so, the pair of terms *partitioning* and *mapping* are defined in the context of the parallel program. Partitioning refers to the manner in which the input data is divided-up among each of the processors. Mapping is concerned with the particular feasible assignment (with respect to the processor interconnection topology) of processor to process which minimizes communication costs.

A distributed network environment is a fully connected multiprocessor system. Furthermore, the interprocessor communication time is virtually homogeneous. Therefore, in the succeeding algorithm description, the reader can assume that any process can be mapped to any processor, and the mapping issue will not be addressed further. The parallel algorithm is now described.

4.4.3 Initialize/Startup All Processors

Observing the initial step for the basic algorithm in section 4.2, 4 steps are noted:

1. Input initial box, B
2. Input initial box width tolerance, ϵ_x
3. Queue the tuple, (B, L_{F_B}), on the *box queue*
4. Update U_{F_*}

These first 4 steps are performed only on the main processor, namely P_0. Next, P_0 will attempt to spawn $N-1$ process copies of itself on $N-1$ remote processors, $P_i, 0 < i < N$. P_0 will then wait until it has received from each P_i a *local state message*, *LSM*, indicating the status of the attempted spawn (many errors can occur when attempting to spawn a remote process on a distributed network). Each of the *LSM*s are compiled, along with the sending processor's unique identification number, domain name, and Ethernet address, into a *global state message*, *GSM*. After all $N-1$ *LSM*s have been received, P_0 sends a copy of the *GSM* to all P_is. All processors now have the necessary information to communicate with any other *living* processor involved in the parallel computation.

4.4.4 Perform "The Algorithm" in Parallel

Once step 4.4.3 above has been completed, only P_0 has a box on the box queue. How do the other $N-1$ processors proceed? This brings us to the issue of *dynamic partitioning* and *load balancing.*

Dynamic Partitioning and Load Balancing

Whenever any processor, P_j, has an empty box queue, it begins sending *box request messages*, *BRM*s, to a random $P_i, i \neq j$. If there are boxes available on P_i's box queue, then P_i sends P_j a *box message*, *BM*, containing half of its queued boxes, but no more than *NUMBOXES* (sending arbitrarily large messages is undesirable).

Otherwise, P_i sends back a short *no boxes available message*, *NBM*, indicating that it has no boxes available. If P_j receives a *NBM*, it then sends requests to processors, $P_{(i+1)modN}$, $P_{(i+2)modN}$, $P_{(i+3)modN}, \cdots, P_{(i+k)modN}$ until it receives a *BM* or until $k = N-1$ (see section 4.4.5).

This partitioning scheme is dynamic and demand driven. The hope is that by sending half of the workload to each box requesting processor, the work load (number of boxes) among all processors can be balanced.

Broadcasting the New U_{F_*}

As each processor executes "the algorithm" in parallel, eventually (assuming there exists a point, $x \in B$ (the initial input box) such that $f(x) < L_{F_B}$) an improved upper bound U_{F_*} on the global minimum will be discovered by a given processor, P_j. At this point, P_j will send this *new* U_{F_*}, NU_{F_*}, to all other processors $P_i, i \neq j$. When a given P_i receives this NU_{F_*} it compares it with its local U_{F_*}. If $NU_{F_*} < U_{F_*}$, P_i updates U_{F_*}. Otherwise, P_i must have received a lower NU_{F_*} from some other processor or calculated a lower U_{F_*} itself during the time it took to receive P_j's NU_{F_*}. In this case, P_i's U_{F_*} is not updated.

4.4.5 Termination

With the sequential version of "the algorithm", it was guaranteed that if the first box on the box queue had width less than ϵ_x, then so did all the other remaining queued boxes. However, in the parallel case, if the first queued box on the box queue of given processor, P_i, has width less than ϵ_x, then this does *not* necessarily imply that all the remaining boxes on the $N - 1$ other processors' box queues will have width less than ϵ_x.

Indeed, P_i may very well only have found a local minimum. What is P_i to do in this case? If P_i simply prints its output as described in section 4.2 and then terminates, then an uninteresting local solution very likely will be outputted, and moreover, a valuable worker processor will be lost.

The solution, for the moment, is to maintain a second queue, called the *possible solution queue*, *PSQ*, on every processor. Now, if the width of the first queued box on P_i's box queue is less than ϵ_x, then all of the boxes on P_i's box queue are placed on *PSQ*. P_i then behaves as in 4.4.4 for a processor with no queued boxes. Furthermore, whenever P_i determines a *new* U_{F_*}, it checks all boxes on *PSQ* and discards those boxes which fail the midpoint test (see section 4.2) using the *new* U_{F_*}. Global termination now becomes a question of detecting when *every* processors' box queue is empty. For the moment, such a state is detected with a simple centralized algorithm. A distributed algorithm using either a ring [3] or a tree [39] would be more efficient and fault tolerant.

Detect Global Termination

If a given P_i does not receive a *BM* after sending $N - 1$ *BRM*s, P_i then sends P_0 a *possible global termination message*, *PGTM*. P_i then waits for either a *BM* or a *terminate message*, *TM*, from P_0. If P_0 receives a *PGTM* and has boxes on its box queue, then P_0 simply sends P_i a *BM*. If P_0 receives a *PGTM* while it has no boxes on its box queue, then P_0 logs P_i's *PGTM*. When P_0 receives $N-1$ *PGTM*s, P_0 sends a *TM* to all other processors. Additionally, if P_0 is sending *BRM*s and receives a *BM*, P_0 must send *BM*s to all processors for which a *PGTM* was logged.

Compute Final Solution List

When a processor, P_i, receives a TM, it prints out all boxes on its PSQ and terminates. P_0 does the same as soon as it detects global termination and has sent $N-1$ TMs. The final solution list is obtained by combining the output from each terminated processor. As in the sequential version, the union of all the boxes on the final solution list will contain the set of all global minimizers.

CHAPTER V

RESULTS

Examples for the methods described in chapter IV are discussed in this chapter. All examples were run on one or more SUN4 workstations connected on an Ethernet network. In addition, all examples utilized the midpoint, monotonicity, and Newton tests described in sections 4.2 and 4.3. Termination was effected when the width of each remaining box containing a solution was less than 10^{-6}.

5.1 Kowalik Problem

For the first example, the Kowalik problem mentioned by Walster and Hansen [41] was solved. For this unconstrained global optimization problem the following function was to be minimized: $f(x_1, x_2, x_3, x_4)$ defined as

$$f(x_1, x_2, x_3, x_4) = \sum_{i=1}^{11}\left(a_i - x_1\frac{b_i^2 + b_i x_2}{b_i^2 + b_i x_3 + x_4}\right)^2$$

with the constants defined in Table 2.

