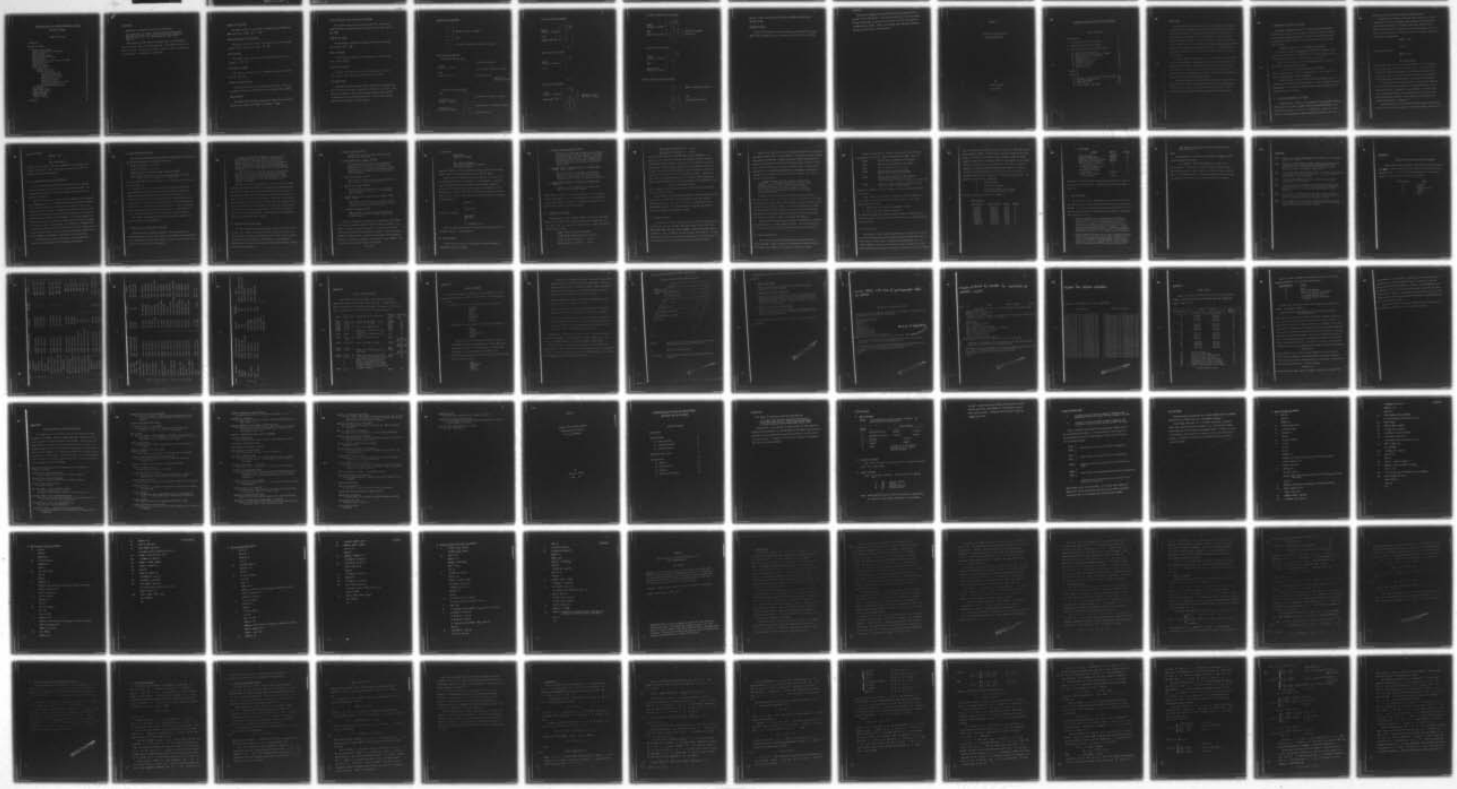
AD-A078 089 LOCKHEED MISSILES AND SPACE CO INC PALO ALTO CA PALO --ETC F/G 12/1
GLOBALMIN - A COMPUTER PROGRAM FOR GLOBAL OPTIMIZATION.(U)
NOV 79 E R HANSEN F49620-76-C-0003
UNCLASSIFIED LMSC/D683307 AFOSR-TR-79-1227 NL

1 OF 2
AD-
A078089



AFOSR-TR- 79-1227

(3)

LEVEL

AD A078089

79 12 10 069

LOCKHEED
MISSILES & SPACE COMPANY INC • SUNNYVALE CALIFORNIA

GLOBALMIN — A COMPUTER PROGRAM FOR GLOBAL OPTIMIZATION

Eldon R. Hansen

15 November 1979 LMSC-D683307

Applied Mechanics Laboratory
Lockheed Palo Alto Research Laboratory
3251 Hanover Street
Palo Alto, CA 94304

(415) 493-4411, X 45133

(18) AFOSR (19) TR-79-1227

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 79-1227	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) GLOBALMIN – A COMPUTER PROGRAM FOR GLOBAL OPTIMIZATION	5. TYPE OF REPORT & PERIOD COVERED Interim report for period 7/15/76 through 9/15/79
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Eldon R. Hansen	8. CONTRACT OR GRANT NUMBER(s) (15) F49620-76-C-0003
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lockheed Palo Alto Research Laboratory 3251 Hanover Street (52-33/205) Palo Alto, California 94304	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A3
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research (NM) Bldg. 410, Bolling Air Force Base Washington, DC 20332	(11) 12. REPORT DATE 15 November 1979
	13. NUMBER OF PAGES 122
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (12) 126	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

(16) 2304

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(14) LMSC/D683907 (17) A3

18. SUPPLEMENTARY NOTES

(cont'd p. 6) Tech. Other (9) Interim rept. 15 Jul 76–15 Sep 79,

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Global optimization
Interval arithmetic
Unconstrained optimization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The implementation of an algorithm that provides infallible bounds on the global minimum of any twice continuously differentiable real function of n real variables on a closed, bounded domain is described. The algorithm also provides infallible bounds on the location of the global minimum. The algorithm uses interval arithmetic and requires the availability of several fundamental interval arithmetic processors for its operation.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

210 118 -B-

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. INTRODUCTION

GLOBALMIN is a program for finding the global minimum of a nonlinear function f (C^2) of n variables. It provides infallible bounds on the minimum value F* of f in any prescribed box (a box is a parallelopiped with sides parallel to the coordinate axes). It also provides infallible bounds on the point(s) x* at which the global minimum occurs.

It is assumed that the minimum occurs at a stationary point of f. It is possible to modify the program to allow for the case in which the global minimum is not a stationary point and hence occurs on the boundary. The necessary modifications are described in Appendix 4, which is to appear in Numerische Mathematik. This paper describes the theoretical and practical considerations used in writing GLOBALMIN. It also provides the results of several test problems treated by GLOBALMIN.

The work described here directly builds upon earlier work carried out at Washington State University in the general area of interval arithmetic. In particular, three reports by Carol Clarke that describe fundamental processors used by GLOBALMIN are included as Appendices in this report for completeness. We describe particular considerations in regard to this earlier work that are germane to GLOBALMIN in the next two sections.

2. THE CLUDGE PRECOMPILER

GLOBALMIN was written assuming a precompiler named CLUDGE would be used. The main purpose of CLUDGE is to accept and work with intervals in the same way Fortran works with real numbers, complex numbers, integers, etc. It also provides interval subroutines for evaluating the elementary transcendental functions with interval arguments.

We now briefly describe some of the features of CLUDGE. For a more detailed discussion, see Appendix 1.

2.1 Interval Variables

In order for CLUDGE to accept a variable, say A, as an interval, the variable must be so declared as INTRVAL A. It is possible to dimension an interval variable. For example, INTRVAL A(5,7). To read in or print out an interval, you must leave room for two numbers. For example,

```
      INTRVAL A
      READ 101,A
  101 FORMAT (2F10.0)
```

When you do use READ or WRITE statements that involve interval variables, a warning appears. It's harmless and usually unnecessary.

2.2 Miscellaneous Warnings

Although the CLUDGE package includes many functions, it doesn't declare their type for you. For example, if you wish to use INTUON, an interval-valued function, you must declare INTUON as an interval name. For example:

```
INTERVAL INTUON, TA,TB,TC
          .
          .
          .
TC=INTUON(TA,TB)
```

As another example, INTINF and INTSUP are real-valued functions and must be declared so:

```
REAL INTINF,INTSUP
INTRVAL TA.
          .
          .
          .
P=INTINF(TA)
Q=INTSUPNTA)
```

Finally, when you write your own functions, explicitly state the type, even if it isn't INTRVAL.

3. NOTATION AND TERMINOLOGY

In the comments within the program and within the following writeup, the authors have used and/or invented some unusual terms. A few comments are in order. As an example, consider AIA(1-N,K).

If you encounter a symbol such as the above in a comment, it refers to the vector residing in the consecutive memory locations AIA(1,K). AIA(2,K)... AIA(N,K). For this particular example, it is an interval vector.

The comments make use of the following terminology:

A box is the cartesian product of N intervals. Equivalently, a box is an interval vector. Note that AIA(1-N,K) is the K-th box in the list of boxes AIA.

The width of a box is the width of the widest interval defining the box. Thus if X is an interval vector (box) with components X_i ($i = 1,\ldots,N$) and if $w(X_i)$ denotes the width of the interval X_i, then the width of the box is

$$\max_{1 \le i \le N} w(X_i)$$

The functional width of a box is the width of the interval obtained by evaluating the object function f over the box (using interval arithmetic). Note that this interval contains the range of the function f over the box.

The driver MAIN. The driver for the program is called MAIN. It is used to call GLOBE which is the central subroutine of the program.

The subroutine GLOBE. The minimization algorithm uses a number of sub-algorithms to find the global minimum. These subroutines are called by GLOBE.

Explanation of arguments. The calling statement for GLOBE is

```
CALL GLOBE (N, XL, XR, AIW, AIWM, ARW, ARWM, ARWMA, AZW, AIA,
AIB, AIC, NDIM, KENDC, AID, EA, EB, P, FLOW, FBAR, NSTEP)
```

The arguments are:

N is the dimension of the domain.

XL is the vector of left endpoints of the current box. Thus XL(I) (I = 1,...,N) is the left endpoint of the interval defining the I-th component of the box.

XR is the vector of right endpoints of the current box.

AIW is an interval matrix of dimension N by 8. It is used as a set of interval work vectors.

AIWM is an N by N matrix containing the interval Hessian.

ARW is a real N by 8 matrix. It is used as a set of real work vectors.

ARWM is a real N by N matrix containing the midpoint of AIWM.

ARWMA is a real N by N matrix containing the approximate inverse of ARWM.

AZW is an integer vector of order N. It is used to store flags for various purposes.

AIA is an interval matrix with N rows. It contains the insufficiently small boxes. Each column is a box generated by the program and the number of boxes changes as the program runs. For small, easy problems the number of columns required may be 10 or less. For larger or more difficult problems it is safer to allow for 100 boxes. The program is written so that the current box, which is chosen from AIA, tends to be the largest box in AIA. This improves the speed of the algorithm but causes the number of boxes to be larger. To reduce storage (while increasing the run time somewhat), the program can be modified so that the current box is chosen to be the smallest one in AIA.

AIB is an interval matrix with N rows. It contains boxes small enough to satisfy the error criterion. As for AIA, the number of columns depends on the problem. If the problem has only one or two solutions, AIB should never contain more than 5 or 10 boxes. If the problem has m solutions, then AIB will probably contain at least m boxes eventually.

AIC is an interval matrix with N rows. It contains boxes which satisfy both the requirement of smallness (width EA) and the criterion of small functional width (EB). At the end of the run, all remaining boxes will be in AIC. Since tentative solution boxes are also put temporarily in AIC, it

must contain, say, ten more columns than there are solution points. The program could be rewritten to occasionally purge AIC of boxes which do not contain solutions. Thus, the dimension of AIC could be smaller than is currently necessary.

NDIM is the number of columns in AIA. Since the number of boxes to be stored in AIA is not known in advance, it must be guessed. A safe choice (generally) is 100.

KENDC is the number of boxes in AIC.

AID is an interval vector containing the functional widths of the boxes in AIC. Thus it has the same number of elements as the number of columns in AIC.

EA is an error tolerance. Any final box must be less than EA in width.

EB is an error tolerance. The functional width of f over each final box must be less than EB. When the algorithm terminates, we will have FBAR-FLOW<EB.

P is a parameter which has been set (rather arbitrarily) to 0.75. If during one complete iteration, the current bos is not reduced in width by at least the factor 1-P, then the box is split in half. More experience is needed to choose P optimally.

FLOW is a lower bound on the global minimum f*.

FBAR is an upper bound on f*.

NSTEP is the number of iterative steps used by algorithm.

User supplied subroutines. The user must supply subroutines to evaluate the object function f, its gradient g, and its Hessian H. The program requires interval forms of these functions. The sharpness of the results, and hence the rate of convergence of the algorithm, depends on the steps used to evaluate these functions. Hence the subroutines should be written so as to obtain as sharp results as feasible.

The object function f(x) must be evaluated as an interval function named FI and be of the form

```
INTRVAL FUNCTION FI (N,AIW,KNEXTA,L)
```

Here N is the dimension of the problem, AIW is matrix wherein the interval argument vector (the box over which f is being evaluated) is stored and KNEXTA is the specific column of AIW in which it is stored. L is a flag which allows recovery or termination if difficulty (such as overflow) occurs in the evaluation. Set L = 0 if all goes well. Set L = 1 otherwise. No actual use of the flag is made by the program. Its use is included to aid the user in recovery or termination if he chooses to modify the program appropriately.

The interval gradient must be called GI and be evaluated using

SUBROUTINE GI (N,AIW,KNEXTA,AJW,JCOL,L).

The arguments N, AIW, KNEXTA and L are the same as for FI. The argument AJW specifies the matrix in which the interval gradient is to be stored and JCOL specifies the column of AJW.

The interval Hessian must be called HI and be evaluated using

SUBROUTINE HI (N,AIW,KNEXTA,AIWM,L).

The arguments N, AIW, KNEXTA, and L are the same as for FI or GI. The interval Hessian should be placed in the matrix AIWM.

When computing the Hessian, different input arguments for different elements can be real (degenerate intervals). See the accompanying paper by E. Hansen. Real arguments should be used when possible to reduce the widths of the interval elements of the Hessian.

There is a need to have the diagonal elements of the Hessian with all interval elements. This is used for a non-convexity check. Thus the user must supply

SUBROUTINE HDIAG (N,AIW,KNEXTA,JCOL).

The argument N, AIW, and KNEXTA are as in HI. The argument JCOL specifies the column of AIW in which the diagonal of the Hessian (with non-degenerate input) should be stored as an interval vector.

Miscellaneous comments. The mode used does not allow passing of parameters. This problem is circumvented by placing parameters in a COMMON block and referencing them in both the main program and the five subprograms. However, space should be reserved in the beginning of the COMMON block for the integers IFI, IGI, and IHI. If, in the main program, each of these counters is initialized to zero, and, in each subprogram these counters are incremented, one can easily keep track of how many times each subprogram has been called.

Subroutines. The central subroutine in GLOBALMIN is GLOBE. It calls various other subroutines which we now describe briefly.

JOIN forms an interval number from two real numbers.

MCHK determines whether a given interval contains zero.

SQUARE, SQURT, and CUBE find the square, square root, and cube, respectively, of an interval number.

CHK examines a box to see if it is less than EA in width. Any such box in AIA is moved to AIB unless it is degenerate (a single point). In the latter case, the box is moved to AIC.

SPLIT divides a box into two pieces and places each in AIA.

MONO examines the interval gradient. If a component is positive or negative over a box, then f(x) is monotonic in the corresponding direction and cannot have a stationary point in the box. Therefore, the box is eliminated.

COEF finds the coefficients of the interval quadratic equation needed by ROOTS.

ROOTS solves the interval quadratic equation and deletes points of the current box where f(x) > FBAR.

NEWTON does a step of an interval Newton method to find stationary points of the gradient of f(x) in the current box.

LINV1F is an IMSL subroutine used to compute the approximate inverse of the center of the Jacobian.

UPDATE evaluates f(x) at the center of each new box and updates FBAR (the smallest known value of f(x)).

DEBUG prints various information useful in debugging the program or in evaluating its performance.

PRINT1 is used by DEBUG to write appropriate information.

SECT1 is called by ROOTS to obtain the intersection of two intervals.

SECT2 is called by ROOTS to obtain the intersection of an interval with the complement (exterior) of an interval.

Note that each subroutine contains comments describing its function and use.

APPENDIX I

Implementation of the Fortran Precompiler, CLUDGE, for the IBM 360/67

By

Carol A. Clarke

June, 1971

IMPLEMENTATION of the FØRTRAN PRECØMPILER, CLUDGE, for the IBM 360/67

Table of Contents

Introduction	1
Changes in CLUDGE	2
Prefix for Type Other	2
Storage Requirement for Type Other	2
Copy Character	2
Files Used by CLUDGE	2
Summary Printed by CLUDGE	2
Type Statement	2
Putting Comments on Instructions to CLUDGE	3
1108 Control Cards	3
Octal Constants	3
Dollar Sign Operator	3
The Symbol Table	3
Section One Entries	4
Section Two Entries	4
Dimensioned Variables	4
N^{th} Continuation Block	4
Unsubscripted Variables	5
Referenced Functions	5
Common Block Names	5
Arithmetic Statement Functions	6
Function Subprograms	6
Section Three and Four Entries	6
Program Size	7
Common Block SCANN	7
Common Block IMPBLK	7
Assigned GØ TØ'S	7
ENCØDE / DECØDE	7
1108 FLD Function	8
Disclaimer	9

Introduction

This report is designed to serve as a supplement to:

F. D. Crary and T. D. Ladner, "A Simple Method of Adding a New Data Type to Fortran," The University of Wisconsin, Mathematics Research Center, U.S. Army, Technical Summary Report #1065, May, 1970.

A description is given here of the changes made in the Fortran pre-compiler, CLUDGE, when it was implemented for the IBM 360/67 at Washington State University. Throughout this report, numbers in square brackets refer to page numbers in the report mentioned above.

Prefix for Type Other

The default prefix for type other in CLUDGE has been changed from ØOØ (oh-zero-oh) to INT. [5, 15, 113].

Storage Requirement for Type Other

The default storage requirement for a type other variable has been changed from one memory word to two. [14, 16].

Copy Character

The default copy character has been changed from a slash (/) to a plus (+). [11, 15].

Files Used by CLUDGE

The data set reference number for SCRATCH2 has been changed from 21 to 1. [12, 16].

Summary Printed by CLUDGE

The summary printed out by CLUDGE at the end of the instructions to CLUDGE no longer includes section one of the symbol table. [16].

Type Statement

The default type statement recognized by CLUDGE for the new data type has been changed from "ØTHER" to "iNTRVAL." [13].

Putting Comments on the Instructions to CLUDGE

The character which stops the examination of an instruction to CLUDGE has been changed from the dollar sign ($) to the "at" sign (@). [13, 23].

1108 Control Cards

The statements in CLUDGE which process 1108 control cards have been removed. [12, 119].

Octal Constants

The statements in CLUDGE which allow for octal constants have been removed. [21].

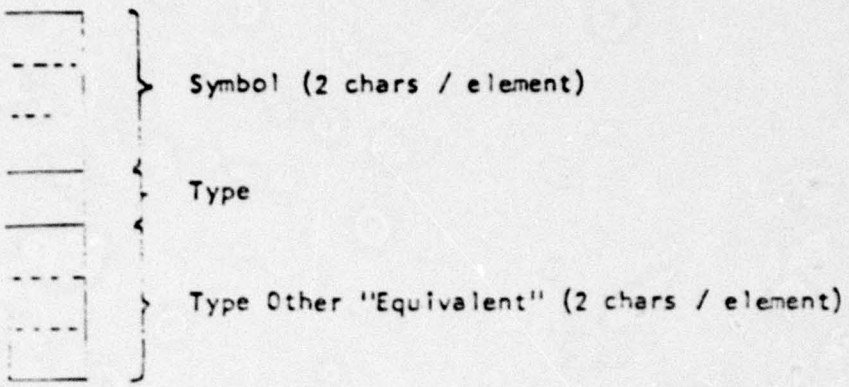
Dollar Sign Operator

Operator #7 has been changed from the dollar sign ($) to the "at" sign (@). [33, 46, 49, 50, 59, 60, 61, 62, 66, 74].

The Symbol Table

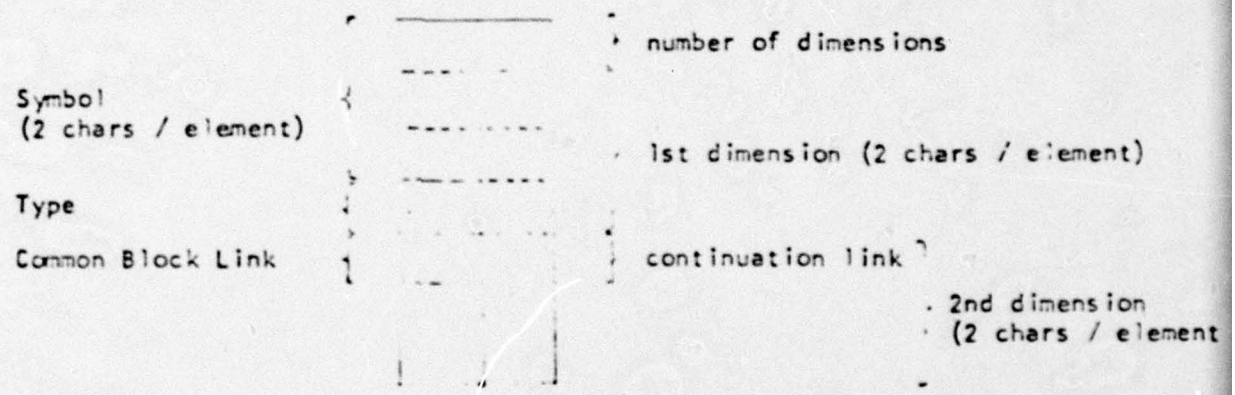
The physical structure of the symbol table had to be changed since 360 Fortran allows at most 4 characters to be stored in one word, not 6. Thus, array TABLE in common block SYMBOL was changed from a 1000 x 3 fullword array to a 7 x 1000 halfword array. The various items are now stored in the symbol table as follows:

Section One Entries [36]:

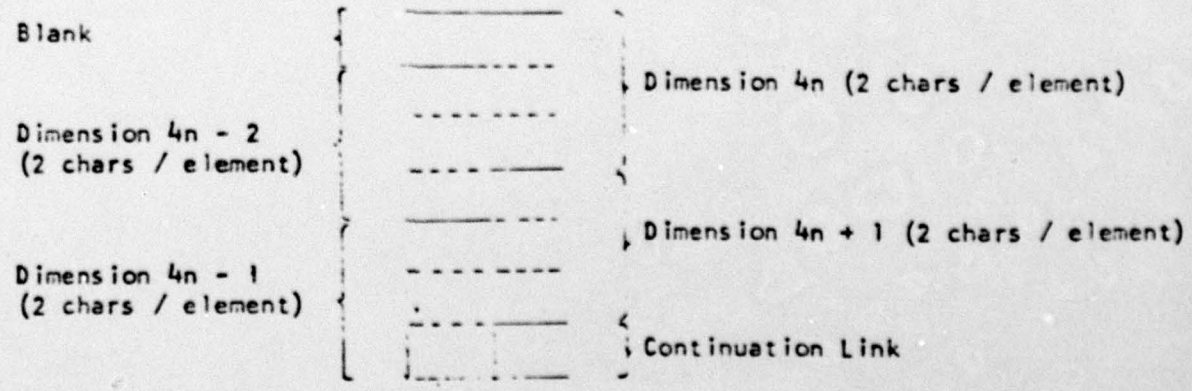


Section Two Entries [36]:

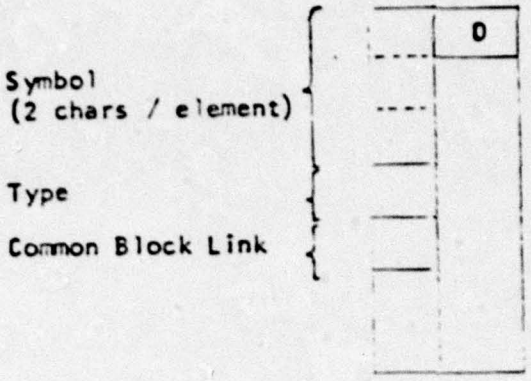
Dimensioned Variable [36]:



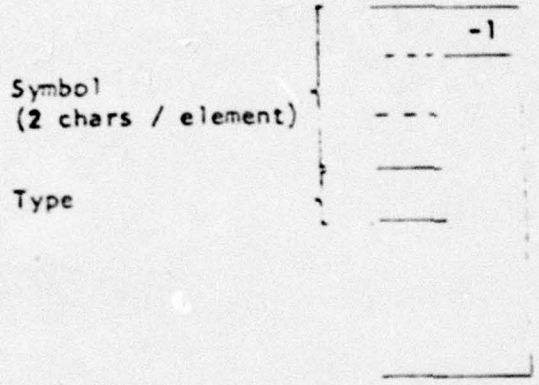
Nth Continuation Block [37]:



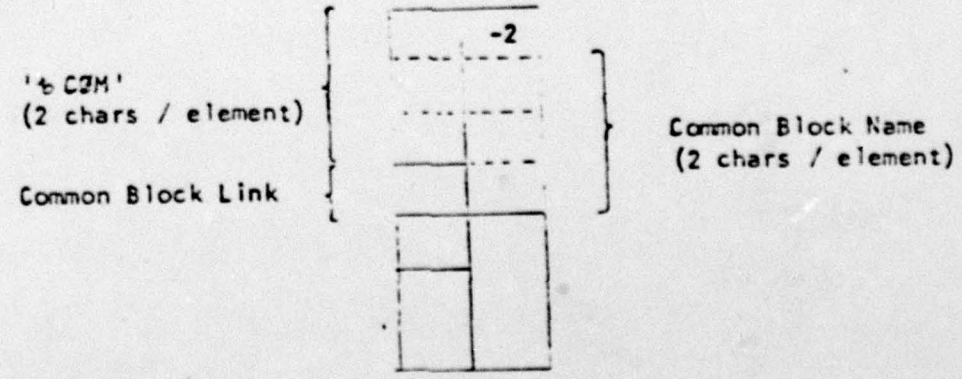
Unsubscripted Variable [37]:



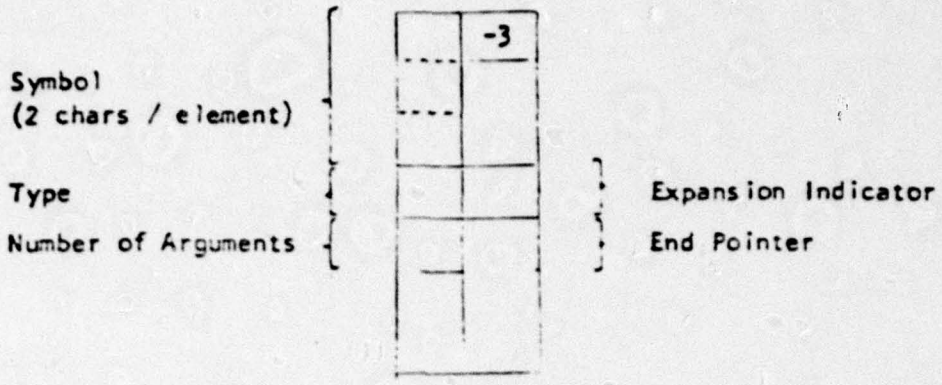
Referenced Function [38]:



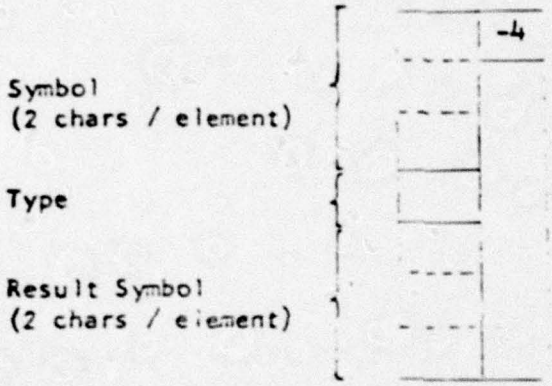
Common Block Name [38]:



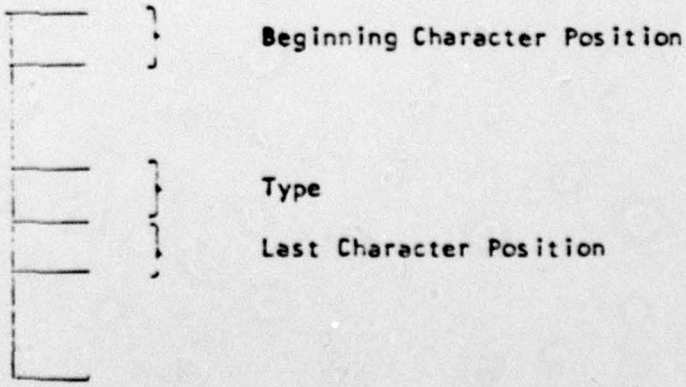
Arithmetic Statement Function [39]:



Function Subprogram [39]:



Section Three and Four Entries [40]:



feature. Other routines using this feature, DMCØDE and DMSTCD were deleted. [117].

1108 FLD Function

To facilitate character and string handling, three Assembler routines, PUTC, GETC, and KØMPAR are used to replace the 1108 FLD function. [118].

Disclaimer

The parts of CLUDGE not mentioned above were left essentially the same as in the 1108 version. Since no claim was given to the correctness of the 1108 version, no guarantee can be given as to the correctness of the 360 version, either. However, the changes made have been thoroughly tested, and seem to be working correctly.

APPENDIX 2

Interval Arithmetic Package for the IBM 360/67

By

Carol A. Clarke

June, 1971

INTERVAL ARITHMETIC PACKAGE FOR THE IBM 360/67

Table of Contents

Introduction 1

I. The Concept of Interval Arithmetic 2

II. Adding a New Data Type to Fortran 2

III. The Implementation of Interval Arithmetic 4

A. Basic Interval Arithmetic 5
B. Best Possible Floating Point Arithmetic 5
C. Conversion Between Data Types 6
D. Exponentiation 8
E. Comparison of Intervals 9
F. Standard Functions 10
G. Supporting Functions 11
H. Error Conditions 12
I. Input and Output 14

References 16

Appendices

A. Routines in the Interval Arithmetic Package A-1
B. Interval Library Routines B-1
C. How to Use CLUDGE C-1
D. Error Control D-1
E. Error Messages from CLUDGE E-1

Introduction

Anyone who has ever tried to do numerical calculations on a computer is probably painfully aware that his results are at best only approximations to real solutions. Many types of error generally contribute to this phenomenon, including error in representation of data within the computer and error inherent in the way the computer does its arithmetic, i.e., "roundoff" error. Through the use of interval arithmetic, the effects of these two types of error may be softened somewhat. Ramon Moore [1] gives some interesting details on the use of interval analysis in numerical methods.

The basic idea behind the Interval Arithmetic Package is that operations can be performed on intervals of real numbers, rather than on the real numbers themselves. These operations are done in such a way that the resulting interval contains the exact solution, and (hopefully) is the smallest machine representable interval containing the exact solution which can be calculated from the data given.

This report explains the basic capabilities of this Interval Arithmetic Package and the use of a Fortran precompiler, CLUDGE (pronounced with a long "u") which allows Fortran variables to by typed "INTRVAL" and used just as standard variables are used in Fortran expressions.

I. The Concept of Interval Arithmetic

Arithmetic operations defined on intervals are merely operations defined on the two sets of real numbers involved. For example, addition of two intervals, $[a,b]$ and $[c,d]$, is the set of sums which may be obtained by adding real numbers, x and y, where $a \leq x \leq b$ and $c \leq y \leq d$. That is to say,

$$[a,b] + [c,d] = \{x + y: \ x \in [a,b] \text{ and } y \in [c,d]\}.$$

In a similar manner, functions may be defined on intervals. Thus for a real valued function f, $f([a,b])$ is the interval $[c,d]$ consisting of all points y such that $y = f(x)$ for some x, $a \leq x \leq b$.

The set notion is also used to define comparisons of intervals. For example, $[a,b] \leq [c,d]$ means that

$$x \leq y \text{ for all } x \in [a,b] \text{ and for all } y \in [c,d].$$

That is, "$\leq$" must be true for all possible pairs, x and y, where x is taken from $[a,b]$ and y is taken from $[c,d]$.

It should be noted that an interval of the form $[a,a]$ is simply another way of representing the real number a. Such an interval is called degenerate. So, in a sense, interval arithmetic can be thought of as an extension of real arithmetic.

II. Adding a New Data Type to Fortran

Standard Fortran compilers allow programmers to use several different data types, such as REAL, INTEGER, COMPLEX, and DOUBLE PRECISION, when performing numerical calculations. In order to add a new data type to Fortran, the programmer must somehow simulate operations on this new

data type in terms of existing operations and data types.

A convenient way of adding a new data type to Fortran is to use a "precompiler" which will recognize a type statement for the new data type, and then translate all arithmetic operations, function calls, comparisons, and data type conversions involving the new data type into calls to the appropriate simulation routines. For example, a precompiler would accept the statements

```
INTRVAL  A,B,C
:
C = A + B
```

and replace them with

```
COMPLEX  A,B,C
:
CALL INTADD (A,B,C)
```

where INTADD is the routine which performs addition on the intervals A and B, and stores the resulting interval in C. (The reason for replacing the INTRVAL type statement with a COMPLEX type statement will be explained later.) The Fortran program produced by the precompiler is then passed to the Fortran compiler to be compiled and executed.

The routines which simulate operations on intervals are contained in the Interval Arithmetic Package, and are explained further in the following sections. A precompiler, named CLUDGE, is available which will recognize the INTRVAL type statement and insert calls to these interval arithmetic routines. Control cards and the deck setup for using CLUDGE are given in Appendix C.

Note that routines in the Interval Arithmetic Package may be called explicitly from the user's program. In the last example, the user could

put the statements

```
COMPLEX  A,B,C
  :
  :
CALL INTADD (A,B,C)
```

directly into his program, and then a precompiler is not needed. The precompiler CLUDGE has been implemented only to make the use of the Interval Arithmetic Package easier.

III. The Implementation of Interval Arithmetic

The following sections describe the methods used in the Interval Arithmetic Package to implement the various interval operations. The implementation preserves the concept of interval arithmetic as it is described in Section I.

Within the computer, an interval of real numbers will be represented by its endpoints. We will thus refer to an interval $[a,b]$ as an ordered pair of machine representable single precision numbers, a and b, such that $a \leq b$. Hence in order to represent an interval by a single variable in Fortran, the variable name must refer to two memory cells, one for the left endpoint and one for the right endpoint. This can be accomplished in either of two ways: by declaring the variable COMPLEX or by declaring it to be a real array of two elements, i.e., COMPLEX A or DIMENSION A(2). If the user is calling routines in the Interval Arithmetic Package directly, he may use either method. If he is using the precompiler, CLUDGE, he must use the INTRVAL type statement as described in the last section.

We can now go on to describe the various interval operations in terms of the representations of the intervals by endpoints.

A. Basic Interval Arithmetic

The four basic binary operations, addition, subtraction, multiplication, and division are defined for intervals as follows:

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

$$[a,b] - [c,d] = [a - d,\ b - c]$$

$$[a,b] \times [c,d] = [\min\{ac,ad,bc,bd\},\ \max\{ac,ad,bc,bd\}]$$

$$[a,b] / [c,d] = [\min\{a/c,\ a/d,\ b/c,\ b/d\},\ \max\{a/c,a/d,b/c,b/d\}]$$

(note: division is defined only if $0 \notin [c,d]$.)

Note that these definitions are consistent with the general set definitions given in Section I.

The subroutines in the Interval Arithmetic Package which perform these operations are INTADD, INTSUB, INTMUL, and INTDIV, respectively. Appendix A describes how these routines may be called directly. If CLUDGE is being used, calls to these routines will be generated whenever the symbols +, -, *, or / are encountered and either or both operands are of type INTRVAL. (Note that mixed mode _is_ allowed by CLUDGE. Conversions between data types are described in Section III. C.) CLUDGE ignores a unary + sign, and unary negation is performed by the routine INTNEG.