The correct solution as reported by Walster and Hansen is as follows:

$$x_1^* = [0.1928334529823, 0.1928334529827]$$

Table 2: Kowalik Data

i	a_i	$1/b_i$
1	0.1957	0.25
2	0.1947	0.5
3	0.1735	1
4	0.1600	2
5	0.0844	4
6	0.0627	6
7	0.0456	8
8	0.0342	10
9	0.0323	12
10	0.0235	14
11	0.0246	16

$$x_2^* = [0.190836238780, 0.190836238785]$$

$$x_3^* = [0.123117296277, 0.123117296279]$$

$$x_4^* = [0.135765989980, 0.135765989983]$$

$$f^* = [0.00030748598779, 0.00030748598781]$$

Given an initial input box of $[-0.2892, 0.2893]$ in all dimensions and running on 1 processor, the program reported the following results:

```
convex hull of boxes on the queue:


x1 = [0.19283345267383470, 0.19283345332189572]

x2 = [0.19083623185103013, 0.19083624496089555]
```

```
x3 = [0.12311729477860661, 0.12311729761465269]

x4 = [0.13576598850717606, 0.13576599113717366]


f(hull): [0.00030748584424075611, 0.00030748613137046295]

max width of 1 box on the queue = 1.31099e-08


performed 545490 function evaluations.

performed 492923 Jacobian evaluations.

performed 200627 Hessian evaluations.


performed  51263 midpoint deletions.

performed  88999 monotonicity deletions.

performed  49741 Newton deletions.

performed  34635 Newton reductions.


computation time = 4057067 milliseconds > 1 hour
```

5.2 Photoelectron Spectroscopy Problem

The second example concerns a *real world* problem which arises in the field of chemistry. More specifically, chemists performing *photoelectron spectroscopy* collide photons with atoms or molecules. These collisions result in the ejection of photoelectrons [4, 1, 36]. A *photoelectron spectrum*, which is a plot of the number of photoelectrons ejected as a function of the kinetic energy of the photoelectron, is produced from monitoring these collisions. A typical spectrum consists of a number of overlapping peaks of various shapes and intensities. The chemist desires to resolve the individual peaks.

One method for isolating each peak attempts to "fit" the spectrum as the sum of *peak functions*. Peak functions are functions of variables which convey information regarding the peak's *position*, *intensity*, *width*, *function type*, and *tail characteristics*. Various types of functions have been used for this purpose, but the most common are Gaussian and/or Lorentzian.

For a specific test problem, a spectral curve as the sum of two Gaussian functions (see Figure 4) was arbitrarily constructed. The function definition is

$$\begin{aligned} x_i &= 4.0 + 0.1(i+1),\ i = 1, 2, \ldots, n \\ y_i &= a_1 e^{-[\frac{x_i - u_1}{s_1}]^2} + a_2 e^{-[\frac{x_i - u_2}{s_2}]^2} \end{aligned}$$

with the constants defined in Table 3.

Table 3: Photoelectron Spectroscopy Data

$a_1 = 130.89$	$a_2 = 52.6$
$u_1 = 6.73$	$u_2 = 9.342$
$s_1 = 1.2$	$s_2 = 0.97$

An attempt to "fit" this curve by recovering a_1, a_2, u_1, u_2, s_1, and s_2 was made. Given $n = 81, (x_i, y_i)$, and the initial input box, B, defined in Table 4, the task was

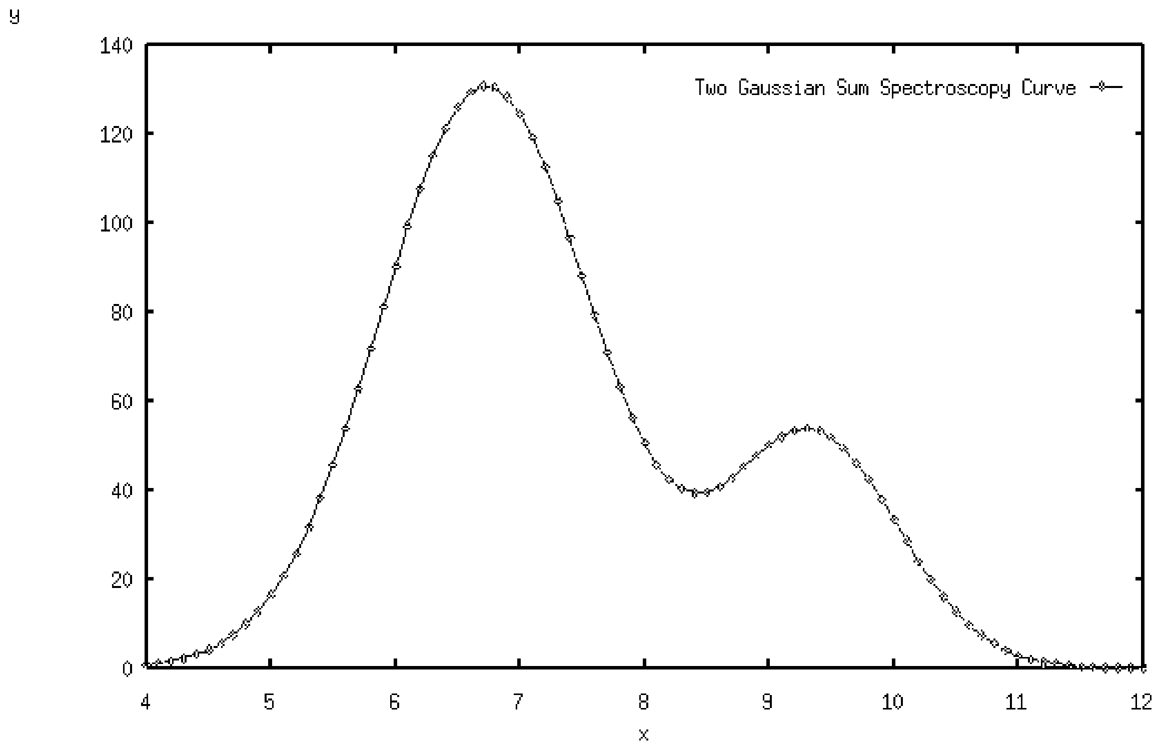

Figure 4: **Graph of Points**

to minimize f defined as follows:

$$f(a_1, a_2, u_1, u_2, s_1, s_2) = \sum_{i=1}^{n} \left(a_1 e^{-[\frac{x_i - u_1}{s_1}]^2} + a_2 e^{-[\frac{x_i - u_2}{s_2}]^2} - y_i \right)^2$$

Table 4: Initial Input Box for the Photoelectron Spectroscopy Problem

B
$a_1 = [130, 135]$
$a_2 = [50, 55]$
$u_1 = [6, 8]$
$u_2 = [8, 10]$
$s_1 = [1, 2]$
$s_2 = [0.5, 1]$