For further information on the theory and algorithms used for this interval arithmetic, see [1] and [2].

B. Best Possible Floating Point Arithmetic

While the interval operations defined in the last section are theoretically correct, they cannot simply be programmed using the computer's binary operations, since this would again introduce the error we are trying to avoid. Consider the following example:

Suppose a hypothetical computer can retain only one digit after every operation. Then [.8, .9] might be a typical interval. The square of this interval would theoretically be [.64, .81], but this would be represented on this computer by [.6, .8]. Now since our original interval contains the number .899, we require that $(.899)^2 = .808201$ be contained in the interval [.6, .8], but obviously it is not.

Note that [.5, 1.0] would be an adequate representation for $[.8, .9]^2$, but this is not the smallest such interval possible. If we "round up" the right endpoint and "round down" the left endpoint of [.64, .81] we get [.6, .9] which is the smallest adequate interval representable on our hypothetical computer.

The Interval Arithmetic Package uses a set of routines with "Best Possible Floating Point Arithmetic" to accomplish the desired bounding of intervals. Essentially these routines calculate the result of a floating point operation to more precision than single precision, and then "round up" or "round down" as desired. The result is the largest machine representable single precision number less than the exact result, or the smallest machine representable single precision number greater than the exact result, as the case may be. For further discussion of these algorithms, see [3] and [6]. Case analyses for the corresponding interval routines may be found in [2].

C. Conversion Between Data Types

The Interval Arithmetic Package contains routines which will convert integer, real, complex, and double precision numbers to intervals (i.e., to the computer's endpoint-representation of an interval) and vice versa. A description of each conversion is given below. The name given in parentheses is the name of the routine in the Interval Arithmetic Package which does the conversion.

1. Real to Interval (INTC58):

   A degenerate interval is created whose endpoints are the given real number.

2. Double Precision to Interval (INTC68):

   If the double precision number can be expressed exactly in single precision, the conversion is the same as in 1. If not, a nondegenerate interval is obtained by rounding down on the double precision number for the left endpoint, and rounding up for the right endpoint.

3. Integer to Interval (INTC48):

   The integer is first converted to double precision, and then to an interval as described in 2.

4. Complex to Interval (INTC78):

   The real part of the complex number is converted to an interval as described in 1. The imaginary part is ignored.

5. Interval to Real, Double Precision, or Integer (INTC85, INTC86, INTC84):

   The midpoint of the interval is computed in double precision and is then converted to the desired type, rounded if necessary to single precision.

6. Interval to Complex (INTC87):

   The real part is set to the rounded value of the midpoint of the interval, and the imaginary part is set to zero.

When using CLUDGE, the programmer may mix variables of type INTRVAL with those of type REAL, DOUBLE PRECISION, INTEGER, or COMPLEX. Type INTRVAL dominates the other data types. Thus mixed mode arithmetic involving a variable of type INTRVAL will be done in interval arithmetic. CLUDGE will generate calls to the various conversion routines whenever necessary. For example, suppose R is real, and A is type INTRVAL. Then during precompilation with CLUDGE, the statement

```
R = 2 * A
```

is replaced by

```
REAL INTC85
COMPLEX A, INTTEM(2)
:
CALL INTC48(2,INTTEM(1))
CALL INTMUL (INTTEM(1),A,INTTEM(2))
R = INTC85(INTTEM(2))
```

Note that CLUDGE will allocate temporary storage locations in the array INTTEM to hold intermediate interval results whenever needed.

The arithmetic IF statement is the one exception to the fact that the INTRVAL data type always dominates. If the expression in an arithmetic IF statement contains a variable of type INTRVAL, CLUDGE will generate the statements necessary to evaluate the expression, just as explained above, but the final result is converted to type REAL and the proper branch is determined on the basis of this REAL value. For example, for the statements

```
INTRVAL A
:
IF (A) 3,4,5
```

CLUDGE will generate

```
REAL INTC85
COMPLEX A
:
IF (INTC85(A)) 3,4,5
```

The user may also call these conversion routines directly from his program if desired. (See Appendix A).

D. Exponentiation

There are four routines in the Interval Arithmetic Package for raising an interval to a power:

1. Interval to Integer Power (INTXP4):

   The result is obtained by successive multiplications of the interval with itself. However, if the power is an even integer, the left endpoint of the result will never be less than zero. For example, if A is the interval $[-1,2]$, then the result of squaring A will be $[0,4]$, even though A multiplied by itself yields $[-2,4]$.

2. Interval to Real, Double Precision, or Interval Power (INTXP5, INTXP6, INTXP3):

   The result of A^B is calculated by exp(Bxlog(A)) where the interval exponential and log routines (described in Section III. F.) are used, and B is converted to an interval beforehand, if necessary.

3. Real, Double Precision, or Integer to Interval Power (INTXP8):

   The base is first converted to an interval, and then the routine INTXP8 is used.

The precompiler CLUDGE will process occurrences of ** in a Fortran program by generating calls to the appropriate routines, depending on the type of base and exponent. If CLUDGE is not used, the user must see that proper conversions are done.

E. Comparison of Intervals

The Interval Arithmetic Package contains logical functions INTEQ, INTNE, INTGT, INTGE, INTLT, INTLE which perform the comparisons .EQ., .NE., .GT., .GE., .LT., and .LE. on two intervals. These relationships are defined as follows:

$[a,b] = [c,d]$ if and only if $a=b=c=d$

$[a,b] \neq [c,d]$ if and only if $b < c$ or $a > d$

$[a,b] < [c,d]$ if and only if $b < c$

$[a,b] \leq [c,d]$ if and only if $b \leq c$

$$[a,b] \succ [c,d] \text{ if and only if } \quad a \succ d$$

$$[a,b] \succeq [c,d] \text{ if and only if } \quad a \succeq d$$

These are consistent with the general definitions in Section I. In particular, note that if R is any one of the above relational operators, then $[a,b]\ R\ [c,d]$ if and only if $x\ R\ y$ for all $x \in [a,b]$ and $y \in [c,d]$. For example, if $x \in [1,3]$ and $y \in [2,4]$ it is not clear how x and y are related. Hence the relational operators can be defined only on disjoint intervals, or intervals with at most one point in common. Care must be taken when using these operators since, for example, $[a,b] \not\prec [c,d]$ does <u>not</u> imply that $[a,b] \succeq [c,d]$. Even more confusing is an expression such as

```
.NOT.(A.EQ.B).AND..NOT.(A.NE.B)
```

which is <u>true</u> if A and B are nondegenerate intersecting intervals, even if they are equal as sets!

These routines may be referenced directly by the user (see Appendix A), or, they will be referenced by CLUDGE whenever one of the relational operators .EQ., .NE., .GT., .GE., .LT., or .LE. is encountered with a type INTRVAL operand. (If the other operand is <u>not</u> of type INTRVAL, it is converted before the comparison is made.)

F. Standard Functions

The Interval Arithmetic Package provides interval counterparts to the standard Fortran functions SQRT, EXP, ALOG, ALOG10, ARSIN, ARCOS, ATAN, SINH, COSH, TANH, COS, SIN, TAN, and ATAN2. These routines are listed in Appendix A. Thus, if desired, the user may call them directly. If CLUDGE is being used, calls to the interval counterparts will be made whenever one of the above routines is encountered with a type INTRVAL argument.

These interval routines utilize the standard double precision functions, and hence although there is a high probability of accuracy, correct bounding here cannot be guaranteed. Appendix B gives the number of bits which are assumed to be accurate for each of the double precision standard functions used. The problem arises when a carry, generated below the last assumed accurate bit in the double precision result, propagates into some of the correct bits, causing erroneous bounding of the interval. Consider the following simple example:

> Suppose the exact value of a function is the binary number 01111. If we use the corresponding Fortran library function, and only the first three bits are accurate, we could get an error of 00001 and our result would be 01111 + 00001 = 10000. Now if we truncate this to three bits we get 100 as a left endpoint, when in fact the left endpoint should be 011.

To prevent this, the result of the double precision function is actually bounded twice: once at the last assumed accurate bit in the least significant part of the number, and once again to create the proper single precision endpoint of the interval. So in the above example, to create a left endpoint from 10000, knowing that only the first three bits are accurate, we first subtract 00100 to get 01100, and then truncate that to three bits to get 011, which is the proper left endpoint.

The resulting interval may not be as small as possible as a result of this process, but it is most probably accurate. For further details on the error analysis of the Fortran double precision routines, see [4].

G. Supporting Functions

Two interval routines which have Fortran counterparts have yet to be mentioned. They are INTABS, which computes the absolute value of an interval, and INTSTR, which replaces one interval by another.

In addition, there are six interval routines which have no Fortran counterparts, and thus will never be referenced by CLUDGE. They are:

INTSUP	Returns the right endpoint of an interval
INTINF	Returns the left endpoint of an interval
INTLEN	Returns the length of an interval
INTUON	Returns the union of two intervals: $[a,b] \cup [c,d] = [\min(a,c), \max(b,d)]$
INTSCT	Returns the intersection of two intervals: $[a,b] \cap [c,d] = [\max(a,c), \min(b,d)]$ (The intersection of disjoint intervals is not defined)
INTBND	Returns an interval created by bounding a double precision number to a given accuracy

These routines are also listed in Appendix A, and may be called directly by the user.

(Note: There is no routine in the Interval Arithmetic Package which will create an interval from two real numbers. However, this may be accomplished as follows:

1. Convert each real number to a degenerate interval.
2. Form the union of these two intervals.

To create an interval from a double precision number, use the subroutine INTBND and specify how many bits of the double precision number are assumed to be accurate.)

H. Error Conditions

During every interval operation performed by a routine in the Interval Arithmetic Package, possible error conditions are monitored and a flag is set whenever an error occurs. At the end of each operation, a call is made to INTRAP, a routine in the Interval Arithmetic Package which takes appropriate action if an error has occurred. Below is a list of the possible

error conditions, along with the value of the result in each case, and the action taken by INTRAP. The meaning of overflow and underflow should be clear. An "infinity" error occurs when an attempt is made to round up (down) a number which is already at least as large (small) as the largest (smallest) single precision number possible. The symbol K is used to represent the largest possible single precision number, L is used to represent the largest possible single precision number which can be correctly converted to an integer, and C indicates a correct value. The possible actions taken by INTRAP are:

A	No Action Taken
B	Prints Error Message
C	Prints Error Message and Traceback, and Increments the Error Counter

The fault conditions recognized by INTRAP are:

1. Bounds Faults:

Left Endpoint	Right Endpoint	Result	Action
no fault	infinity	[C,K]	C
no fault	underflow	[C,0]	A
overflow	infinity	[K,K]	C
-infinity	no fault	[-K,C]	C
-infinity	overflow	[-K,K]	C
-infinity	infinity	[-K,K]	C
-infinity	underflow	[-K,0]	C
underflow	no fault	[0,C]	A
underflow	infinity	[0,K]	C
underflow	underflow	[0,0]	A

2. Other Faults

Fault	Result	Action
Division by Zero	undefined	C
Zero to the Zero Power	[1,1]	B
Square Root of Negative Number	[-K,K]	C
Log of Non-Positive Number	[-K,K]	C
Underflow during Interval to Real Conversion	0	A
Overflow during Interval to Integer Conversion	L	C
Intersection of Disjoint Intervals	undefined	C
ARCOS or ARSIN Argument Out of Range	[-K,K]	C

Provisions have also been made to allow the user some control over the action taken when errors occur. These are explained in more detail in Appendix D.

I. Input and Output

The Interval Arithmetic Package contains no special routines for input and output of interval variables, so the user must devise his own methods. This is a straightforward task and no more difficult than the task of input and output for ordinary Fortran variables as long as the user remembers two things:

1. Interval variables require two memory cells and hence must be treated as either complex variables or arrays of two elements. If they are declared as arrays, they may be read in or written out just as ordinary arrays are. If they are declared COMPLEX, they may be read in or written out just as complex variables are. If CLUDGE is used, all variables of type INTRVAL are automatically declared COMPLEX and should be treated so in all input and output statements.

2. Fortran FORMAT statements allow the user to specify the number of digits to be written out, and so what is actually written is the value rounded to that many places. Care should be taken to insure that this feature does not unintentionally misrepresent the actual endpoints of the interval computed. That is, rounding up on the left endpoint or down on the

right endpoint could cause the interval to appear smaller than it actually is.

Note:

Any appearance of a variable of type INTRVAL in a READ or WRITE statement causes the message

****** WARNING - TYPE OTHER ARGUMENT IN I/O LIST ******

to be printed. This message is simply a reminder to the user that a type INTRVAL variable is being used and must have two fields reserved for it in the FORMAT statement. The message does not indicate an error in syntax, but merely points out a frequent source of I/O errors.

References

[1] Ramon E. Moore, Interval Analysis, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1966.

[2] Donald I. Good and Ralph L. London, "Interval Arithmetic for the Burroughs B-5500: Four Algol Procedures and Proofs of Their Correctness," The University of Wisconsin Computer Sciences Department, Technical Report #26, June, 1968.

[3] J. M. Yohe, "Best Possible Floating Point Arithmetic," The University of Wisconsin, Mathematics Research Center, U.S. Army, Technical Summary Report #1054, March, 1970.

[4] "IBM System/360 FORTRAN IV Library: Mathematical and Service Subprograms," Form C28-6818-0.

[5] F. D. Crary and T. D. Ladner, "A Simple Method of Adding a New Data Type to Fortran," The University of Wisconsin, Mathematics Research Center, U.S. Army, Technical Summary Report #1065, May, 1970.

[6] C. A. Clarke, "Implementation of Best Possible Floating Point Arithmetic for the IBM 360/67," June, 1971. (Available from the Washington State University Computing Center Library)

[7] C. A. Clarke, "Implementation of the Fortran Precompiler, CLUDGE, for the IBM 360/67," June, 1971. (Available from the Washington State University Computing Center Library)

Appendix A:

Routines in the Interval Arithmetic Package

In the following table, the NAME column gives the value of the parameter NAME in common block INTFLT when INTRAP is called by that routine (see Appendix D). In the column marked "Arguments" the type of each argument is implied by the first letter as follows:

First Letter	Type
A	Interval
C	Complex
D	Double Precision
I	Integer
L	Logical
R	Real

NAME	Operation	Routine Name and Type	Arguments	Fortran Equivalent	Effect
	Arithmetic:				
1	Addition	INTADD Subroutine	(A1,A2,ARES)	+	ARES = A1 + A2
2	Subtraction	INTSUB Subroutine	(A1,A2,ARES)	-	ARES = A1 - A2
3	Multiplication	INTMUL Subroutine	(A1,A2,ARES)	*	ARES = A1 * A2
4	Division	INTDIV Subroutine	(A1,A2,ARES)	/	ARES = A1 / A2
	Exponentiation:				
5	Interval to Integer	INTXP4 Subroutine	(A1,I1,ARES)	**	ARES = A1 ** I1
	Interval to Real	INTXP5 Subroutine	(A1,R1,ARES)	**	ARES = A1 ** R1
	Interval to Dbl.Prec.	INTXP6 Subroutine	(A1,D1,ARES)	**	ARES = A1 ** D1
	Interval to Interval	INTXP8 Subroutine	(A1,A2,ARES)	**	ARES = A1 ** A2
	Conversions:				
6	Integer to Interval	INTC48 Subroutine	(I1,ARES)		ARES = I1
	Real to Interval	INTC58 Subroutine	(R1,ARES)		ARES = R1
7	Dbl.Prec. to Interval	INTC68 Subroutine	(D1,ARES)		ARES = D1
	Complex to Interval	INTC78 Subroutine	(C1,ARES)		ARES = C1
8	Interval to Integer	INTC84 Integer Function	(A1)		IRES = A1
9	Interval to Real	INTC85 Real Function	(A1)		RRES = A1
	Interval to Dbl.Prec.	INTC86 Dbl.Prec. Function	(A1)		DRES = A1
	Interval to Complex	INTC87 Complex Function	(A1)		CRES = A1
	Comparisons:				
	Equals	INTEQ Logical Function	(A1,A2)	.EQ.	LRES = A1.EQ.A2
	Not Equals	INTNE Logical Function	(A1,A2)	.NE.	LRES = A1.NE.A2
	Less Than	INTLT Logical Function	(A1,A2)	.LT.	LRES = A1.LT.A2

			Arguments	Fortran Equivalent	Effect
	Less - Equal	INTLE Logical Function	(A1,A2)	.LE.	RES = A1.LE.A2
	Greater Than	INTGT Logical Function	(A1,A2)	.GT.	RES = A1.GT.A2
	Greater - Equal	INTGE Logical Function	(A1,A2)	.GE.	RES = A1.GE.A2
	Basic External Fns:				
	Absolute Value	INTABS Subroutine	(A1,ARES)	ABS,IABS,DABS	ARES = ABS(A1)
19	Inverse Cosine	INTACS Subroutine	(A1,ARES)	ARCØS,DARCØS	ARES = ARCØS(A1)
18	Inverse Sine	INTASN Subroutine	(A1,ARES)	ARSIN,DARSIN	ARES = ARSIN(A1)
17	Inverse Tangent	INTATN Subroutine	(A1,ARES)	ATAN,DATAN	ARES = ATAN(A1)
26	(two arguments)	INTAT2 Subroutine	(A1,A2,ARES)	ATAN2,DATAN2	ARES = ATAN2(A1,A2)
24	Cosine	INTCØS Subroutine	(A1,ARES)	CØS,DCØS,CCØS	ARES = CØS(A1)
22	Hyperbolic Cosine	INTCSH Subroutine	(A1,ARES)	CØSH,DCØSH,CCØSH	ARES = CØSH(A1)
15	Exponential	INTEXP Subroutine	(A1,ARES)	EXP,DEXP,CEXP	ARES = EXP(A1)
	Integer	INTINT Subroutine	(A1,ARES)	AINT,DINT	ARES = AINT(A1)
14	Natural Logarithm	INTLN Subroutine	(A1,ARES)	ALØG,DLØG,CLØG	ARES = ALØG(A1)
16	Common Logarithm	INTLOG Subroutine	(A1,ARES)	ALØG10,DLØG10,CLØG10	ARES = ALØG10(A1)
23	Sine	INTSIN Subroutine	(A1,ARES)	ASIN,DSIN,CSIN	ARES = SIN(A1)
21	Hyperbolic Sine	INTSNH Subroutine	(A1,ARES)	SINH,DSINH,CSINH	ARES = SINH(A1)
13	Square Root	INTSQT Subroutine	(A1,ARES)	SQRT,DSQRT,CSQRT	ARES = SQRT(A1)
25	Tangent	INTTAN Subroutine	(A1,ARES)	TAN,DTAN,CTAN	ARES = TAN(A1)
20	Hyperbolic Tangent	INTTNH Subroutine	(A1,ARES)	TANH,DTANH,CTANH	ARES = TANH(A1)
	Supporting Fns:				
	Store	INTSTR Subroutine	(A1,ARES)	=	ARES = A1
	Negate	INTNEG Subroutine	(A1,ARES)	- (unary)	ARES = -A1

NAME	Operation	Routine Name and Type	Arguments	Fortran Equivalent	Effect
	Supremum	INTSUP Real Function	(A1)	none	RRES = right endpoint of A1
	Infimum	INTINF Real Function	(A1)	none	RRES = left endpoint of A1
	Length	INTLEN Dbl.Prec. Function	(A1)	none	DRES = LEN(A1)
12	Union	INTUON Subroutine	(A1,A2,ARES)	none	ARES = A1 ∪ A2
11	Intersection	INTSCT Subroutine	(A1,A2,ARES)	none	ARES = A1 ∩ A2
10	Bound	INTBND Subroutine	(D1,I1,ARES)	none	ARES = BND(D1,I1)
	Error Control	INTRAP Subroutine	(IFAULT,INAME, A1,A2,A3,IARG, RARG,DARG)	ERRTRA	

Appendix B:

Interval Library Routines

In the following table, the values in the range column are the approximate extreme values of the interval function. M stands for the maximum single precision floating point number. The number in the "Errors" column is the value of the fault flag for this error condition.