The results were as follows:

```
convex hull of boxes on the queue:


a1 = [130.889999624668920, 130.890000237423440]

a2 = [52.5999994426222910, 52.6000003353821410]

u1 = [6.72999999580056230, 6.73000000523584680]

u2 = [9.34199999170696670, 9.34200000792551850]

s1 = [1.19999999502502950, 1.20000000672384770]

s2 = [0.96999998507893725, 0.97000001469388031]
```

```
f(hull): [6.3015390640982946e-13, 9.9696829305332294e-11]

max width of 31 boxes on the queue = 5.57378e-07

computation time on 1 processor = 109240 seconds > 30 hours
```

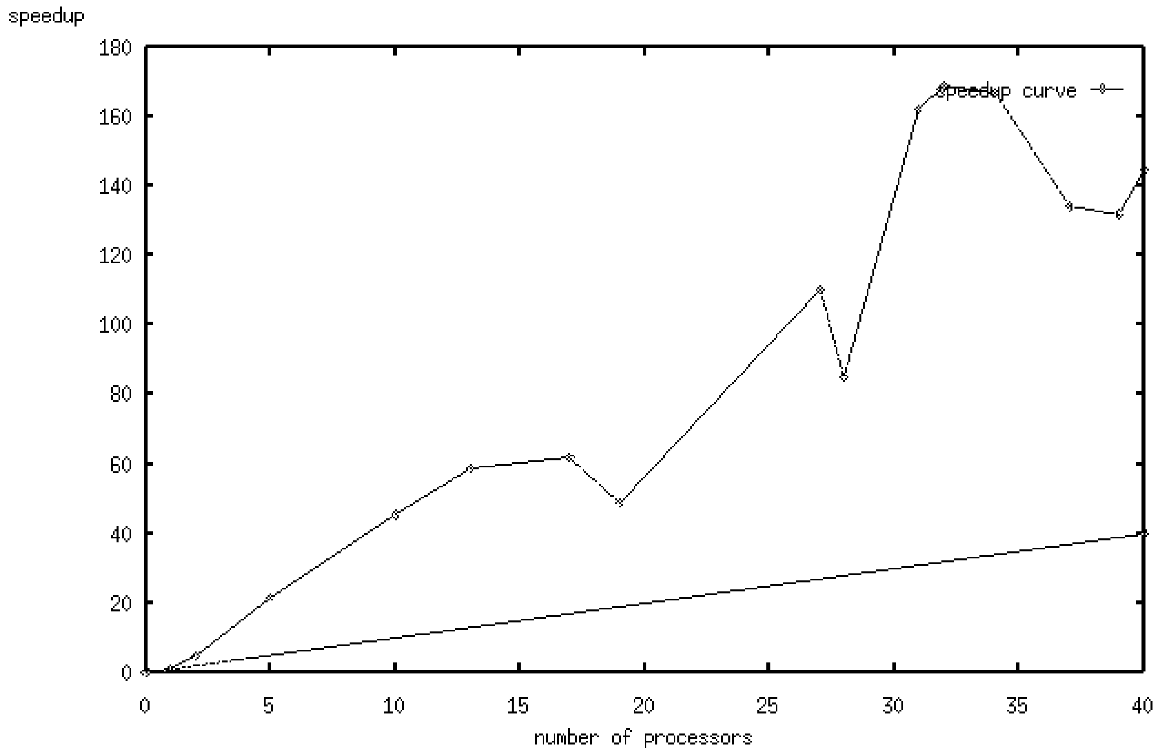

Figure 5: **Speedup Graph**

The parallel algorithm was run on up to 40 processors and achieved superlinear speedup as indicated by Figure 5. In order to explain this superlinear speedup, the progress of the parallel algorithm will be considered in the form of a binary tree.

At the beginning of the algorithm, one is usually given a single initial input box (denoted as the root of the tree). One either eliminates this box or divides it in half yielding two new boxes (depicted as child nodes). Likewise these two new boxes can be eliminated or split. Continuing in this manner, one gradually creates what shall be called a *binary progress tree.*

A portion of one possible binary progress tree is given in Figure 6. The rectangularized regions represent sets of boxes which would be deleted using the current upper bound U_{F_*} on the global minimum. An improved upper bound NU_{F_*} on the global minimum exists within box B_{30}.

In the single processor case, boxes are tested in the order B_1, B_2, $\ldots$, B_{31}. The reason for this is the fact that boxes are queued based upon the time in which they were generated. This progress amounts to a breadth first search of the entire tree for a "small" enough box containing a solution. Because of this searching strategy, the NU_{F_*} within box B_{30} would require 22 tests before being discovered.

In the two processor case (P_1 initially getting B_2 and P_2 initially getting B_3), each processor would employ a breadth first search on its respective half of the tree. Therefore P_2 would discover the NU_{F_*} in 6 tests (nearly 1/4 the Number of tests it took in the single processor case). Furthermore, P_2 would broadcast the NU_{F_*} to P_1 thus allowing P_1 to make sharper midpoint tests earlier and possibly "pruning" other subtrees from consideration. It is this combination of breadth first and depth first searching which is believed to account for the superlinear speedup of the parallel algorithm.

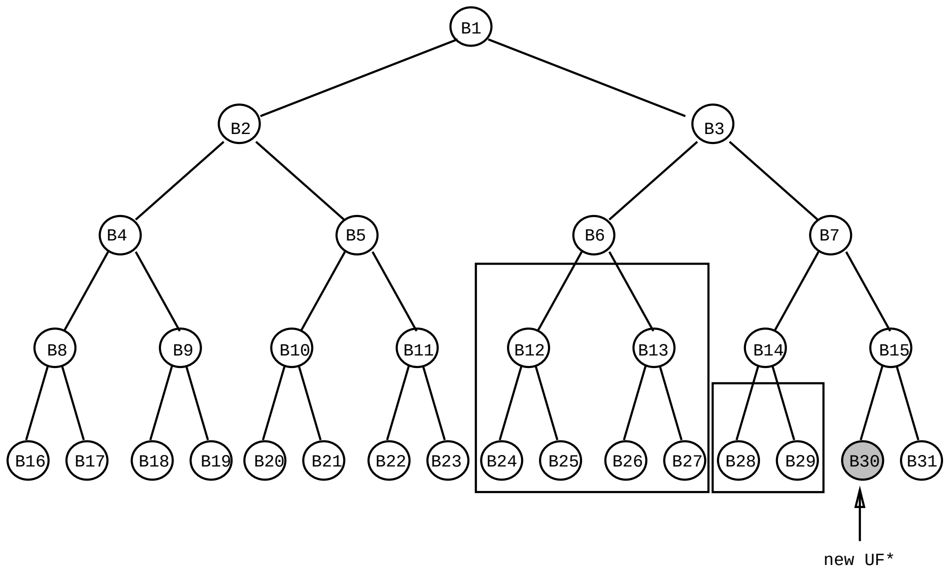

Figure 6: **Portion of One Possible Binary Progress Tree**

CHAPTER VI

CONCLUSIONS

This thesis has considered the topic of global optimization. The first objective was to consider algorithms which are *reliable.* The second goal was *efficiency.*

With regard to reliability, bounding methods were favored over point methods since the later are susceptible to the indentation argument as well as roundoff error. Among bounding techniques, interval arithmetic was chosen because it is

- capable of producing reliable, "tight", and asymptotically accurate bounds,
- efficient to compute, and
- applicable to *any* programmable function.