Name	Range	Error	Explanation or Other Comments	Library Routines Used	Assumed Accuracy in # of bits
INTACS	[0,π]	23	Argument out of range: -1 to 1	DARCØS	52
INTASN	[-π/2, π/2]	23	Argument out of range: -1 to 1	DARSIN	52
INTATN	[-π/2, π/2]	--	---	DATAN	52
INTAT2	[-π/2, π/2]	16	Division by zero attempted	DATAN	52
INTCØS	[-1,1]	--	---	DCØS	48
INTCSH	[1,M]	--	If abs. val. of endpoint ≥ 175.0, 175.0 used	DCØSH	51
INTEXP	[0,M]	--	If abs. val. of endpoint ≥ 170.0, 170.0 used	DEXP	52
INTLN	[-181, 175]	19	Log of non-positive number	DLØG	50 except 25 i [.9995, 1.000
INTLØG	[-79,76]	19	Log of non-positive number	DLØG10	50 except 25 i [.9995, 1.000
INTSIN	[-1,1]	--	---	DCØS	48
INTSNH	[-M,M]	--	If abs. val. of endpoint ≥ 175.0, 175.0 used	DSINH	51 except 40 i [-.0005, .00
INTSQT	[0,M]	18	Square root of a negative number	DSQRT	53
INTTAN	[-M,M]	10	Error when interval argument spans the discontinuity of tan at -π/2 or at π/2. The standard function DTAN will also generate an error if either endpoint is "too close" to -π/2 or to π/2. No check for this is made in INTTAN.	DTAN	38
INTTNH	[-1,1]	--	---	DTANH	52

Appendix C:

How to Use CLUDGE

The Fortran precompiler, CLUDGE, will accept almost all Fortran statements allowed by the IBM Fortran-G compiler. The exceptions are given here:

1. CLUDGE will not accept any of the following statements:

```
INTEGER*2
INTEGER*4
REAL*4
REAL*8
COMPLEX*8
COMPLEX*16
LOGICAL*1
LOGICAL*4
```

   CLUDGE does, however, accept the following type statements:

```
REAL
INTEGER
COMPLEX
DOUBLE PRECISION
LOGICAL
INTRVAL
```

   Thus the user should avoid using INTEGER*2, LOGICAL*1, and COMPLEX*16 data types altogether when using CLUDGE, and should replace other "*" type statements by ordinary type statements, i.e., use REAL rather than REAL*4, etc.

2. The following statements are accepted by CLUDGE, but no further action is taken on them:

```
DATA
EQUIVALENCE
EXTERNAL
FORMAT
NAMELIST
```

In particular, extreme caution should be taken if type INTRVAL variables are used in DATA or EQUIVALENCE statements, since the form of these variables is altered by CLUDGE. Also, since FORMAT statements are recognized but not processed further, the user should avoid using FORMAT as the name of an array.

3. The main program must be the first input to CLUDGE, and at most one main program may appear in the deck.

4. The names INTTEM and INTRLT are generated by CLUDGE for special purposes, and should not be explicitly used by the programmer.

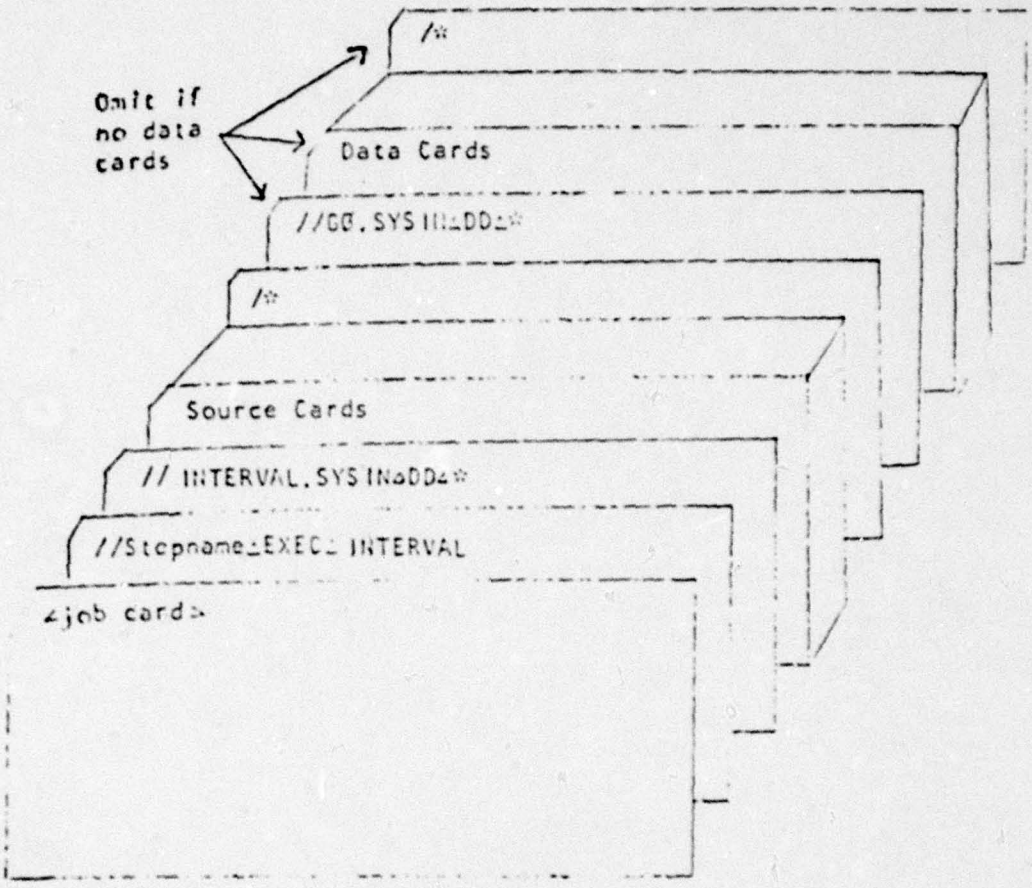
5. If CLUDGE must generate additional statement numbers, they will start with 30000, or the first integer after 30000 which has not already been used. Thus the programmer should attempt to use statement numbers below 30000 whenever possible.

CLUDGE does a minimal amount of error checking, but may occasionally produce error messages. These messages are listed in Appendix E. Note that the type OTHER referred to by these messages is type INTRVAL.

CLUDGE is actually a general purpose precompiler for FORTRAN, and accepts several instructions which will alter the way in which it processes a program. These instructions are described in [5] and [7], but the inexperienced programmer should not attempt to use them.

The control card setup for CLUDGE is given below:

Deck Setup to Precompile, Compile, and GO



Stepname	may be any name of 1 to 8 characters (first character must be alphabetic) or may be left blank.
␣	denotes at least one blank.
Source Cards	are the Fortran program which uses the INTRVAL type statement.

A program run under this control card setup will produce three levels of output:

1. Output from CLUDGE:

   This is a listing of the source deck, along with any error messages which CLUDGE generates.

2. Output from the Fortran Compiler:

   This is a listing of the Fortran program which CLUDGE produces by changing the type INTRVAL statement to COMPLEX and replacing operations on type INTRVAL variables with subroutine calls and function references. Any error messages produced by the Fortran compiler will be given here.

3. Output from Execution of the Program:

   This is the output which is generated by the execution of the compiled Fortran program. Any error messages generated by this execution will be given here.

These three levels of output are illustrated in the sample program which follows.

Source listing with lines of print messages added by CLUDGE.

IBM 360/67 VERSION OF CLUDGE

```
      SAMPLE PROGRAM TO COMPUTE A TABLE OF FACTORIALS AND THEIR NATURAL
      LOGARITHMS USING INTERVAL ARITHMETIC

      REAL INTINF,INTSUP,LEFT
      INTRVAL FACT,LNFACT
      WRITE(6,1)
      FACT=1.
      DO 10 I=1,25
      FACT=FACT*I                                    See p. 15 for explanation
      LNFACT=ALOG(FACT)
      LEFT=INTINF(FACT)
      RIGHT=INTSUP(FACT)
   10 WRITE(6,2)I,LEFT,RIGHT,LNFACT
***** WARNING - TYPE OTHER ARGUMENT IN I/O LIST *****

      NOTE THAT THE PRINTING OF THE VALUES OF THE TWO TYPE INTRVAL
      VARIABLES IS PROGRAMMED IN DIFFERENT WAYS

    1 FORMAT('1TABLE OF INTERVAL FACTORIALS AND THEIR LOGARITHMS',///,
     1'  N',16X,'N FACTORIAL',22X,'LOG(N FACTORIAL)',//)
    2 FORMAT(' ',I2,2(4X,'I',E15.8,',',E15.8,')'))
      CALL EXIT
      END
```

Program produced by CLUDGE for compilation by FORTRAN compiler.

```
LEVEL  18                    MAIN                    DATE = 71166          15/2
C                ***** PROCESSED BY IBM 360/67 VERSION OF CLUDGE *****
      COMPLEX INTTEM(2)
      REAL INTINF, INTSUP, LEFT
      COMPLEX FACT, LNFACT
C      SAMPLE PROGRAM TO COMPUTE A TABLE OF FACTORIALS AND THEIR NATU
C      LOGARITHMS USING INTERVAL ARITHMETIC
C
      WRITE(6,1)
      CALL INTC58 (1., FACT)
      DO    10 I=1,25
      CALL INTC48 (I, INTTEM(1))
      CALL INTMUL (FACT, INTTEM(1), INTTEM(2))
      CALL INTSTR (INTTEM(2), FACT)
      CALL INTLN (FACT, LNFACT)
      LEFT = INTINF (FACT)
      RIGHT = INTSUP (FACT)
   10 WRITE(6,2) I,LEFT,RIGHT,LNFACT
C  ***** WARNING - TYPE OTHER ARGUMENT IN I/O LIST *****
C
C      NOTE THAT THE PRINTING OF THE VALUES OF THE TWO TYPE INTRVAL
C      VARIABLES IS PROGRAMMED IN DIFFERENT WAYS
C
    1 FORMAT('1TABLE OF INTERVAL FACTORIALS AND THEIR LOGARITHMS',///,
     *' N',16X,'N FACTORIAL',22X,'LOG(N FACTORIAL)',//)
    2 FORMAT(' ',I2,2(4X,'(',E15.8,',',E15.8,')'))
      CALL EXIT
      END
```

Output from program execution.

OF INTERVAL FACTORIALS AND THEIR LOGARITHMS

N	FACTORIAL	LOG(N FACTORIAL)
(	0.1000000E 01, 0.1000000E 01)	(0.0 , 0.0)
(	0.2000000E 01, 0.2000000E 01)	(0.69314712E 00, 0.69314718E 00)
(	0.6000000E 01, 0.6000000E 01)	(0.17917585E 01, 0.17917595E 01)
(	0.2400000E 02, 0.2400000E 02)	(0.31780529E 01, 0.31780539E 01)
(	0.1200000E 03, 0.1200000E 03)	(0.47874908E 01, 0.47874918E 01)
(	0.7200000E 03, 0.7200000E 03)	(0.65792503E 01, 0.65792513E 01)
(	0.5040000E 04, 0.5040000E 04)	(0.85251608E 01, 0.85251617E 01)
(	0.4032000E 05, 0.4032000E 05)	(0.10604603E 02, 0.10604604E 02)
(	0.3628800E 06, 0.3628800E 06)	(0.12801827E 02, 0.12801828E 02)
(	0.3628800E 07, 0.3628800E 07)	(0.15104412E 02, 0.15104413E 02)
(	0.3991680E 08, 0.3991680E 08)	(0.17502304E 02, 0.17502319E 02)
(	0.4790016E 09, 0.4790016E 09)	(0.19987213E 02, 0.19987228E 02)
(	0.6227018E 10, 0.6227020E 10)	(0.22552155E 02, 0.22552170E 02)
(	0.8717821E 11, 0.8717834E 11)	(0.25191208E 02, 0.25191223E 02)
(	0.1307672E 13, 0.1307676E 13)	(0.27899261E 02, 0.27899277E 02)
(	0.2092277E 14, 0.2092282E 14)	(0.30671844E 02, 0.30671875E 02)
(	0.3556874E 15, 0.3556879E 15)	(0.33505066E 02, 0.33505081E 02)
(	0.6402373E 16, 0.6402387E 16)	(0.36395432E 02, 0.36395462E 02)
(	0.1216451E 18, 0.1216454E 18)	(0.39339874E 02, 0.39339890E 02)
(	0.2432902E 19, 0.2432908E 19)	(0.42335602E 02, 0.42335632E 02)
(	0.5109408E 20, 0.5109418E 20)	(0.45380127E 02, 0.45380142E 02)
(	0.1124000E 22, 0.1124003E 22)	(0.48471176E 02, 0.48471191E 02)
(	0.2585199E 23, 0.2585209E 23)	(0.51606659E 02, 0.51606689E 02)
(	0.6204464E 24, 0.6204501E 24)	(0.54784714E 02, 0.54784744E 02)
(	0.1551111E 26, 0.1551125E 26)	(0.58003601E 02, 0.58003616E 02)

Appendix D:

Error Control

Below is a complete table of the faults which are recognized by INTRAP. The value of the result after each fault has already been given in Section III.H.

Fault flag	Fault Condition		Default Response
	left endpoint	right endpoint	
0	no fault	no fault	--
1	no fault	overflow	4*
2	no fault	infinity	3
3	no fault	underflow	0
4	overflow	no fault	4*
5	overflow	overflow	4*
6	overflow	infinity	3
7	overflow	underflow	4*
8	-infinity	no fault	3
9	-infinity	overflow	3
10	-infinity	infinity	3
11	-infinity	underflow	3
12	underflow	no fault	0
13	underflow	overflow	4*
14	underflow	infinity	3
15	underflow	underflow	0
16	division by zero		3
17	zero to the zero power		1
18	square root of a negative number		3
19	log of a non-positive number		3
20	underflow during interval-to-real		0
21	overflow during interval-to-integer		3
22	intersection of disjoint intervals		3
23	ARCØS or ARSIN argument out of range		3

* cannot logically occur

The action taken by INTRAP is determined by the value of the array MONTOR corresponding to the value of the fault flag:

MONTOR(IFAULT)	Action
0	Exit
1	Print error message
2	Print error message and traceback
3	Print error message, traceback, and step error counter
4	Print error message, traceback, and stop

The user may communicate directly with INTRAP through common block INTFLT. The statement needed is

```
      COMMON/INTFLT/IFAULT,NAME,PARAMS(2,3),IPAR,PAR,DPAR,MONTOR(23),
     *              KNTR(23),KOUNTS(23)
```

where IFAULT is the value of the fault flag, NAME indicates the routine in which the error occurred (see Appendix A for the values of NAME), the array PARAMS contains the interval arguments (up to three of them, stored as columns), IPAR contains the possible integer argument, PAR the possible real argument, and DPAR the possible double precision argument. Obviously, the type of arguments passed depends on the type of arguments of the routine in which the error occurred. MONTOR gives the default response for each of the possible 23 errors. The values in this array may be altered dynamically by the user to change the action taken by INTRAP. For example,

```
MONTOR(16) = 1
```

causes INTRAP to simply print an error message when division by zero (fault #16) is attempted. The array KNTR gives the maximum number of times each error may occur before execution is terminated. This array may also be altered dynamically. Thus the statement

```
KNTR(18) = 4
```

allows taking the square root of a negative number to be attempted four

times before the job is halted. The default value for this array is 1 for each error. The array KØUNTS contains the number of times each error has occurred during execution. Whenever the corresponding value of MONTOR is 3 for a given error, and that error occurs, then the appropriate value in KØUNTS is incremented and tested against the one in KNTR. If the maximum number of occurrences for that error has been reached, the job is terminated by INTRAP.

Hence if the default options for INTRAP are used, any error having a corresponding value of 3 or 4 in MONTOR will be fatal.

Appendix E:

Error and Diagnostic Messages from CLUDGE

This appendix gives a list of all the error and diagnostic messages included in the CLUDGE. They are listed alphabetically according to the first invariant information contained in the message. Variables are indicated by XXX in the case of names and numbers (occasionally M and N are used for numbers), and X or Y in the case of characters. We also give the meaning of the message if it is not self-explanatory and the meaning of any variable parts of the message. Further explanations of the table names and code words used in the messages can be found in [5]. The words TYPE OTHER in the messages refer to TYPE INTRVAL.