Machine interval arithmetic traps roundoff error thus always producing reliable bounds. Using C++, an added benefit is that machine interval arithmetic is

- easy to generalize and automate, and
- convenient for an unsophisticated user to utilize.

The second goal, that of efficiency, was attacked in two ways. The first direction sought to make the basic interval global optimization algorithm more efficient by taking advantage of certain properties of the objective function and constraint functions. With differentiability, the acceleration devices of the monotonicity test, convexity test, and Newton's method can be used to improve efficiency.

The second technique to improve efficiency was parallelization. A parallel interval global optimization algorithm was devised and mapped onto a network of workstations. The resulting speedup was encouraging.

6.1 Future Work

Much work can be done to improve the efficiency of the parallel interval global optimization algorithm discussed in this thesis. A few such possibilities will be briefly mentioned.

1. Methods such as centered forms, meanvalue forms, and Taylor forms [32] which attempt to produce tighter interval bounds could be investigated.

2. Since Newton's method is expensive in terms of time, a test signaling when Newton's method should be applied may improve efficiency. The author has experimented with such a test based upon the width of the "box" being considered.

3. Since a better U_{F_*} will almost certainly permit the midpoint test to dispose of more boxes, a method for determining such a better U_{F_*} early will, in general, improve efficiency [12]. The author has experimented with such possibilities using a local Newton's method search.

4. With regard to parallelization, several areas can be improved. Global termination is detected with a simple centralized algorithm. A distributed algorithm using either a ring [3] or a tree [39] would be more efficient and fault tolerant. Much experimentation can be made to determine the optimal load balancing scheme. For instance, more boxes could be sent when an idle processor requests work. Also, possibly only certain selected boxes should be sent to a requesting idle processor. Indeed the areas of partitioning and load balancing offer the richest bed of possibilities for improvement.

Appendix A

C++ Programs Evaluating the Rump Function

A.1 Floating Point Version

```
#include <math.h>


main()
{
  double x, y, f;


  x = 77617;
  y = 33096;


  f = 333.75*y*y*y*y*y*y+x*x * (11*x*x*y*y-y*y*y*y*y*y-121*y*y*y*y-2)
    + 5.5*y*y*y*y*y*y*y*y+x/(2*y);


  cout << f << endl;
}
```

A.2 Interval Version

```
#include <interval.h>


main()
{
  interval x, y, f;


  x = 77617;
  y = 33096;


  f = 333.75*y*y*y*y*y*y+x*x * (11*x*x*y*y-y*y*y*y*y*y-121*y*y*y*y-2)
    + 5.5*y*y*y*y*y*y*y*y+x/(2*y);


  cout << f << endl;
}
```

Appendix B

C++ Interval Class Used in this Thesis

B.1 Header File "interval.h"

```
#ifndef interval_h
#define interval_h

#include "mymath.h"
#include <stream.h>

class interval {
  double left;
  double right;
public:
         inline          interval()  {};
         inline          interval(double);
         inline          interval(double, double);
         inline          interval(interval &);
```

```
       inline interval &operator =  (interval &);
       inline interval &operator =  (double);
friend inline int       operator == (interval&, double);
friend inline int       operator == (double, interval&);
friend inline int       operator == (interval&, interval&);
// less than or equals is used for subset
friend inline int       operator <= (double, interval&);
friend inline int       operator <  (double, interval&);
friend inline int       operator <= (interval&, interval&);
friend inline int       operator >  (interval&, interval&);
friend inline int       operator >= (interval&, interval&);
friend inline int       operator != (interval&, double);
friend inline int       operator != (double, interval&);
friend inline int       operator != (interval&, interval&);

       inline interval  operator -  ();
friend        interval  operator +  (interval&, double);
friend        interval  operator +  (double, interval&);
friend        interval  operator +  (interval&, interval&);
friend        interval  operator -  (interval&, double);
friend        interval  operator -  (double, interval&);
friend        interval  operator -  (interval&, interval&);

friend        interval  operator *  (interval&, double);

friend        interval  operator *  (double, interval&);

friend        interval  operator *  (interval&, interval&);

friend        interval  operator /  (interval&, double);

friend        interval  operator /  (double, interval&);

friend        interval  operator /  (interval&, interval&);


// ampersand is used for intersection

friend inline int       operator &  (interval&, double);

friend inline int       operator &  (double, interval&);

friend        interval  operator &  (interval&, interval&);

friend        interval  operator |  (interval&, interval&);


              void      operator += (interval&);

              void      operator -= (interval&);

              void      operator *= (interval&);

              void      operator /= (interval&);


friend inline interval  floor(interval&);

friend        interval  sin(interval&);

friend        interval  cos(interval&);
friend        interval  atan(interval&);
friend        interval  arctan(interval&);
friend        interval  exp(interval&);
friend        interval  log(interval&);
friend        interval  ln(interval&);
friend        interval  pow(interval&, double);
friend        interval  sqrt(interval&);
friend        interval  sqr(interval&);
friend        interval  abs(interval&);
friend inline interval  poshalf(interval&);
friend inline interval  neghalf(interval&);
friend inline double&   left(interval&);
friend inline double&   right(interval&);
friend inline double&   leftpart(interval&);
friend inline double&   rightpart(interval&);
friend inline double    mid(interval&);
friend inline double    width(interval&);
friend inline double    sign(interval&);
friend inline int       empty(interval&);
friend inline int       negative(interval&);

friend        istream&  operator >> (istream&, interval&);
```

```
  friend        istream&  operator >> (istream&, decimal_record&);
  friend        ostream&  operator << (ostream&, interval&);
};

extern interval pi;
extern interval pi2;
extern interval piby2;
extern interval pi3by2;
extern interval pi5by2;

inline interval::interval(interval &x)
{
  left = x.left; right = x.right;
}

inline interval::interval(double c)
{
  left = right = c;
}

inline interval::interval(double l, double r)
{
  left = l; right = r;
}

inline interval &interval::operator = (interval &x)
{
  left = x.left; right = x.right; return *this;
}

inline interval &interval::operator = (double x)
{
  left = right = x; return *this;
}

inline int operator == (interval &x, interval &y)
{
  return x.left == y.left && x.right == y.right;
}

inline int operator == (interval &x, double y)
{
  return x.left == y && x.right == y;
}
```

```
inline int operator == (double x, interval &y)
{
  return x == y.left && x == y.right;
}

inline int operator <= (interval &x, interval &y)
{
  return x.left >= y.left && x.right <= y.right;
}

inline int operator <= (double x, interval &y)
{
  return x >= y.left && x <= y.right;
}

inline int operator < (double x, interval &y)
{
  return x > y.left && x < y.right;
}

inline int operator > (interval &x, interval &y)
```

```
{
  return x.left < y.left && x.right > y.right;
}

inline int operator >= (interval &x, interval &y)
{
  return x.left <= y.left && x.right >= y.right;
}

inline int operator != (interval &x, interval &y)
{
  return x.left != y.left || x.right != y.right;
}

inline int operator != (interval &x, double y)
{
  return x.left != y || x.right != y;
}

inline int operator != (double x, interval &y)
{
  return x != y.left || x != y.right;
}

inline interval interval::operator - ()
{
  return interval(-right, -left);
}

inline int operator & (interval &x, double y)
{
  return y >= x.left && y <= x.right;
}

inline int operator & (double x, interval &y)
{
  return x >= y.left && x <= y.right;
}

inline interval poshalf(interval &x)
// assumes x straddles 0
{
  return interval(0.0, x.right);
}
```

```
inline interval neghalf(interval &x)
// assumes x straddles 0
{
  return interval(x.left, -0.0);
}

inline double& left(interval &x)
{
  return x.left;
}

inline double& right(interval &x)
{
  return x.right;
}

inline double& leftpart(interval &x)
{
  return x.left;
}
```

```
inline double& rightpart(interval &x)
{
  return x.right;
}

inline double mid(interval &x)
{
  return (scalbn(x.left + x.right, -1));
}

inline double width(interval &x)
{
  return x.right - x.left;
}

inline double sign(interval &x)
{
  if (signbit(x.right))
    return -1.0;
  else if (signbit(x.left))
    return 0.0;
  else return 1.0;
}

inline int empty(interval &x)
{
  return isnan(x.left) && isnan(x.right);
}

inline int negative(interval &x)
{
  return signbit(x.right);
}

inline interval floor(interval &x)
{
  return interval(floor(x.left), floor(x.right));
}

#endif
```

B.2 Source File “interval.C”

```
#include <ctype.h>
#include "interval.h"

void ierror(char* p) {
  cerr << "interval: " << p << endl;
  abort();
}

#define MAXEXP     308
// number of digits of precision for I/O
#define NDIGITS     17

// mask for rounding down on sparc station
static int rdval = 0xc0000000;
// mask for rounding up on sparc station
static int ruval = 0x80000000;
// fsr() returns current rounding here
static int rval;
```

```
#define fsr()       {\
                    asm("sethi %hi(_rval),%o1");\
                    asm("st    %fsr,[%o1+%lo(_rval)]");\
                    }

#define rounddown() {\
                    asm("sethi %hi(_rdval),%o1");\
    asm("ld    [%o1+%lo(_rdval)], %fsr");\
    }

#define roundup()   {\
    asm("sethi %hi(_ruval),%o1");\
    asm("ld    [%o1+%lo(_ruval)], %fsr");\
    }

interval pi(3.1415926535897932, 3.1415926535897934);
interval pi2(scalbn(3.1415926535897932, 1),
     scalbn(3.1415926535897934, 1));
interval piby2(scalbn(3.1415926535897932, -1),
       scalbn(3.1415926535897934, -1));
interval pi3by2(3 * piby2);
interval pi5by2(5 * piby2);
interval operator + (interval &x, interval &y)
{
  rounddown();
  register double left = x.left + y.left;
  roundup();

  return interval(left, x.right + y.right);
}

interval operator + (interval &x, double y)
{
  rounddown();
  double left = x.left + y;;
  roundup();

  return interval(left, x.right + y);
}

interval operator + (double x, interval &y)
{
  rounddown();
  double left = x + y.left;
  roundup();

  return interval(left, x + y.right);
}

interval operator - (interval &x, interval &y)
{
  rounddown();
  register double left = x.left - y.right;
  roundup();

  return interval(left, x.right - y.left);
}

interval operator - (interval &x, double y)
{
  rounddown();
  double left = x.left - y;
  roundup();

  return interval(left, x.right - y);
```

```
}

interval operator - (double x, interval &y)
{
  rounddown();
  double left = x - y.right;
  roundup();

  return interval(left, x - y.left);
}

void interval::operator += (interval &x)
{
  rounddown();
  left += x.left;
  roundup();
  right += x.right;
}

void interval::operator -= (interval &x)
{
  rounddown();
  left -= x.right;

  roundup();

  right -= x.left;

}


interval sqrt(interval &x)

{

  rounddown();

  double left = sqrt(x.left);

  roundup();


  return interval(left, sqrt(x.right));

}


interval exp(interval &x)

{

  rounddown();

  double left = exp(x.left);

  roundup();


  return interval(left, exp(x.right));

}
```

```
interval log(interval &x)
{
  rounddown();
  double left = log(x.left);
  roundup();

  return interval(left, log(x.right));
}

interval ln(interval &x)
{
  rounddown();
  double left = log(x.left);
  roundup();

  return interval(left, log(x.right));
}

interval arctan(interval &x)
{
  rounddown();
  double left = atan(x.left);
  roundup();

  return interval(left, atan(x.right));
}

interval operator & (interval &x, interval &y)
{
  if (x.right < y.right) {
    if (y.left < x.left)
      return x;
    else if (y.left <= x.right)
      return interval(y.left, x.right);
    else
      return interval(quiet_nan(0));
  }
  else {
    if (x.left < y.left)
      return y;
    else if (x.left <= y.right)
      return interval(x.left, y.right);
    else
```

```
      return interval(quiet_nan(0));
    }
  }

  interval operator | (interval &x, interval &y)
  {
    ierror("not implemented yet");
  }

  void interval::operator *= (interval &y)
  // cases are from Moore's book ``Methods and Applications
  // of Interval Analysis'' page 12
  {
    double lft, rght, temp;

    if (signbit(left)) {                  // cases 2,3,5,7,8 and 9
      if (signbit(y.left)) {              // cases 5,7, and 8
        if (signbit(right)) {             // cases 5 and 8
          roundup();
          rght = left * y.left;
          rounddown();
          if (signbit(y.right))           // case 8
      lft = right * y.right;
    else                               // case 5
      lft = left * y.right;
  }
  else {                               // cases 7 and 9
    if (signbit(y.right)) {            // case 7
      rounddown();
      lft = right * y.left;
      roundup();
      rght = left * y.left;
    }
    else {                             // case 9
      rounddown();
      if ((temp = left * y.right) < (lft = right * y.left))
        lft = temp;
      roundup();
      if ((temp = left * y.left) > (rght = right * y.right))
        rght = temp;
    }
  }
}
 else {                                // cases 2 and 3
```

```
    rounddown();
    lft = left * y.right;
    roundup();
    if (signbit(right))                 // case 3
      rght = right * y.left;
    else                                // case 2
      rght = right * y.right;
  }
}
else {                                  // cases 1,4, and 6
  if (signbit(y.left)) {                // cases 4 and 6
    rounddown();
    lft = right * y.left;
    roundup();
    if (signbit(y.right))               // case 6
      rght = left * y.right;
    else                                // case 4
      rght = right * y.right;
  }
  else {                                // case 1
    rounddown();
    lft = left * y.left;
```

```
      roundup();
      rght = right * y.right;
    }
  }
  left = lft; right = rght;
}


interval operator * (interval &x, interval &y)
// cases are from Moore's book ‘‘Methods and Applications
// of Interval Analysis’’ page 12
{
  double left, right, temp;

  if (signbit(x.left)) {                // cases 2,3,5,7,8 and 9
    if (signbit(y.left)) {              // cases 5,7, and 8
      if (signbit(x.right)) {           // cases 5 and 8
        roundup();
        right = x.left * y.left;
        rounddown();
        if (signbit(y.right))           // case 8
          left = x.right * y.right;
        else                            // case 5