ARITHMETIC STATEMENT FUNCTION XXX REFERENCED WITH N ARGUMENTS -- M ARGUMENTS DECLARED

ATTEMPT TO POP EMPTY OPERATOR STACK
Attempt to compile a syntactically improper statement

ATTEMPT TO POP EMPTY OPERAND STACK
Attempt to compile a syntactically improper statement

ATTEMPT TO READ PAST END OF FILE
Pre-compiler error

COMPILER ERROR ** ILLEGAL OPERATOR IN STACK
(1) Closing parenthesis ')' in stack
(2) Operator associated with commas is not a function call

COMPILER ERROR ** LOOP IN COMMON BLOCK LINKS
Common block links have been improperly constructed resulting in a circle of entries in some common block

COMPILER ERROR ** $ OR) IN STANDARD CODE LIST
Code list was improperly generated

CONSTRUCTION ERROR -- UNEXPECTED LETTER FOLLOWS CONSTANT
The character following a string interpreted as a constant is not an operator

CONVERSION WITH TYPE LOGICAL UNDEFINED
One side of an arithmetic statement function declaration is type logical and the other has a standard data type (other than logical)

DO LOOP NOT ENDED OR BADLY NESTED

'X' DOES NOT FOLLOW 'Y' IN ALPHABET
A construct of the form (Y-X) was found in an implicit type declaration where 'X' precedes 'Y' alphabetically

END INSERTED
Warning message -- a new program unit was begun, but no end line was present in the preceeding unit and no '*NO ENDS' instruction was present in the instructions to the CLUDGE

ERROR IN CONSTRUCTION
Generalized error message indicating error in the construction of a constant

ERROR IN INSTRUCTIONS TO PRE-COMPILER.
EXECUTION DELETED.
Fatal errors were encountered in the instructions to the CLUDGE

HOLLERITH CONSTANT EXTENDS PAST END OF CARD IMAGE
A symbol interpreted as a Hollerith constant has too great a character count

HOLLERITH CONSTANTS MAY NOT APPEAR IN STATEMENT FUNCTIONS

I THINK YOU FORGOT SOMETHING
No data has been provided for the CLUDGE

ILLEGAL COMBINATION OF OPERATORS
An illegal combination of operators has been found -- see the description of the parsing method and decision table

ILLEGAL COMMON BLOCK NAME
Common block name is not blank, an integer constant, or an identifier

ILLEGAL LIST ITEM
A dimension or type declaration statement is improperly constructed or punctuated

ILLEGAL TYPE NAME
An improper type name has been specified in a *type instruction (note: The test generating this message is not exhaustive)

ILLEGAL VARIABLE
An identifier begins with a character other then a letter

IMPROPER APPEARANCE OF DECIMAL POINT
A decimal point or period was encountered unexpectedly while examining a constant

IMPROPER APPEARANCE OF UNARY OPERATOR
An operator (other than '(') immediately follows a '+' or '-'

IMPROPER FILE ASSIGNMENT
Two files, other than output and print, have been assigned to the same logical unit

INVALID OPERATOR (XX) ENCOUNTERED IN TYPE OTHER CODE
Pre-compiler error -- a parenthesis, comma, logical, or relational operator appears with a type OTHER result indicated. XX is the operator number found

LOGICAL IF STATEMENT FOLLOWS LOGICAL IF STATEMENT

XXX MAY NOT BE DELETED
The intrinsic function named XXX may not be deleted by the user

MORE THAN 19 CONTINUATION CARDS

NEW INTRINSIC INCOMPLETELY SPECIFIED

NON-TERMINATING DIMENSION SIZES
The list of dimension sizes is not followed by a ')'

XXX NOT IN INTRINSIC TABLE
Function XXX specified in a *delete instruction cannot be found in intrinsic table or is a type OTHER intrinsic function (non-fatal)

OVERFLOW OF CODE TABLE -- XXX LINES
Not enough room in the code table. XXX is the value of CODSIZ

OVERFLOW OF CORRESPONDENCE TABLE
Not enough table space to associate a user defined statement label terminating a 'DO' loop with a temporary internal statement number (non-fatal unless caused by a statement with a copy character)

OVERFLOW OF DO STACK WITH N ENTRIES
DO loops are nested too deeply. 'N' is the depth of nesting allowed by the CLUDGE

OVERFLOW IMAGE -- XXX CHARACTERS
Extended area of image exhausted while adding to section 4 of the symbol table. XXX is the value of IMSIZ

OVERFLOW OF OPERATOR-OPERAND STACK
Space for the combined operator-operand stack has been exhausted

OVERFLOW OF STATEMENT FUNCTION CODE TABLE. XXX ENTRIES
Skeleton of expanded arithmetic statement function cannot be stored in skeleton table. XXX is the value of STSIZ

OVERFLOW OF STATEMENT NUMBER TABLE
More than 'RNGSIZ' user defined statement numbers have been declared in the range that CLUDGE must remember (non-fatal, but may be the cause of duplicate statement numbers)

OVERFLOW OF SYMBOL TABLE -- XXX ENTRIES
Not enough symbol table space for a new entry. XXX is the value of TABMAX

PASS 2 *** END-OF-FILE ON SCRATCH 2
Pre-compiler error or program unit with no correct statements

PASS 2 *** EQUIVALENCE TABLE OVERFLOW
Not enough table space to store equivalences between internal and external statement label numbers

PASS 2 *** S-FLAG PRECEEDS D-FLAG ON SCRATCH 1
Pre-compiler error

SYMBOL PREVIOUSLY ASSIGNED TO A COMMON BLOCK
A symbol has been declared to be in COMMON more than once in the same program unit

SYNTAX ERROR IN ARGUMENT LIST
The argument list in the declaration of an arithmetic statement function is not properly punctuated

TOO MANY ARGUMENTS (XXX) FOR TABLE
The argument list in an arithmetic statement function is too long to fit in the table. XXX is the maximum number of arguments that will fit into the table.

TOO MANY CONTINUATION CARDS -- *END POSSIBLY MISSING

TYPE OTHER TEMPORARY STACK EXHAUSTED
More than MAXTEM temporary storage locations are needed for this statement

UNBALANCED PARENTHESES

UNEXPECTED CHARACTER 'X'
The character 'X' was found where a letter was expected

UNRECOGNIZED CONSTRUCT ENCLOSED BY DECIMAL POINTS

UNRECOGNIZED INSTRUCTION
An unrecognized instruction to the CLUDGE has been encountered

UNRECOGNIZED LIST ITEM
Improperly constructed list

UNRECOGNIZED STATEMENT
Non-fatal

UNRECOGNIZED TYPE
An unrecognized type occurs in a *convert instruction

UNRECOGNIZED TYPE BEGINNING WITH XXX
A type name beginning with the string XXX was found in an implicit type declaration statement

WARNING - TYPE OTHER ARGUMENT IN I/O LIST
Non-fatal (See page 15)

APPENDIX 3

Implementation of Best Possible Floating Point Arithmetic for the IBM 360/67

By

Carol A. Clarke

June, 1971

IMPLEMENTATION OF BEST POSSIBLE FLOATING POINT ARITHMETIC FOR THE IBM 360/67

Table of Contents

Introduction 1

Indicators Used 2

A. Input Indicator 2

B. Internal Indicator 2

C. Output Indicator 2

Special Registers Used 4

The Algorithms 5

A. Addition 6

B. Multiplication 8

C. Division 10

D. Bounding and Rounding 12

Introduction

This report is designed to serve as a supplement to:

> J. M. Yohe, "Best Possible Floating Point Arithmetic,"
> The University of Wisconsin, Mathematics Research Center,
> U.S. Army, Technical Summary Report #1054, March, 1970.

A description is given here of the implementation of "Best Possible Floating Point Arithmetic" for the IBM 360/67 at Washington State University, and algorithms are supplied which describe the Assembler routines used in the implementation. A complete listing of these routines is contained in the Interval Arithmetic Package.

Indicators Used

A. Input Indicator:

ØPTIØN is the name of the correction option indicator. Its values and results are as follows:

ØPTIØN		Sign of Result +	Sign of Result -
1	Least Upper Bound	Augment	Truncate
2	Greatest Lower Bound	Truncate	Augment
3	Truncate	Truncate	
5	Augment	Augment	
4	Round	Truncate if 7th hex digit of mantissa of result is < 8. Augment otherwise.	

B. Internal Indicators:

The internal indicators used are essentially the same as described in MRC Tech. Report #1054.

C. Output Indicator:

FAULT is the fault indicator. Its values and their meanings are:

1	EØ	exponent overflow
2	INF	infinity
3	EU	exponent underflow
4	DZ	division by zero

Note: ØPTIØN and FAULT are the first and second words, respectively, of a special control section named BPAIND in the Assembler

routines. Hence they may be accessed from Fortran by using a labeled common area named BPAIND, or from Assembler using a dummy control section. OPTION and FAULT are both treated as integer quantities.

Special Registers Used

A originally contains the first operand in addition, the multiplicand in multiplication, and the dividend in division.

U originally contains the second operand in addition, the multiplier in multiplication, and the divisor in division.

A and U represent single precision floating point numbers as they are represented by the 360, i.e., the high order bit is the sign bit, the next seven bits are the biased exponent, and the last 24 bits are the normalized mantissa.

SIGNA, **SIGNU** — these contain the sign bits of A and U respectively

EXPA, **EXPU** — these contain the exponents of A and U respectively

RESULT contains the resulting single precision floating point number

MANTA, **MANTU** — these contain the expanded mantissas of A and U respectively

V contains the multiplier during multiplication and the quotient during division

MANTA, MANTU, and V are double words. In the algorithms, $MANTA_1$ and $MANTA_2$ refer to the leftmost and rightmost words of MANTA, and $MANTU_1$ and $MANTU_2$ refer to the leftmost and rightmost words of MANTU.

The Algorithms

Below are given the algorithms for routines BPAADD (addition), BPAMUL (multiplication), BPADIV (division), and BPABND (bounding).

In all cases, the arrow "$\leftarrow$" indicates replacement, the double arrow "$\leftrightarrow$" indicates mutual replacement, i.e., a switching of values, and a subscript of 16 indicates a hexidecimal number. Transfer of control is indicated by the next step to be executed, without the words "go to." However, if transfer of control is to a different routine, the words "go to" are used.

A. Addition Algorithm (BPAADD)

1. SC ← 0;

   RI ← 0;

   FAULT ← 0;

2. If |A| > |U|, A ↔ U;

3. If A ≠ 0, Step 5;

4. RESULT ← U;

   Return;

5. If A ≥ 0, Step 7;

6. A ← -A;

   U ← -U;

   SC ← 1;

7. $MANTA_1$ ← 0;

   $MANTA_2$ ← mantissa of A with 00_{16} as rightmost two digits;

   EXPA ← exponent of A;

   SIGNA ← sign of A;

8. $MANTU_1$ ← 0;

   If U ≥ 0, Step 9;

   $MANTU_2$ ← one's complement of mantissa of U with FF_{16} as rightmost two digits;

   Step 10;

9. $MANTU_2$ ← mantissa of U with 00_{16} as rightmost two digits;

10. EXPU ← exponent of U;

    SIGNU ← sign of U;

11. SCOUNT ← (EXPU - EXPA)x4;

12. If SCOUNT ≤ 32, Step 13;

If $MANTA_2 \neq 0$, RI ← 1;

$MANTA_2$ ← 0;

Step 14;

13. Shift $MANTA_2$ right by SCØUNT;

If nonzero digits shifted out, RI ← 1;

14. EXPA ← EXPU;

15. MANTA ← MANTA + MANTU;

16. If SIGNU ≠ 0, Step 20;

17. If $MANTA_1 = 0$, go to BPABND;

18. Shift MANTA right by 4;

If nonzero digits shifted out, RI ← 1;

19. EXPA ← EXPA + 1;

Go to BPABND;

20. If $MANTA_2 \neq 0$, Step 22;

21. RESULT ← 0;

Return;

22. If RI = 1, MANTA ← MANTA + 1;

23. $MANTA_2$ ← one's complement of $MANTA_2$;

SC ← 1 - SC;

24. If first hex digit of $MANTA_2 \neq 0$, go to BPABND;

25. Shift $MANTA_2$ left by 4;

EXPA ← EXPA - 1;

Step 24;

End

B. Multiplication Algorithm (BPAMUL)

1. SC ← 0;

   RI ← 0;

   FAULT ← 0;

2. If A ≠ 0, Step 3;

   RESULT ← 0;

   Return;

3. If A > 0, Step 4;

   A ← -A;

   SC ← 1;

4. $MANTA_2$ ← 0;

   $MANTA_2$ ← mantissa of A with 00_{16} as rightmost two digits;

   EXPA ← exponent of A;

   SIGNA ← sign of A;

5. If U ≠ 0, Step 6;

   RESULT ← 0;

   Return;

6. If U > 0, Step 7;

   U ← -U;

   SC ← 1 - SC;

7. $MANTU_1$ ← 0;

   $MANTU_2$ ← mantissa of U with 00_{16} as rightmost two digits;

   EXPU ← exponent of U;

   SIGNU ← sign of U;

8. V ← MANTA;

   $MANTA_2$ ← 0;

```
9.    MCOUNT ← 6;
10.   Shift V right by 4;
11.   Shift MANTA right by 4;
      If nonzero digits shifted out, RI ← 1;
12.   RCOUNT ← 15th hex digit of V;
13.   If RCOUNT = 0, Step 17;
14.   MANTA ← MANTA + MANTU;
15.   RCOUNT ← RCOUNT - 1;
16.   Step 13;
17.   MCOUNT ← MCOUNT - 1;
      If MCOUNT ≠ 0, Step 10;
18.   If MANTA1 = 0, Step 20;
19.   Shift MANTA right by 4;
      If nonzero digits shifted out, RI ← 1;
      EXPA ← EXPA + 1;
20.   EXPA ← EXPA + EXPU - 41₁₆;
      Go to BPABND;
      End
```

C. Division Algorithm (BPADIV)

1. SC ← 0;
   RI ← 0;
   FAULT ← 0;
   V ← 0;

2. If A ≠ 0, Step 3;
   RESULT ← 0;
   Return;

3. If A > 0, Step 4;
   A ← -A;
   SC ← 1;

4. $MANTA_1$ ← 0;
   $MANTA_2$ ← mantissa of A with 00_{16} as rightmost two digits;
   EXPA ← exponent of A;
   SIGNA ← sign of A;

5. If U ≠ 0, Step 6;
   FAULT ← 4;
   Return;

6. If U > 0, Step 7;
   U ← -U;
   SC ← 1 - SC;

7. $MANTU_1$ ← 0;
   $MANTU_2$ ← mantissa of U with 00_{16} as rightmost two digits;
   EXPU ← exponent of U;
   SIGNU ← sign of U;

8. DCOUNT ← 9;

9. If MANTA < MANTU, Step 11;

10.. MANTA ← MANTA - MANTU;

V ← V + 1;

Step 9;

11. DCØUNT ← DCØUNT - 1;

If DCØUNT = 0, Step 13;

12. Shift MANTA left by 4;

Shift V left by 4;

Step 9;

13. If $MANTA_2 \neq 0$, RI ← 1;

14. MANTA ← V;

15. If $MANTA_1 = 0$, Step 17;

16. Shift MANTA right by 4;

If nonzero digits shifted out, RI ← 1;

Add 1 to EXPA;

17. EXPA ← EXPA -(EXPU - 40_{16});

Go to BPABND;

End

D. Bounding and Rounding Algorithm (BPABND)

1. If EXPA $< 40_{16}$, Step 3;
   If EXPA $\leq 7F_{16}$, Step 6;
2. FAULT $\leftarrow$ 1;
   $MANTA_1 \leftarrow 0$;
   $MANTA_2 \leftarrow FFFFFFF0_{16}$;
   $EXPA \leftarrow 7F_{16}$;
   Step 6;
3. If EXPA $\geq$ 0, Step 6;
   FAULT $\leftarrow$ 3;
   SCOUNT $\leftarrow$ (-EXPA + 5)x4;
   If SCOUNT $\leq$ 32, Step 4;
   If $MANTA_2 \neq 0$, RI $\leftarrow$ 1;
   $MANTA_2 \leftarrow 0$;
   Step 5;
4. Shift MANTA right by SCOUNT;
   If nonzero digits shifted out, RI $\leftarrow$ 1;
5. EXPA $\leftarrow$ 0;
6. If rightmost byte of MANTA = 00_{16} and RI = 0, Step 14;
7. If OPTION = 3, Step 14;
   If OPTION = 5, Step 10;
   If OPTION $\neq$ 4, Step 8;
   If rightmost byte of MANTA $< 80_{16}$, Step 14;
   Step 10;
8. If OPTION = 1, Step 9;
   If SC $\neq$ 0, Step 10;

Step 14;

9. If SC ≠ 0, Step 14;

10. If FAULT ≠ 3, Step 11;

EXPA ← 0

$MANTA_1$ ← 0;

$MANTA_2$ ← 10000000_{16};

Step 14;

11. If FAULT ≠ 1, Step 12;

FAULT ← 2;

Step 14;

12. MANTA ← MANTA + 100_{16};

If $MANTA_1$ = 0, Step 14;

13. Shift MANTA right by 4;

If nonzero digits shifted out, RI ← 1;

EXPA ← EXPA + 1;

If EXPA ≥ 40_{16}, Step 14;

If EXPA > $7F_{16}$, Step 2;

14. If SC = 0, Step 15;

SIGNA ← 1 - SIGNA;

15. RESULT ← floating point number whose sign, exponent, and mantissa are defined by SIGNA, EXPA, $MANTA_2$;

Return;

End

APPENDIX 4

GLOBAL OPTIMIZATION USING INTERVAL ANALYSIS — THE MULTI-DIMENSIONAL CASE

Eldon Hansen*

Abstract. We show how interval analysis can be used to compute the global minimum of a twice continuously differentiable function of n variables over an n-dimensional parallelopiped with sides parallel to the coordinate axes. Our method provides infallible bounds on both the globally minimum value of the function and the point(s) at which the minimum occurs.

Key words: global minimum, interval analysis, minimization, optimization.