      left = x.left * y.right;
    }
    else {                              // cases 7 and 9
      if (signbit(y.right)) {           // case 7
        rounddown();
        left = x.right * y.left;
        roundup();
        right = x.left * y.left;
      }
      else {                            // case 9
        rounddown();
        if ((temp = x.left * y.right) < (left = x.right * y.left))
          left = temp;
        roundup();
        if ((temp = x.left * y.left) > (right = x.right * y.right))
          right = temp;
      }
    }
  }
   else {                               // cases 2 and 3
    rounddown();
    left = x.left * y.right;
    roundup();
    if (signbit(x.right))               // case 3
      right = x.right * y.left;
    else                                // case 2
      right = x.right * y.right;
  }
}
else {                                  // cases 1,4, and 6
  if (signbit(y.left)) {                // cases 4 and 6
    rounddown();
    left = x.right * y.left;
    roundup();
    if (signbit(y.right))               // case 6
      right = x.left * y.right;
    else                                // case 4
      right = x.right * y.right;
  }
  else {                                // case 1
    rounddown();
    left = x.left * y.left;
    roundup();
    right = x.right * y.right;
    }
  }
  return interval(left, right);
}

interval operator * (interval &x, double y)
{
  double left;

  if (signbit(y)) {
    rounddown();
    left = x.right * y;
    roundup();
    return interval(left, x.left * y);
  }
  else {
    rounddown();
    left = x.left * y;
    roundup();
    return interval(left, x.right * y);
  }
}
```

```
interval operator * (double x, interval &y)
{
  double left;

  if (signbit(x)) {
    rounddown();
    left = x * y.right;
    roundup();
    return interval(left, x * y.left);
  }
  else {
    rounddown();
    left = x * y.left;
    roundup();
    return interval(left, x * y.right);
  }
}

void interval::operator /= (interval &y)
// cases are from Moore's book ``Methods and Applications
// of Interval Analysis'' page 12
```

```
{
  double lft, rght;

  if (signbit(left)) {                  // cases 2,3,5,7,8 and 9
    if (signbit(y.left)) {              // cases 5,7, and 8
      if (signbit(right)) {             // cases 5 and 8
        if (signbit(y.right)) {         // case 8
          rounddown();
          lft = right / y.left;
          roundup();
          rght = left / y.right;
        }
        else {                          // case 5
  cerr << y << endl;
          ierror("divide by interval containing zero");
}
      }
      else {                            // cases 7 and 9
        if (signbit(y.right)) {         // case 7
          rounddown();
          lft = right / y.right;
          roundup();
        rght = left / y.right;
      }
      else {                         // case 9
        lft = -HUGE_VAL; rght = HUGE_VAL;
      }
    }
  }
  else {                              // cases 2 and 3
    rounddown();
    lft = left / y.left;
    roundup();
    if (signbit(right))               // case 3
      rght = right / y.right;
    else                              // case 2
      rght = right / y.left;
  }
}
else {                                // cases 1,4, and 6
  if (signbit(y.left)) {              // cases 4 and 6
    if (signbit(y.right)) {           // case 6
      rounddown();
      lft = right / y.right;
```

```
        roundup();
        rght = left / y.left;
      }
      else {                            // case 4
cerr << y << endl;
        ierror("divide by interval containing zero");
      }
    }
    else {                              // case 1
      rounddown();
      lft = left / y.right;
      roundup();
      rght = right / y.left;
    }
  }
  left = lft; right = rght;
}

interval operator / (interval &x, interval &y)
// cases are from Moore's book ``Methods and Applications
// of Interval Analysis'' page 12
{
  double left, right;

  if (signbit(x.left)) {                 // cases 2,3,5,7,8 and 9
    if (signbit(y.left)) {               // cases 5,7, and 8
      if (signbit(x.right)) {            // cases 5 and 8
        if (signbit(y.right)) {          // case 8
          rounddown();
          left = x.right / y.left;
          roundup();
          right = x.left / y.right;
        }
        else {                            // case 5
  cerr << x << "\n" << y << endl;
          ierror("divide by interval containing zero");
}
      }
      else {                             // cases 7 and 9
        if (signbit(y.right)) {          // case 7
          rounddown();
          left = x.right / y.right;
          roundup();
          right = x.left / y.right;
      }
      else {                        // case 9
        left = -HUGE_VAL; right = HUGE_VAL;
      }
    }
  }
  else {                              // cases 2 and 3
    rounddown();
    left = x.left / y.left;
    roundup();
    if (signbit(x.right))             // case 3
      right = x.right / y.right;
    else                              // case 2
      right = x.right / y.left;
  }
}
else {                                // cases 1,4, and 6
  if (signbit(y.left)) {              // cases 4 and 6
    if (signbit(y.right)) {           // case 6
      rounddown();
      left = x.right / y.right;
      roundup();
```

```
        right = x.left / y.left;
      }
      else {                                   // case 4
cerr << x << "\n" << y << endl;
        ierror("divide by interval containing zero");
      }
    }
    else {                                     // case 1
      rounddown();
      left = x.left / y.right;
      roundup();
      right = x.right / y.left;
    }
  }
  return interval(left, right);
}

interval operator / (interval &x, double y)
{
  double left;

  if (signbit(y)) {
    rounddown();
    left = x.right / y;
    roundup();