Subject classification: 65K05, 90C30

*Department of Pure and Applied Mathematics, Washington State University, Pullman, Washington. This research was supported by the U.S. Air Force Office of Scientific Research under Grant F49620-76-C-0003. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

1. Intoduction.

Consider the function $f(x)$ in C^2 of n variables $x_1, \ldots, x_n$. We shall describe a method for computing the minimum value f^* of $f(x)$ over a box $X^{(0)}$. A box is defined to be a closed rectangular parallelopiped with sides parallel to the coordinate axes. We assume the number of points in $X^{(0)}$ at which $f(x)$ is globally minimum is finite. Our method provides infallible bounds on f^* and on the point(s) x^* for which $f(x^*) = f^*$. That is, our algorithm produces bounds on x^* and f^* which are always correct despite the presence of rounding errors. How sharp these bounds can be depends on the function f and the precision of the computer used.

For a highly oscillatory function f, our algorithm could be prohibitively slow. Presumably this will always be the case for any future global optimization algorithm. However, our algorithm is sufficiently fast for "reasonable" functions.

We assume that interval extensions (see [9]) of f and its derivatives are known. This is the case if every function in terms of which f and its derivatives are defined have known rational approximations with either uniform or rational error bounds for the arguments of interest.

Since the initial box can be chosen as large as we please, our algorithm actually solves the unconstrained minimization problem provided it is known that the solution occurs in some finite region (which we enclose in the initial box).

There is a common misconception among researchers in optimization that it is impossible to obtain infallible bounds on x^* and f^* computationally. The argument is that we can only sample $f(x)$ and a few derivatives of $f(x)$ at a finite number of points. It is possible to interpolate a function having the necessary values and derivatives values at these points and still have its global minimum at any other arbitrary point. The fallacy of this argument is that interval analysis can provide bounds on a function over an entire box; that is over a continuum of points. It is only necessary to make the box sufficiently small in order to make the bounds arbitrarily sharp. This is what our algorithm does. It narrows the region of interest until the bound is as sharp as desired (subject to roundoff restrictions).

In a previous paper [5], we gave a method of this type for the one-dimensional case. The method never failed to converge provided $f'(x)$ and $f''(x)$ had only a finite number of isolated zeros. Our method for the n-dimensional problem appears to always converge also; but we have not yet attempted to prove it. When it does converge, there is never a question that x^* and f^* satisfy the computed bounds.

Recently, R. E. Moore [10] published a method for computing the range of a rational function of n variables over a bounded region. (See also [14].) Although he does not note the fact, his method will serve to bound the global minimum value f^* of a rational function. However, our algorithm is more efficient. Moreover, it is designed to bound x^* as well as f^*.

We suggest the reader read the previous paper [5] before the current one. The one-dimensional case therein serves as an easier introduction. However, the current paper is essentially self contained. It would be better if the reader had some familiarity with the rudiments of interval analysis such as can be found in the first three chapters of [9]. However, we shall review some of its relevant properties.

Our method will find the global minimum (or minima). Because of computer limitations of accuracy, it may also find near-global minima such that rounding errors prevent determination of which is the true minimum. However, if the termination criteria are sufficiently stringent, our algorithm will always eliminate a local minimum whose value is substantially larger than f^*.

Our algorithm is composed of four separate parts. One part uses an interval version of Newton's method to find stationary points. A second part eliminates points of $X^{(0)}$ where f is greater than the smallest currently known value $\overline{f}$.

A third part of our algorithm tests whether f is monotonic in a sub-box X of $X^{(0)}$. If so, we delete part or all of X depending on whether X contains boundary points of $X^{(0)}$.

A fourth part checks for convexity of f in a sub-box X of $X^{(0)}$. If f is not convex anywhere in X, there cannot be a stationary point of f in X.

The first part of the algorithm, if used alone, would find all stationary points in $X^{(0)}$. The second part serves to eliminate stationary points where $f > f^*$. Usually they are eliminated before they are found with any great accuracy. Hence computational effort is not wasted using the first part to accurately find an unwanted stationary point. The second part also serves to eliminate boundary points of $X^{(0)}$ and to find a global minimum if it occurs on the boundary. The second part of the algorithm used alone would find the global minimum (or minima) but its asymptotic convergence is relatively slow compared to that of the Newton method. Hence the latter is used also. The third and fourth parts of the algorithm merely improve convergence.

2. Interval analysis.

The tool which allows us to be certain we have bounded the global minimum is interval analysis. We bound rounding errors by using interval arithmetic. More importantly, however, we use interval analysis to bound the range of a function over a box.

Let $g(x)$ be a rational function of n variables $x_1,\ldots,x_n$. On a computer, we can evaluate $g(x)$ for a given x by performing a sequence of arithmetic operations involving only addition, subtraction, multiplication, and division.

Let X_i $(i = 1,\ldots,n)$ be closed intervals. If we use X_i in place of x_i and perform the same sequence of operations using interval arithmetic (see[9]) rather than ordinary real arithmetic, we obtain a closed interval $g(X)$ containing the range

$$\{g(x) : x_i \in X_i \quad (i = 1,\ldots,n)\}$$

of $g(x)$ over the box X. This result will not be sharp, in general, but if outward rounding (see[9]) is used, then $g(X)$ will always contain the range. The lack of sharpness results from other causes besides roundoff. With exact interval arithmetic, the lack of sharpness disappears as the widths of the intervals decrease to zero.

If $g(x)$ is not rational, we assume an algorithm is known for computing an interval $g(X)$ containing the range of $g(x)$ for $x \in X$. Methods for deriving such algorithms are discussed in [9]).

3. Taylor's Theorem.

We shall use interval analysis in conjunction with Taylor's theorem in two ways. First, we expand f as

$$f(y) = f(x) + (y-x)^T g(x) + \tfrac{1}{2}(y-x)^T H(x,y,\xi)\,(y-x) \tag{3.1}$$

where $g(x)$ is the gradient of $f(x)$ and has components $g_i(x) = \partial f(x)/\partial x_i$. The quantity $H(x,y,\xi)$ is the Hessian matrix to be defined presently. For reasons related to the use of interval analysis, we shall express it as a lower triangular matrix instead of a symmetric matrix so that there are fewer terms in the quadratic form involving $H(x,y,\xi)$.

We define the element in position (i,j) of $H(x,y,\xi)$ as

$$h_{ij} = \begin{cases} \partial^2 f/\partial x_i^2 & \text{for } j = i \quad (i = 1, \cdots, n), \\ 2\partial^2 f/\partial x_i \partial x_j & \text{for } j < i \quad (i = 1, \cdots, n;\ j = 1, \cdots, i-1), \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

The arguments of h_{ij} depend on i and j. If we expand f sequentially in one of its variables at a time, we can obtain the

following results illustrating the case $n = 3$

$$H(x,y,\xi) = \begin{bmatrix} h_{11}(\xi_{11},x_2,x_3) & 0 & 0 \\ h_{21}(\xi_{21},x_2,x_3) & h_{22}(y_1,\xi_{22},x_3) & 0 \\ h_{31}(\xi_{31},x_2,x_3) & h_{32}(y_1,\xi_{32},x_3) & h_{33}(y_1,y_2,\xi_{33}) \end{bmatrix}.$$

Assume $x_i \in X_i$ and $y_i \in X_i$ for $i = 1,\ldots,n$. Then $\xi_{ij} \in X_j$ for each $j = 1, \ldots,i$. For general n, the arguments of H_{ij} are $(y_1,\ldots,y_{j-1},\xi_{ij}, x_{j+1},\ldots,x_n)$. Other arrangements of arguments could be obtained by reordering the indices.

Let x be a fixed point in X. Then for any point $y \in X$,

$$H(x,y,\xi) \in H(x,X,X) ;$$

that is, for $i \geq j$,

$$h_{ij}(y_1,\ldots,y_{j-1}, \xi_{ij}, x_{j+1},\ldots,x_n) \in h_{ij}(X_1,\ldots,X_j, x_{j+1},\ldots,x_n).$$

In the sequel, we shall shorten notation and use $H(\xi)$ to denote $H(x,y,\xi)$ and $H(X)$ to denote $H(x,X,X)$.

The purpose of this particular Taylor expansion is to obtain real (non-interval) quantities for as many arguments of the elements of $H(X)$ as possible. The standard Taylor expansion would have intervals for all arguments of all elements of $H(X)$. This type of expansion was introduced in [3]. A more general approach of this kind is discussed in [4].

The other Taylor expansion we shall want is of the gradient g. Each element g_i $(i = 1,\ldots,n)$ of g can be expanded as

$$\begin{aligned} g_i(y) = g_i(x) &+ (y_1-x_1)J_{i1}(\eta_1,x_2,\cdots,x_n) + (y_2-x_2)J_{i2}(y_1,\eta_2,x_3,\cdots,x_n) \\ &+ (y_3-x_3)J_{i3}(y_1,y_2,\eta_3,x_4,\cdots,x_n) + \cdots + (y_n-x_n)J_{in}(y_1,\cdots,y_{n-1},\eta_n) , \end{aligned} \tag{3.3}$$

where

$$J_{ij} = \partial^2 f/\partial x_i \partial x_j \qquad (i,j = 1,\cdots,n)$$

This Jacobian matrix J and the Hessian H introduced above are, of course, essentially the same. However, they will be evaluated with different arguments depending on whether we are expanding f or g. Also, H is lower triangular while J is a full matrix.

Let $J(x,y,\eta)$ denote the Jacobian matrix with elements $J_{ij}(y_1, \cdots, y_{j-1}, \eta_j, x_{j+1}, \cdots, x_n)$. Then

$$g(y) = g(x) + J(x,y,\eta)(y-x) . \tag{3.4}$$

If $x \in X$ and $y \in X$, then $\eta_i \in X_i$ for all $i = 1, \cdots, n$. Hence

$$g(y) \in g(x) + J(x,X,X)(y-x) . \tag{3.5}$$

We shall again shorten notation and denote $J(x,y,\eta)$ by $J(\eta)$ and $J(x,X,X)$ by $J(X)$.

Note that the elements of $H(X)$ on and below the diagonal have the same arguments as the corresponding elements of $J(X)$. Thus we need only calculate $J(X)$; then $H(X)$ follows easily.

4. The approximate value of the global minimum.

As we proceed with our algorithm, we shall evaluate $f(x)$ at various points in $X^{(0)}$. Let $\bar{f}$ denote the currently smallest value of f found so far. The very first step is to evaluate f at the center of $X^{(0)}$. This value serves as the first one for $\bar{f}$.

One part of our algorithm deletes sub-boxes of $X^{(0)}$ wherein $f > \bar{f}$ since this implies $\inf f > f^*$. (See section ~.)

In practice we cannot generally evaluate $f(x)$ exactly because of rounding errors. Hence we do the evaluation using interval arithmetic. Suppose we obtain the interval $[f^L, f^R]$. Then we know that $f(x) \leq f^R$ and hence that $\bar{f} \leq f^R$. Hence when we evaluate $f(x)$, we update $\bar{f}$ by replacing it by f^R only if f^R is less than the previous value of $\bar{f}$. In this way, we assure that $\bar{f}$ is always an upper bound for f^*.

5. A test for convexity.

As our algorithm proceeds, we dynamically subdivide $X^{(0)}$ into sub-boxes. Let X denote such a sub-box. We evaluate $h_{ii}(X_1, \cdots, X_n)$ for $i = 1, \cdots, n$, where h_{ii} is the diagonal element of the Hessian. Note that every argument of h_{ii} is an interval and hence the resulting interval contains the value of $h_{ii}(x)$ for every $x \in X$. That is, if $[u_i, v_i]$ denotes the computed interval $h_{ii}(X_1, \cdots, X_n)$, then

$$h_{ii}(x) \in [u_i, v_i]$$

for all $x \in X$.

Suppose we find $v_i < 0$ for some value of i. Then $h_{ii}(x) < 0$ for every $x \in X$. Hence there is no point in X at which the real (non-interval) Hessian is positive definite. Hence f is not convex and cannot have a minimum which is a stationary point in X. Hence we can delete all of X except for any boundary points of $X^{(0)}$ which might lie in X.

When we evaluate $h_{ii}(X_1, \cdots, X_n)$, we may find that the left endpoint $u_i \geq 0$ for all $i = 1, \cdots, n$. When this occurs, we know from inclusion monotonicity (see [9]) that we will find each $u_i \geq 0$ for any sub-box of X. Hence we could save some computational effort by noting when a box is a sub-box of one for which $u_i \geq 0$ for all $i = 1, \cdots, n$. We would skip this test for such a box.

Note that an element h_{ii} with arguments $(X_1, \cdots, X_n)$ is not obtained when we compute $H(X)$ since the diagonal elements of $H(X)$ have arguments different from $(X_1, \cdots, X_n)$ except for the

element in position (n,n). Hence our test for convexity requires recalculation of the diagonal of the Hessian.

6. The interval Newton method.

For each sub-box X of $X^{(0)}$ that our algorithm generates, we can apply an interval Newton method to the gradient g. Such methods seek the zeros of g and hence the stationary points of f. Such a method produces from X a new box or boxes $N(X)$. Any points in X not in $N(X)$ cannot contain a zero of g and can be discarded unless they are boundary points of $X^{(0)}$.

These methods, in effect, solve (3.5) for points y where $g(y) = 0$. The first such method was derived by Moore [9]. Variants of Moore's method can be found in [5,6,8,13]. The most efficient variant can be found in [6]. Krawczyk's method [8] is a suitable alternative to the method in [6]. Discussions of Krawczyk's method can be found in [11] and [12].

We now give a brief synopsis of the method in [6]. We wish to solve the set of equations

$$g(x) + J(\xi)(y-x) = 0 \tag{6.1}$$

for the set of points y obtained by letting ξ range over X. We shall find a subset Y of X containing this set.

Let J_c be the matrix whose element in position (i,j) is the midpoint of the corresponding interval element $J_{ij}(X)$ of the Jacobian $J(X)$. Let B be an approximate inverse of J_c. As pointed out in [5], a useful first step in solving for Y is to multiply (6.1) by B giving

(6.2) $$Bg(x) + BJ(\xi)(y-x) = 0 .$$

Note that the product $BJ(\xi)$ approximates the identity matrix. However it may be a very poor approximation when X is a large box.

We "solve" (6.2) by a process similar to a single sweep of the Gauss-Seidel method. Write

$$BJ(X) = L + D + U$$

where L, D, and U are the lower triangular, diagonal, and upper triangular part of $BJ(X)$, respectively. The interval matrix

(6.3) $$D^{-1} = \mathrm{diag}[1/D_{11}, 1/D_{22}, \cdots, 1/D_{nn}]$$

contains the inverse of every matrix in D. The box Y "solving" (6.2) is obtained as

(6.4) $$Y = x - D^{-1}[Bg(x) + L(Y-x) + U(X-x)] .$$

When obtaining the component Y_i of Y, the components $Y_1, \cdots, Y_{i-1}$ appearing in the right member of this equation have already been obtained.

This formulation presupposes that the intervals D_{ii} $(i = 1, \cdots, n)$ do not contain zero. When X is a small box, $BJ(X)$ is closely approximated by the identity matrix and hence D is also. However, for X large, it is possible to have $0 \in D_{ii}$ for one or more values of i. This case is easily handled. We simply use an extended interval arithmetic which allows division by an interval containing zero. (See [6] for details.)

Note that we cannot allow the Newton procedure to delete boundary points of $X^{(0)}$ since the global minimum need not be a stationary point if it occurs on the boundary. We discuss this point further in Section 10.

If we were to use this Newton method only, we would in general find stationary points of f which were not minima. Moreover, we would find local minima which were not global minima. To avoid this, we use an additional procedure to delete points where f exceeds the smallest known value $\bar{f}$. This procedure is described in the next section.

In some applications, it may be desirable to find all the stationary points of f in a given box. This can be done using the Newton method alone or in conjunction with the monotonicity check of Section 9. If, in addition, the convexity check of Section 5 were used, all stationary points except maximum would be found.

7. Bounding f.

We now consider how to delete points $y \in X$ where we know $f(y) > \bar{f}$ and hence where $f(y)$ is not a global minimum. We retain the complementary set which is a sub-box (or sub-boxes) $Y \subset X$ wherein $f(y)$ may be $\leq \bar{f}$.

As pointed out in [5], if we only wish to bound f^* and not x^*, we can delete points where

$$f(y) > \bar{f} - \varepsilon_1 \tag{7.1}$$

for some $\varepsilon_1 > 0$. We can allow ε_1 to be nonzero only if we do not need to know the point(s) x^* at which f is globally minimum.

We want to retain points where (7.1) is not satisfied. From (3.1), this is the case for points y if

$$f(x) + (y-x)^T g(x) + \tfrac{1}{2}(y-x)^T H(\xi)(y-x) \leq \bar{f} - \varepsilon_1$$

because the left member equals $f(y)$. Denote

$$E = \bar{f} - f(x) - \varepsilon_1 .$$

Then

$$\tilde{y}^T g(x) + \tfrac{1}{2}\tilde{y}^T H(\xi)\tilde{y} \leq E \tag{7.2}$$

where $\tilde{y} = y - x$. We shall use this relation to reduce X in one dimension at a time to yield the sub-box(es) Y resulting from deleting points where $f(y) > \bar{f} - \varepsilon_1$.

We shall illustrate the process for the case $n = 2$. The higher dimensional case follows in the same way. For $n = 2$, (7.2) becomes

$$\tilde{y}_1 g_1(x) + \tilde{y}_2 g_2(x) + \tfrac{1}{2}\left|\tilde{y}_1^2 h_{11}(\xi) + \tilde{y}_1\tilde{y}_2 h_{21}(\xi) + \tilde{y}_2^2 h_{22}(\xi)\right| \le E . \tag{7.3}$$

We first wish to reduce X in the x_1-direction. Thus we solve this relation for acceptable values of y_1. After collecting terms in y_1, we replace y_2 by X_2. In the higher dimensional case we would also replace y_i by X_i for all $i = 3, \cdots, n$. We also replace ξ by X (since $\xi \in X$). We obtain

$$\tilde{y}_1[g_1(x) + \tfrac{1}{2}\tilde{X}_2 h_{21}(X)] + \tfrac{1}{2}\tilde{y}_1^2 h_{11}(X) + \tilde{X}_2 g_2(x) + \tfrac{1}{2}\tilde{X}_2^2 h_{22}(X) - E \le 0 \tag{7.4}$$

where $\tilde{X}_2 = X_2 - x_2$.