    return interval(left, x.left / y);
  }
  else {
    rounddown();
    left = x.left / y;
    roundup();
    return interval(left, x.right / y);
  }
}

interval operator / (double x, interval &y)
{
  double left;

  if (signbit(y.left) && !signbit(y.right)) {
    cerr << x << "\n" << y << endl;
    ierror("divide by interval containing zero");
  }
  if (signbit(x)) {
    rounddown();
    left = x / y.left;
    roundup();
    return interval(left, x / y.right);
  }
  else {
    rounddown();
    left = x / y.right;
    roundup();
    return interval(left, x / y.left);
  }
}

interval sqr(interval &x)
{
  double left, right;

  if (signbit(x.right)) {           // all negative interval
    rounddown();
    left = x.right * x.right;
    roundup();
    right = x.left * x.left;
  }
  else if (signbit(x.left)) {       // stradle zero interval
    roundup();
    left = x.left * x.left;
    right = x.right * x.right;
    if (left > right)
      right = left;
    left = 0.0;
  }
  else {                           // all positive interval
    rounddown();
    left = x.left * x.left;
    roundup();
    right = x.right * x.right;
  }
  return interval(left, right);
}

interval abs(interval &x)
{
  double left, right;
  if (signbit(x.right)) {           // all negative interval
    left  = -x.right;
    right = -x.left;
  }
  else if (signbit(x.left)) {       // stradle zero interval
    left = -x.left;
    right = x.right;
    if (left > right)
      right = left;
    left  = 0.0;
  }
  else {                           // all positive interval
    left  = x.left;
    right = x.right;
  }
  return interval(left, right);
}

interval sin(interval &x)
{
  interval sine;
  double temp;
```

```
interval distr, distl;

if (x.left < 0 || x.right > pi2.right) {
  cout << "bad argument for restricted sine of " << x << "\n";
  return interval(-1, 1);
}

if (!empty(x & piby2)) {

  if (!empty(x & pi3by2)) {
    sine.left = -1.0;
    sine.right = 1.0;
    return sine;
  }

  sine.right = 1.0;

  distr = x.right - piby2;
  distl = piby2 - x.left;

  rounddown();
```

```
    if (distr.left > distl.right)
      sine.left = sin(x.right);
    else if (distl.left > distr.right)
      sine.left = sin(x.left);
    else {
      sine.left = sin(x.left);
      temp = sin(x.right);
      if (temp < sine.left)
sine.left = temp;
    }
    return sine;
  }

  if (!empty(x & pi3by2)) {

    sine.left = -1.0;

    distr = x.right - pi3by2;
    distl = pi3by2 - x.left;

    roundup();
    if (distr.left > distl.right)
      sine.right = sin(x.right);
    else if (distl.left > distr.right)
      sine.right = sin(x.left);
    else {
      sine.right = sin(x.left);
      temp = sin(x.right);
      if (temp > sine.right)
sine.right = temp;
    }
    return sine;

  }

  if (x.right < piby2.left || x.left > pi3by2.right) {
    rounddown();
    sine.left = sin(x.left);
    roundup();
    sine.right = sin(x.right);
    return sine;
  }
  rounddown();
  sine.left = sin(x.right);
  roundup();
  sine.right = sin(x.left);

  return sine;
}

interval cos(interval &x)
{
  interval cossine;
  double   temp;

  if (x.left < 0 || x.right > pi2.right) {
    cout << "bad argument for restricted cossine of " << x << "\n";
    return interval(-1, 1);
  }

  if (!empty(x & pi)) {

    cossine.left = -1.0;
    roundup();
  cossine.right = cos(x.left);
  temp = cos(x.right);
  if (temp > cossine.right)
    cossine.right = temp;

  return cossine;
}

if (x.left > pi.right) {

  rounddown();
  cossine.left = cos(x.left);
  roundup();
  cossine.right = cos(x.right);

  return cossine;
}

rounddown();
cossine.left = cos(x.right);
roundup();
cossine.right = cos(x.left);
```

```
  return cossine;
}

interval pow(interval &x, double p)
{
  rounddown();
  double left = pow(x.left, p);
  roundup();

  return interval(left, pow(x.right, p));
}

istream &operator >> (istream &s, decimal_record &pd)
{
  char c;
  register int i, exp = 0;

  pd.fpclass = fp_normal;
  // indicates no additional nonzero digits follow
  pd.more = 0;
```

```
// get first not white space character
s >> c;

// assume, at first, that the sign is positive
pd.sign = 0;
if (c == '-') {
  // set sign to negative
  pd.sign = 1;
  s >> c;
}
else if (c == '+')
  s >> c;

// extract each digit left of the decimal point
for (i = 0; isdigit(c); s.get(c))
  pd.ds[i++] = c;

if (c == '.')
  // extract each digit right of the decimal point
  for (s.get(c); isdigit(c); s.get(c), exp--)
    pd.ds[i++] = c;
  if (!i)
    ierror("invalid decimal input string base");

  pd.ds[i] = '\0';

  if (c == 'e' || c == 'E') {
    s >> pd.exponent;
    if (!s.good())
      ierror("invalid decimal input string exponent");
    if (pd.exponent > MAXEXP || pd.exponent < -MAXEXP)
      ierror("input string exponent is out of range");
    pd.exponent += exp;
  }
  else {
    pd.exponent = exp;
    s.putback(c);
  }
  return s;
}

istream &operator >> (istream &s, interval &x)
{
```

```
char c;
decimal_mode            pm;
decimal_record          pd;
fp_exception_field_type ps;

s >> c;
if (c == '[') {
  s >> pd >> c;
  pm.rd = fp_negative;
  decimal_to_double(&x.left, &pm, &pd, &ps);
  if (c == ',')
    s >> pd >> c;
  if (c == ']') {
    pm.rd = fp_positive;
    decimal_to_double(&x.right, &pm, &pd, &ps);
  }
}
else {
  s.putback(c);
  s >> pd;
  pm.rd = fp_negative;
  decimal_to_double(&x.left, &pm, &pd, &ps);
```

```
    pm.rd = fp_positive;
    decimal_to_double(&x.right, &pm, &pd, &ps);
  }
  return s;
}

ostream &operator << (ostream &s, interval &x)
{
  char ds[NDIGITS + 6];

  s.put('[');
  s.put(' ');

  rounddown();
  // 0 parameter means NO trailing zeros
  gconvert(x.left, NDIGITS, 0, ds);

  s << ds;

  s.put(',');
  s.put(' ');
  roundup();
  // 0 parameter means NO trailing zeros
  gconvert(x.right, NDIGITS, 0, ds);


  s << ds;


  s.put(' ');
  s.put(']');


  return s;
}
```