We solve this quadratic for the interval or intervals of points y_1 as described below. Call the resulting set Z_1. Since we are only interested in points with $y_1 \in X_1$, we compute the desired set Y_1 as $Y_1 = X_1 \cap Z_1$.

For the sake of argument, suppose Y_1 is a single interval. We can then try to reduce X_2 the same way we (hopefully) reduced X_1 to get Y_1. We again rewrite (7.3). This time we replace y_1 by Y_1 and (as before) ξ by X. We could obtain better results by replacing ξ_1 by Y_1 rather than X_1 but this would require re-evaluation of the elements of H. We obtain

$$\tilde{y}_2[g_2(x) + \tfrac{1}{2}\tilde{Y}_1 h_{21}(X)] + \tfrac{1}{2}\tilde{y}_2^2 h_{22}(X) + \tilde{Y}_1 g_1(x) + \tfrac{1}{2}\tilde{Y}_1^2 h_{11}(X) - E \le 0 \tag{7.5}$$

where $\tilde{Y}_1 = Y_1 - x_1$.

If the solution set Y_2 is strictly contained in X_2, we could replace X_2 by Y_2 in (7.4) and solve for a new Y_1. We have not tried to do this in practice. Instead, we start over with the box Y in place of X as soon as we have tried to reduce each X_i to Y_i $(i = 1, \cdots, n)$. Note this means we re-evaluate $H(X)$.

We now consider how to solve the quadratic equation (7.4) or (7.5). These have the general form

$$A + Bt + Ct^2 \leq 0 \tag{7.6}$$

where A, B, and C are intervals and we seek values of t satisfying this inequality.

Denote $C = [c_1, c_2]$ and let c be an arbitrary point in C. Similarly, let $a \in A$ and $b \in B$ be arbitrary. Suppose t is such that (7.6) is violated; that is $Q(t) > 0$, where

$$Q(t) = a + bt + ct^2 .$$

If this is true for $c = c_1$, then it is true for all $c \in C$. Hence if we wish to find the complementary values of t where (7.6) might hold we need only consider

$$A + Bt + c_1 t^2 \leq 0 . \tag{7.7}$$

If $c_1 = 0$, this relation is linear and the solution set T is as follows: Denote $A = [a_1, a_2]$ and $B = [b_1, b_2]$. Then the set of solution points t is

$$T = \begin{cases} [-a_1/b_2, \infty] & \text{if } a_1 \le 0 , \ b_2 < 0 , \\ [-a_1/b_1, \infty] & \text{if } a_1 > 0 , \ b_1 < 0 , \ b_2 \le 0 , \\ [-\infty, \infty] & \text{if } a_1 \le 0 , \ b_1 \le 0 \le b_2 , \\ [-\infty, -a_1/b_2] \cup [-a_1/b_1, \infty] & \text{if } a_1 > 0 , \ b_1 < 0 < b_2 , \\ [-\infty, -a_1/b_1] & \text{if } a_1 \le 0 , \ b_1 > 0 , \\ [-\infty, -a_1/b_2] & \text{if } a_1 > 0 , \ b_1 \ge 0 , \ b_2 > 0 , \\ \text{empty set} & \text{if } a_1 > 0 , \ b_1 = b_2 = 0 . \end{cases}$$

Recall that we will intersect T with X_i for some value of i. Thus although T may be unbounded, the intersection is bounded.

If $c_1 \neq 0$, the quadratic (7.6) may have no solution or it may have a solution set T composed of either one or two intervals. In the latter case, the intervals may be semi-infinite. However, after intersecting T with X_i, the result is finite.

Denote

$$Q_1(t) = a + bt + c_1 t^2$$

where $a \in A$, $b \in B$, and c_1 is the left endpoint of C. We shall delete points t where $Q_1(t) > 0$ for all $a \in A$ and $b \in B$. Thus we retain a set T of points where $Q_1(t) \le 0$, as desired. But we also retain (in T) points where, for fixed t, $Q_1(t) > 0$ for some $a \in A$ and $b \in B$ and $Q_1(t) \le 0$ for other $a \in A$ and $b \in B$. This same criterion was used to obtain T when $c_1 = 0$. This assures that we shall always retain points in X_i where $f(x)$ is a minimum.

Denote

$$q_1(t) = \begin{cases} a_1 + b_2 t + c_1 t^2 & \text{if } t \le 0, \\ a_1 + b_1 t + c_1 t^2 & \text{if } t \ge 0 \end{cases}$$

and

$$q_2(t) = \begin{cases} a_2 + b_1 t + c_1 t^2 & \text{if } t \le 0, \\ a_2 + b_2 t + c_1 t^2 & \text{if } t \ge 0. \end{cases}$$

Then we can write the interval quadratic as

$$\begin{aligned} Q_1(t) &= [a_1, a_2] + [b_1, b_2]t + c_1 t^2 \\ &= [q_1(t), q_2(t)] . \end{aligned}$$

Thus for any finite t, $q_1(t)$ is a lower bound for $Q_1(t)$ and $q_2(t)$ is an upper bound for $Q_1(t)$ for any $a \in A$ and any $b \in B$.

For a given value of t, if $q_1(t) > 0$, then $Q_1(t) > 0$ for all $a \in A$ and $b \in B$. Hence we need only to solve the real quadratic equation $q_1(t) = 0$ in order to determine intervals wherein, without question, $Q_1(t) > 0$. This is a straightforward problem.

The function $q_1(t)$ is continuous but its derivative is discontinuous at the origin when $b_1 \neq b_2$ which will generally be the case in practice. Hence we must consider the cases $t \le 0$ and $t \ge 0$ separately.

If $c_1 > 0$, the curve $q_1(t)$ is convex for $t \le 0$ and convex for $t \ge 0$. Consider the case $t \le 0$. If $q_1(t)$ has real roots, then $Q_1(t) > 0$ outside these roots, provided $t \le 0$. Hence, we retain the interval between these roots. We need only examine the discriminant of $q_1(t)$ to determine whether the roots are real or not. Hence it is a simple procedure to determine which part (if any) of the half line $t \le 0$ can be deleted. The same procedure can be used for $t \ge 0$.

For $c_1 < 0$, $q_1(t)$ is concave for $t \le 0$ and for $t \ge 0$. In this case we can delete the interval (if any) between the roots of $q_1(t)$ in each half line. The set T is the complement of this interval. It is composed of two semi-infinite intervals.

In determining T for either the case $c_1 < 0$ or in the case $c_1 > 0$, it is necessary to know whether the discriminant of $q_1(t)$ is non-negative or not. Denote

$$\Delta_1 = b_1^2 - 4a_1c_1, \qquad \Delta_2 = b_2^2 - 4a_1c_1 .$$

These are the discriminants of $q_1(t)$ when $t \ge 0$ and $t \le 0$, respectively.

When we compute Δ_1 or Δ_2, we shall make rounding errors. Thus we should compute them using interval arithmetic to bound these errors. When computing Δ_i $(i = 1,2)$, suppose we obtain the interval

$$\Delta_i^I = [\Delta_i^L, \Delta_i^R] \qquad (i = 1,2).$$

We use the appropriate endpoint of Δ_1^I or Δ_2^I to determine T which assures that we never delete a point t where $Q_1(t)$ could be non-positive. Thus we use the endpoint of Δ_1^I or Δ_2^I which yields the larger set T.

When we compute the roots of $q_1(t)$, we shall make rounding errors. Hence we compute them using interval arithmetic and again use the endpoints which yield the larger set T to assure we do not delete a point in X_i where f is a minimum.

For $i = 1$ and 2, denote

$$R_i^{\pm} = (-b_i \pm \Delta_i^{1/2})/(2c_1)$$

and

$$S_i^{\pm} = 2a/(-b_i \pm \Delta_i^{1/2}).$$

Note that $R_i^{+} = S_i^{-}$ and $R_i^{-} = S_i^{+}$. As is well known, the rounding error is less if we compute a root in the form R_i^{+} rather than in

the form S_i^- when $b_i < 0$. The converse is true when $b_i > 0$. Similarly, the rounding error is less when using R_i^- rather than S_i^+ when $b_i > 0$. Hence we compute the roots of $q_1(t)$ as R_i^+ and S_i^+ when $b_i < 0$ and as R_i^- and S_i^- when $b_i > 0$.

Note that computing R_i^- or S_i^- involves taking the square root of the interval Δ_i^I. In exact arithmetic this would be the real quantity Δ_i. We would never be computing roots of $q_1(t)$ when Δ_i was negative. Hence if we find that the computed result Δ_i^I contains zero, we can replace it by its non-negative part. Thus we will never try to take the square root of an interval containing negative numbers.

Given any interval I, let I^L and I^R denote its left and right endpoint, respectively. We use this notation below. Using the above prescriptions on how to compute the set T, we obtain the following results:

For $b_1 \geq 0$ and $c_1 > 0$,

$$(7.8)\quad T = \begin{cases} \phi \text{ (the empty set)} & \text{if } \Delta_2^R < 0, \\ [(R_2^-)^L,\ (S_2^-)^R] & \text{if } a_1 > 0 \text{ and } \Delta_2^R \geq 0, \\ [(R_2^-)^L,\ (S_1^-)^R] & \text{if } a_1 \leq 0. \end{cases}$$

For $b_2 \leq 0$ and $c_1 > 0$,

$$(7.9)\quad T = \begin{cases} \phi & \text{if } \Delta_1^R < 0, \\ [(S_1^+)^L,\ (R_1^+)^R] & \text{if } a_1 > 0 \text{ and } \Delta_1^R \geq 0, \\ [(S_2^+)^L,\ (R_1^+)^R] & \text{if } a_1 \leq 0. \end{cases}$$

For $b_1 \le 0 \le b_2$ and $c_1 > 0$,

$$(7.10)\quad T = \begin{cases} \phi & \text{if } \max(\Delta_1^R, \Delta_2^R) < 0, \\ [(R_2^-)^L, (S_2^-)^R] & \text{if } |b_1| < b_2 \text{ and } \min(\Delta_1^R, \Delta_2^R) \le 0 \le \max(\Delta_1^R, \Delta_2^R), \\ [(S_1^+)^L, (R_1^+)^R] & \text{if } |b_1| > b_2 \text{ and } \min(\Delta_1^R, \Delta_2^R) \le 0 \le \max(\Delta_1^R, \Delta_2^R), \\ [(R_2^-)^L, (S_2^-)^R] \cup [(S_1^+)^L, (R_1^+)^R] & \text{if } a_1 > 0 \text{ and } \min(\Delta_1^R, \Delta_2^R) > 0 \\ [(R_2^-)^L, (R_1^+)^R] & \text{if } a_1 \le 0. \end{cases}$$

For $b_1 \ge 0$ and $c_1 < 0$,

$$(7.11)\quad T = \begin{cases} [-\infty, (S_2^-)^R] \cup [(R_1^-)^L, \infty] & \text{if } a_1 > 0, \\ [-\infty, (S_1^-)^R] \cup [R_1^-)^L, \infty] & \text{if } a_1 \le 0 \le \Delta_1^L, \\ [-\infty, \infty] & \text{if } \Delta_1^L < 0. \end{cases}$$

For $b_2 \le 0$ and $c_1 < 0$,

$$(7.12)\quad T = \begin{cases} [-\infty, (R_2^+)^R] \cup [(S_1^+)^L, \infty] & \text{if } a_1 > 0, \\ [-\infty, (R_2^+)^R] \cup [(S_2^+)^L, \infty] & \text{if } a_1 \le 0 \le \Delta_2^L, \\ [-\infty, \infty] & \text{if } \Delta_2^L < 0. \end{cases}$$

For $b_1 \le 0 \le b_2$ and $c_1 < 0$,

$$(7.13)\quad T = \begin{cases} [-\infty, (S_2^-)^R] \cup [(S_1^+)^L, \infty] & \text{if } a_1 \ge 0, \\ [-\infty, \infty] & \text{if } a_1 < 0, \end{cases}$$

Note that if $c_1 > 0$, then Δ_2 can be negative only if $a_1 > 0$. Hence the condition $\Delta_2 < 0$ implies $a_1 > 0$. This, and similar cases, has been used to shorten the conditional statements in the above expressions for T.

We have seen that the solution of the quadratic inequalities such as (7.4) or (7.5) can be an interval Z_i or two semi-infinite intervals, say $Z_i^{(1)}$ and $Z_i^{(2)}$. The desired solution set Y_i is obtained by intersection with X_i. In the former case, $Y_i = X_i \cap Z_i$ can be empty or a single interval. In the latter case,

$$Y_i = X_i \cap \left(Z_i^{(1)} \cup Z_i^{(2)}\right)$$

can be empty, a single interval, or two intervals. We now consider the logistics of handling these cases.

The quadratic inequality to be solved for Z_i will have quadratic term $\frac{1}{2}\tilde{y}_i^2 h_{ii}(X)$ so the interval C in (7.6) is $\frac{1}{2} h_{ii}(X)$ and the left endpoint is $c_1^{(i)} = [\frac{1}{2} h_{ii}(X)]^L$. If $c_1^{(i)} \geq 0$ the solution set is a single interval. But if $c_1^{(i)} < 0$, it is two semi-infinite intervals and it may be that Y_i will be two intervals. This would complicate the process of finding $Y_{i+1}, \cdots, Y_n$. Thus we proceed as follows.

Let I_1 denote the set of indices i for which $c_1^{(i)} \geq 0$ and I_2 denote the set of indices i for which $c_1^{(i)} < 0$. We first find Y_i for each $i \in I_1$. We then begin to find Y_i for $i \in I_2$. Let $j \in I_2$ be such that Y_j is composed of two intervals, say $Y_j^{(1)}$ and $Y_j^{(2)}$. Then X_j is the smallest interval containing both $Y_j^{(1)}$ and $Y_j^{(2)}$. When finding Y_i for the remaining values of i, we use X_j in place of Y_j.

After finding all Y_i for $i = 1, \cdots, n$, we wish to use the fact that we can delete the interval, say Y_j^c, between $Y_j^{(1)}$ and $Y_j^{(2)}$. We would like to do this for all the values of j for which Y_j was two intervals. However, it could be that this occurred for all the indices $j = 1, \cdots, n$. After deleting the interior interval Y_j^c in each dimension, the resulting set would be composed of 2^n boxes. For large n, this is too many boxes to handle separately. Hence we delete only a few (one, two, or

AD-A078 089 LOCKHEED MISSILES AND SPACE CO INC PALO ALTO CA PALO --ETC F/G 12/1
GLOBALMIN - A COMPUTER PROGRAM FOR GLOBAL OPTIMIZATION.(U)
NOV 79 E R HANSEN F49620-76-C-0003
UNCLASSIFIED LMSC/D683307 AFOSR-TR-79-1227 NL

2 OF 2
AD-
A078089

END
DATE
FILMED
1-80
DDC

three) of the largest of the intervals Y_j^c . We then process each of the new boxes separately.

We would like to prevent the generation of long, narrow boxes. Thus a good choice of which Y_j^c to delete is the one(s) corresponding to the component for which the smallest interval containing both $Y_j^{(1)}$ and $Y_j^{(2)}$ is largest. However, we have chosen to delete the largest interval Y_j^c .

Let us call the process we have described in this section the *quadratic method*. We can combine the quadratic method with the Newton method. It is desirable to do this as we now explain.

If the left endpoint of $H_{ii}(X)$ is negative, then the quadratic method can give rise to two new intervals $Y_i^{(1)}$ and $Y_i^{(2)}$ in place of X_i . When trying to improve X_{i+1} (say), it is impractical to use $Y_i^{(1)}$ and $Y_i^{(2)}$ separately and we use X_i , instead. Thus the improvement of X_i is of no help when trying to improve X_{i+1} , etc. Similarly, when applying the Newton step, if $J_{ii}(X)$ contains zero as an interior point, we can obtain two subintervals in place of X_i . Again, we cannot conveniently use this fact in the remaining part of the Newton step.

We would like to do those steps first which are of help in subsequent steps. Hence the following sequence is suggested. First try to improve X_i by the quadratic method for each value of $i = 1, \cdots, n$ for which the left endpoint of $H_{ii}(X)$ is positive. Then apply the interval Newton method to the (old or

new) components for which $0 \notin B\ J_{ii}(X)(i = 1, \cdots, n)$. Next use the quadratic method for those components for which the left end-point of $H_{ii}(X)$ is non-positive. Finally, complete the Newton step for those components with $0 \notin B\ J_{ii}(X)$.

At each stage of either method, when trying to improve the i-th component of the box, we use the currently best interval for the other components. This may be the smallest interval containing two disjoint intervals in some cases. In fact it would be possible for the quadratic method and the Newton method to each delete disjoint sub-intervals for a given component. This would give rise to three sub-intervals to be retained. However, it seems better to simplify this case and only delete the larger of the two sub-intervals.

When both methods are completed, we may have several components divided into two sub-intervals. If so, we find the one for which the largest interior sub-interval has been deleted. We replace all the others by the smallest sub-interval containing the two disjoint parts. We then divide the remaining part of the current box into two sub-boxes by deleting the sub-interval for the component in question. We could do this for more than one component, but each deletion would double the number of boxes. It seems better to keep the number of boxes small.

8. *Choice of ε_1.*

Suppose we want to bound the value f^* of the global minimum to within a tolerance ε_1 but we do not care where f takes on this value. Then, as pointed out in Section 7, we can delete points y where

$$f(y) > \bar{f} - \varepsilon_1 . \tag{8.1}$$

Once we have found a point $\bar{x}$ where $f(\bar{x}) = \bar{f}$ is such that $\bar{f} - f^* \leq \varepsilon_1$, our algorithm will eventually delete all of $X^{(0)}$ if we use (8.1). However, $\bar{x}$ may be far from the point x^* where f is globally minimal. When all of $X^{(0)}$ is deleted, we will know that

$$\bar{f} - \varepsilon_1 \leq f^* \leq \bar{f} .$$

Choosing $\varepsilon_1 > 0$ will speed up our algorithm. However, if we wish to obtain good bounds on x^*, we must choose $\varepsilon_1 = 0$. We then terminate our algorithm when the remaining set of points is sufficiently small. See Section 13 for a termination procedure.