Bibliography

[1] D. Briggs and M. P. Seah. *Practical Surface Analysis: by Auger and X-ray Photoelectron Spectroscopy.* Wiley, Chichester, New York, 1983.

[2] Matthew Bromberg and Tsu-Shuan Chang. *Recent Advances in Global Optimization*, pages 200–220. Princeton University Press, 41 William Street, Princeton, 1992. Linear lower bound approach.

[3] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.

[4] Russell S. Drago. *Physical Methods in Chemistry.* Saunders Golden Sunburst Series. W.B. Saunders Company, Philadelphia, PA 19105, 1977.

[5] P. S. Dwyer. *Computation with Approximate Numbers*, pages 11–34. Wiley and Sons, New York, 1951. Linear Computations book.

[6] J. S. Ely. *Prospects for Using Variable Precision Interval Software in C++ for Solving Some Contemporary Scientific Problems.* PhD thesis, The Ohio State University, 1990.

[7] Yu. G. Evtushenko, M. A. Potapov, and V. V. Korotkich. *Recent Advances in Global Optimization*, pages 274–297. Princeton University Press, 41 William Street, Princeton, 1992. Covering methods.

[8] E. A. Galperin. Control and games. *International Journal of Global Optimization.* To appear, personal communication.

[9] E. A. Galperin. Global optimization, control and games. *Special Issue of the International Journal of Computers and Mathematics*, 1990.

[10] E. A. Galperin. Global optimization, control and games. *Second Special Issue of the International Journal of Computers and Mathematics*, 1991. To appear.

[11] E. R. Hansen. Global optimization using interval analysis — the multi-dimensional case. *Numer. Math.*, 34:247–270, 1980.

[12] E. R. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, Inc., New York, 1992. To be published.

[13] E. R. Hansen and R. I. Greenberg. An interval newton method. *Appl. Math. Comp.*, 12:89–98, 1983.

[14] R. Horst. Journal of global optimization. 1(1), 1991.

[15] K. Ichida and Y. Fujii. An interval arithmetic method for global optimization. *Computing*, 23:85–97, 1979.

[16] A. H. G. Rinnooy kan and G. T. Timmer. *New Methods in Optimization and their Industrial Uses*, pages 133–155. Birkhäuser Verlag, Basel, 1989. Argument for the unsolvability of global optimization problems.

[17] Hiroshi Konno and Yasutoshi Yajima. *Recent Advances in Global Optimization*, pages 259–273. Princeton University Press, 41 William Street, Princeton, 1992. First example global optimization problem.

[18] A. P. Leclerc. Newton's method. Technical Report OSU-CISRC-01/91-TR-04, The Ohio State University Computer and Information Science Research Center, January 1991.

[19] R. E. Moore. Automatic error analysis in digital computation. Technical Report LMSD-4842, Lockheed Missiles and Space Division, Sunnyvale, California, 1959. Early interval arithmetic paper.

[20] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[21] R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM Studies in Applied Mathematics. SIAM, Philadelphia, 1979.

[22] R. E. Moore. *Reliability in Computing.* Academic Press, 1988. See especially the papers by E. Hansen 289-308, G. W. Walster 309-324, H. Ratschek 325-340, and W. A. Lodwick 341-354.

[23] R. E. Moore. Global optimization to prescribed accuracy. *Computers Math. Applic.*, 21:25–39, 1991.

[24] R. E. Moore. Interval tools for computer aided proofs in analysis. *The IMA Volumes in Mathematics and Its Applications*, 28:211–216, 1991.

[25] R. E. Moore, E. R. Hansen, and A. P. Leclerc. *Recent Advances in Global Optimization*, pages 321–342. Princeton University Press, 41 William Street, Princeton, 1992. Interval Global Optimization.

[26] R. E. Moore and H. Ratschek. Inclusion functions and global optimization II. *Mathematical Programming*, 41:341–356, 1988.

[27] P. M. Pardalos and S. A. Vavasis. Quadratic programming with one negative eigenvalue is np-hard. *Journal of Global Optimization*, 1(1):15–22, 1991. NP-hard example.

[28] Jean-Paul Penot. *New Methods in Optimization and their Industrial Uses*, page ix. Birkhäuser Verlag, Basel, 1989. Quote emphasizing the importance of global optimization.

[29] János Pintér. *Recent Advances in Global Optimization*, pages 399–432. Princeton University Press, 41 William Street, Princeton, 1992. Lipschitzian Global Optimization.

[30] H. Ratschek. Inclusion functions and global optimization. *Mathematical Programming*, 33(3):300–317, 1985.

[31] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions.* Ellis Horwood and John Wiley, 1984.

[32] H. Ratschek and J. Rokne. *New Computer Methods for Global Optimization.* Ellis Horwood and John Wiley, 1988.

[33] S. M. Rump. *Reliability in Computing. The Role of Interval Methods in Scientific Computing.* Academic Press, 1988. roundoff error example.

[34] Sartaj Sahni. Computationally related problems. *SIAM Journal on Computing*, 3(4):262–279, December 1974.

[35] S. Skelboe. Computation of rational interval functions. *BIT*, 14:87–95, 1974.

[36] Peter E. Sobol. A comparison of techniques for compositional and chemical analysis of surfaces. *Perkin-Elmer*, 11(2):2–5, Winter 1989.

[37] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.

[38] T. Sunaga. Theory of an interval algebra and its application to numerical analysis. *RAAG Memoirs*, 2:547–564, 1958. Early investigation into interval arithmetic.

[39] Rodney W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18(1):33–36, January 1984.

[40] Jean VIGNES. *New Methods in Optimization and their Industrial Uses*, pages 219–227. Birkhäuser Verlag, Basel, 1989. Estimation of the accuracy of results.

[41] G. W. Walster, E. R. Hansen, and S. Sengupta. *Test Results for a Global Optimization Algorithm*, pages 272–287. Numerical Optimization. SIAM, 1985. Boggs, Byrd, Schnabel (eds.).

[42] M. Warmus. Calculus of approximations. *Bull. Acad. Polon. Sci. Cl. III*, 4:463–464, 1956. Early investigation into interval arithmetic.

[43] Zelda B. Zabinsky, Douglas L. Graesser, Mark E. Tuttle, and Gun-In Kim. *Recent Advances in Global Optimization*, pages 343–368. Princeton University Press, 41 William Street, Princeton, 1992. Second example global optimization problem.