9. Monotonicity.

Another step in our algorithm makes use of the monotonicity of f. Suppose, for example, the i-th component $g_i(x)$ of the gradient is non-negative for all $x \in X$. Then the smallest value of $f(x)$ for $x \in X$ must occur for x_i equal to the left end-point of X_i.

To make use of a fact such as this, we evaluate $g_i(X_1, \cdots, X_n)$. The resulting interval, which we denote by $[\sigma_i, \omega_i]$, contains $g_i(x)$ for all $x \in X$. Denote $X_i = [x_i^L, x_i^R]$. If $\sigma_i \geq 0$, $f(x)$ is smallest (in X) for $x_i = x_i^L$. Hence we can delete all of X except the points with $x_i = x_i^L$. If $\sigma_i > 0$, $f(x)$ cannot have a stationary point in X. Hence we can delete all of X unless the boundary at $x_i = x_i^L$ contains boundary points of the initial box $X^{(0)}$. Similar results occur if $\omega_i \leq 0$ or if $\omega_i < 0$.

We evaluate $g_i(X_1, \cdots, X_n)$ for $i = 1, \cdots, n$ and reduce the dimensionality of X for any value of i for which $\sigma_i \geq 0$ or $\omega_i \leq 0$. Of course, we delete all of X, if possible.

It is possible that we can reduce X in every dimension in this way. If so, only a single point, say $\tilde{x}$, remains. In this case, we evaluate $f(\tilde{x})$. If $f(\tilde{x}) > \overline{f}$, we can eliminate $\tilde{x}$ and hence all of X is deleted by the process. If $f(\tilde{x}) \leq \overline{f}$, we reset $\overline{f}$ equal to $f(\tilde{x})$. In this latter case, X is again deleted; but we store $\tilde{x}$ for future reference.

10. Boundary points.

The process just described in Section 9 can sometimes eliminate points of the boundary of $X^{(0)}$ which lie in X. Suppose that for some X, we find $g_j(X_1, \cdots, X_n) > 0$ for some $j = 1, \cdots, n$. Then we can delete all of X except for any boundary points of $X^{(0)}$ occurring at $x_j = x_j^L$. Any other boundary points of $X^{(0)}$ which are in X are thus deleted.

The quadratic method of Section 7 deletes any point x where $f(x) > \bar{f}$ whether x lies on the boundary of $X^{(0)}$ or not. However, the Newton method of Section 6 and the procedure in Section 5 (which considers convexity) cannot delete any boundary points of $X^{(0)}$.

Suppose we apply a step of the Newton method to a box X and obtain a new box X' contained in X. A simple way to proceed is to retain the smallest box containing both X' and all boundary points of $X^{(0)}$ which are in X. This will generally save points of X outside X' thus reducing the efficiency of the procedure. In fact, it may be that the smallest box containing the boundary points of $X^{(0)}$ which are in X is X itself. If this were the case, we would bypass the Newton step for the box X. This approach would rely upon the methods of Sections 7 and 9 to delete boundary points of $X^{(0)}$.

This same idea can be used for the method of Section 5. If f is not convex in X, we can simply replace X by the smallest (perhaps degenerate) box, say $\bar{X}$, containing the boundary

points of $X^{(0)}$ which lie in X. In this case, either $\overline{X} = X$ or else $\overline{X}$ is a degenerate box of dimension less than that of X.

Suppose we are given a box X. For this approach, if $\overline{X} = X$, we do not apply either the Newton method or the convexity test. We could use the Newton method in this case also or we might bypass its use whenever X contains boundary points of $X^{(0)}$.

A more straightforward procedure is to simply express the boundary of $X^{(0)}$ as $2n$ separate (degenerate) boxes of dimension $n-1$. The interior of $X^{(0)}$ can then be treated as a box wherein the global minimum must be a stationary point. However, the $(n-1)$-dimensional faces of $X^{(0)}$ have $(n-2)$-dimensional boundaries which must, in turn, be separated from the interiors, and so on. Finally the vertices of $X^{(0)}$ would have to be separated. These vertices alone are 2^n points. Even for moderate values of n, this separation process produces too many (degenerate) boxes. Thus it is better, in general, not to try to separate the boundaries from $X^{(0)}$.

These two approaches represent extreme cases. Intermediate methods might be used wherein the boundaries of $X^{(0)}$ in a given box X are separated off under special circumstances.

It should, perhaps, be pointed out that the Newton method can delete boundary points of $X^{(0)}$ under certain circumstances.

Suppose our algorithm has produced a degenerate box X which is all or part of a face of $X^{(0)}$. In this degenerate $(n-1)$-dimensional box, the Newton method can delete points which are not in the $(n-2)$-dimensional boundary of the face of $X^{(0)}$. Such deleted points are, of course, on the boundary of $X^{(0)}$.

In some examples, we shall know a priori that the global minimum is a stationary point. In this case we are free to delete boundary points by any of our procedures.

11. The list of boxes.

When we begin our algorithm, we shall have a single box $X^{(0)}$. We apply the four procedures described in Sections 5, 6, 7, and 9 to this box. It is possible that none of these procedures can delete any of $X^{(0)}$. If so, we divide $X^{(0)}$ in half in a direction of its maximum width. We put one of these new boxes in a list L to be processed and work on the other. These and subsequent boxes may also have to be subdivided, thus adding to the list L of boxes yet to be processed. If boundary points are handled appropriately, the four procedures described in Sections 5, 6, 7 and 9 can each produce more than one new sub-box; and all but one are added to the list. Thus the number of boxes in the list tends to grow initially.

Eventually, however, the boxes become small and often a box is entirely eliminated. Thus the number of boxes in the list eventually decreases to one, or just a few, or to none at all when ϵ_1 is chosen to be nonzero.

12. Subdividing a box.

In the initial stages of our algorithm, we shall be applying it to large boxes. For example, we begin by applying it to the entire initial box $X^{(0)}$. Thus it could be that, for a given box X, none of the procedures described in Sections 5, 6, 7, and 9 can delete any of X. When this occurs, we wish to subdivide X.

We could subdivide each component X_i of X into two parts. But this would give rise to 2^n sub-boxes. To prevent generation of too many sub-boxes, we shall divide only one component in half. It is best to subdivide the largest component X_i to prevent generation of a long, narrow box.

Suppose we divide $X_i = [x_i^L, x_i^R]$ in half giving two new boxes X' and X'' whose i-th components are $X_i' = [x_i^L, \bar{x}_i]$ and $X_i'' = [\bar{x}_i, x_i^R]$, respectively, where $\bar{x}_i = (x_i^L + x_i^R)/2$. The boxes X' and X'' have a boundary in common at $x_i = \bar{x}_i$. If f had a global minimum on this common boundary, we would subsequently find it twice. This is unlikely to be the case. To avoid having the same points in two boxes, we could define one of them in terms of a half open interval. Thus we could define $X_i' = [x_i^L, \bar{x}_i)$.

It is simpler to always use closed intervals. The extra work of keeping track of whether an interval contains a given endpoint is probably not worth the effort. In practice, we have elected to avoid this problem. Thus we have always used closed intervals

only. In general, this does not cause the algorithm to find a given global minimum more than once.

13. Termination.

If we have chosen $\varepsilon_1 > 0$, we can continue our algorithm until $X^{(0)}$ is entirely eliminated. As pointed out in Section 8, we then have f^* bounded to within a error ε_1. In this case, we do not obtain a bound on x^*.

If $\varepsilon_1 = 0$, we cannot eliminate all of $X^{(0)}$ since we always retain a box or boxes containing the point(s) x^* where $f(x^*) = f^*$. As pointed out above, we might also retain a box or boxes wherein f has a value very near f^* but no value equal to f^*.

Suppose that at some stage in our algorithm, the list L contains s boxes. Denote these boxes by $X^{(1)}, \cdots, X^{(s)}$. Let $X_j^{(i)}$ denote the interval defining the j-th component (dimension) of $X^{(i)}$. Let $w(X_j^{(i)})$ denote the width of the interval $X_j^{(i)}$ and let

$$w_i = \max_{1 \le j \le n} [w(X_j^{(i)})] .$$

That is, w_i is the maximum dimension of $X^{(i)}$.

We could continue processing boxes in the list L by our algorithm until

$$\sum_{i=1}^{s} w_i \le \varepsilon_2 \tag{13.1}$$

for some $\varepsilon_2 > 0$. This is provided ε_2 is chosen large enough that the prescribed precision is attainable using (say) single precision arithmetic. However, it is more convenient computationally to require only that

(13.2) $$w_i \leq \varepsilon_2$$

for each $i = 1, \cdots, s$. If $\varepsilon_1 > 0$, we set $\varepsilon_2 = 0$ for convenience.

Thus whenever a new box $X^{(i)}$ is obtained by our algorithm we can check whether (13.2) is satisfied. If so, we no longer apply our algorithm to $X^{(i)}$ (except as discussed below). If $X^{(i)}$ contains a point x^* where f is a global minimum, then the location of x^* is bounded. In fact, if $x^{(i)}$ is the center of $X^{(i)}$ and (13.2) holds for $X^{(i)}$, then

$$|x_j^{(i)} - x_j^*| \leq \varepsilon_2/2 \quad (j = 1, \cdots, n) .$$

Let s denote the number of boxes remaining and denote the boxes by $X^{(i)}$ $(i = 1, \cdots, s)$. As a final step, we want to assure that f^* is bounded sufficiently sharply. We do this as follows.

For each $i = 1, \cdots, s$ we evaluate $\bar{f}(X^{(i)})$; that is we evaluate f with interval arguments $X_j^{(i)}$ $(j = 1, \cdots, n)$. The result, say $[F_i^L, F_i^R]$ contains the range of f for all $x \in X^{(i)}$, but will not be sharp, in general. However, if ε_2 was chosen to be small, the interval result should be "close to sharp" since ε_2 is an upper bound on the largest dimension of any box $X^{(i)}$; and the smaller $X^{(i)}$ is, the sharper $[F_i^L, F_i^R]$ is . (See [9].) Therefore, it is generally not necessary to use special procedures to sharpen the computed interval.

Since $[F_i^L, F_i^R]$ contains the range of $f(x)$ for all $x \in X^{(i)}$, we have

$$F_i^L \le f(x) \le F_i^R$$

for any $x \in X^{(i)}$. Denote

$$\underline{f} = \min_{1 \le i \le s} F_i^L$$

Then

$$\underline{f} \le f(x)$$

for any x in any of the boxes $X^{(i)}$ $(i = 1, \cdots, s)$. Therefore, since any global minimum must occur at a point x^* lying in one of the boxes $X^{(i)}$, we have $\underline{f} \le f^*$. But also $f^* \le \bar{f}$ (as discussed above) and hence

$$\underline{f} \le f^* \le \bar{f} . \tag{3.3}$$

We thus have bounds on f^*. However, they may not be sharp enough since our specified requirement is to bound f^* to within, say, ε_0; and it may be that $\overline{f} - \underline{f} > \varepsilon_0$. If this is the case, we shall improve our bounds. To do this, we find a value j for which $\underline{f} = f_j^L$. We apply our main algorithm to $X^{(j)}$. This will increase f_j^L, in general. It might also decrease $\overline{f}$. For exact interval arithmetic, this must decrease $\overline{f} - f_j^L$ since $X^{(j)}$ will be reduced in size (even if it is merely subdivided). Repeating this step for each j such that $\underline{f} = f_j^L$, we must decrease $\overline{f} - \underline{f}$ (at least, if exact interval arithmetic is used) and hence eventually have

$$\overline{f} - \underline{f} \leq \varepsilon_0$$

so that f^* is bounded to sufficient accuracy since (13.3) holds.

Because of rounding errors, we cannot reduce $\overline{f} - \underline{f}$ arbitrarily, in practice. Hence we assume ε_0 is chosen commensurate with achievable accuracy using (say) single precision arithmetic.

We also require that

$$F_i^R - F_i^L \leq \varepsilon_3$$

for each box $X^{(i)}$ $(i = 1, \cdots, s)$. For convenience, we can choose $\varepsilon_3 = \varepsilon_0$ to reduce the number of quantities to be specified. Note that $F_i^L \leq \overline{f}$ since otherwise $f(x) > \overline{f}$ for all $x \in X^{(i)}$ in which case $X^{(i)}$ can be deleted. Hence

$$F_i^R \leq \overline{f} + \varepsilon_3 \leq \underline{f} + \varepsilon_0 + \varepsilon_3 \leq f^* + \varepsilon_0 + \varepsilon_3$$

for every $i = 1, \cdots, s$. That is, every remaining box contains a point x at which $f(x)$ differs from f^* by no more than $\varepsilon_0 + \varepsilon_3$.

14. The steps of the algorithm.

We now describe the steps involved in our algorithm. Initially, the list L of boxes to be processed consists of a single box $X^{(0)}$. In general, divide the list L into the list L_1 of intervals $X^{(i)}$ satisfying the condition $w_i \le \varepsilon_2$ (see (13.2)) and a list L_2 which do not satisfy this condition.

We assume we have evaluated f at the center of $X^{(0)}$ and thus obtained an initial value for $\overline{f}$. The subsequent steps are to be done in the following order except as indicated by branching.

(1) Of the boxes in L_2, choose one which has been in L_2 longest. Call it X. If L_2 is empty, go to step (11) if $\varepsilon_1 > 0$. If L_2 is empty and $\varepsilon_1 = 0$ choose a box which has been in L_1 longest and go to step 12. If both L_1 and L_2 are empty, print the bounds $\overline{f} - \varepsilon_1$ and $\overline{f}$ on f^* and stop.

(2) Check for monotonicity. Evaluate $g(X)$ as described in Section 9. For $i = 1, \cdots, n$, if $g_i(X) > 0$ (< 0) and the boundary of X at $x_i = x_i^L$ $(= x_i^R)$ does not contain a boundary point of $X^{(0)}$, delete X and go to step (1). Otherwise, if $g_i(X) \ge 0$ (≤ 0), replace $X_i = [x_i^L, x_i^R]$ by $[x_i^L, x_i^L]$ $([x_i^R, x_i^R])$. Rename the result X again.

(3) Test for non-convexity as in Section 5. Let X' denote the smallest box in X containing all the boundary points of $X^{(0)}$ which lie in X. If $X' = X$ go to step 4. Otherwise, evaluate $h_{ii}(X_1, \cdots, X_n)$ for $i = 1, \cdots, n$. If the resulting interval is strictly negative for any value of i, replace X by X'.

If X' is empty, go to step 1 . If X' is not empty, put it in the list L and go to step 1.

(4) Begin use of the quadratic method of Section 7. For those values of $i = 1, \cdots, n$ for which the left endpoint of $H_{ii}(X)$ is non-negative, solve the quadratic for the interval Y_i to replace X_i . Rename the result X_i .

(5) Begin use of the Newton method. For those values of $i = 1, \cdots, n$ for which $0 \notin [BJ(X)]_{ii}$, solve for the new interval to replace X_i . Rename the result X_i . For a given value of i , omit this step if a reduction of X_i will delete boundary points of $X^{(0)}$ from the box X .

(6) Complete the quadratic method. For those values of i not used in step 4, solve the quadratic for Y_i . If Y_i is a single interval, replace X_i by Y_i , renaming it X_i . Otherwise save Y_i for use in step 8.

(7) Complete the Newton method. For those values of i not used in step 5, solve for the new set (say) Y_i' . For a given value of i , omit this step if a reduction of X_i will delete boundary points of $X^{(0)}$ from the box X . If Y_i' is a single interval, replace X_i by Y_i' , renaming it X_i .

(8) Combine the results from the quadratic and Newton methods for those components X_i for which both methods divided X_i into two sub-intervals. That is, find the intersection Y_i'' of Y_i from step 6 and Y_i' from step 7. If Y_i'' is composed of three intervals, replace it by either Y_i or Y_i' , whichever has the

smallest intersection with X_i . Of all the Y_i" , save the one (or two or three) which deletes the largest subinterval of X_i . That is, save that Y_i" whose complement in X_i is largest. Let j be its index. For all relevant values of $i \neq j$, replace Y_i" by X_i , that is, ignore the fact that part of X_i could be deleted.

(9) If Y_j" exists. That is, if at least one interval X_j was divided into two sub-intervals, say $Y_j^{(1)}$ and $Y_j^{(2)}$, subdivide the box X into two sub-boxes. These sub-boxes will have the same components X_i as X except one will have j-th component $Y_j^{(1)}$ and the other will have j-th component $Y_j^{(2)}$. If no such Y_j" exists, we may wish to subdivide the current box. Let X denote the box chosen in step 1 and let X" denote the current box resulting from applying steps 2 through 8 to X . If the improvement of X" over X is so small that (say)

$$w(X") > .75w(X),$$

then divide X" in half in its greatest dimension.

(10) Evaluate f at the center of the box or boxes resulting after step 9. Update $\bar{f}$ as described in Section 4. Put the box(es) in the list L and go to step 1.

(11) Evaluate $f(X^{(i)})$ for each remaining box $X^{(i)}$ in L. Denote the result by $[F_i^L, F_i^R]$. If $F_i^R - F_i^L > \varepsilon_0$ for any value of i, use $X^{(i)}$ for X and go to step 4. If $F_i^R - F_i^L \leq \varepsilon_0$ for all $i = 1, \cdots, s$, find

$$\underline{f} = \min_{1 \leq i \leq s} F_i^L.$$

Then print the bounds $\underline{f}$ and $\bar{f}$ on f* and stop.

For $\varepsilon_1 > 0$, these steps bound f^* to within an error ε_1. If $\varepsilon_1 = 0$, they found f^* to within ε_0; they bound x^* to within ε_2; and they assure that for any point x in any final box, $f(x)$ exceeds f^* by no more than $\varepsilon_0 + \varepsilon_3$.

In this step, we sometimes branch to step 4. Note that we could go to step 2, but it is unlikely that either step 2 or step 3 will be helpful. This is because we expect each box remaining at this stage to contain a minimum.

15. A numerical example.

We now illustrate the steps of our algorithm. We shall consider the so-called three hump camel function.

$$f(x) = 2x_1^2 - 1.05x_1^4 + \tfrac{1}{6}x_1^6 - x_1x_2 + x_2^2 \tag{15.1}$$

which has three minima and two saddle points. The gradient $g(x)$ has components

$$\begin{aligned} g_1(x) &= 4x_1 - 4.2x_1^3 + x_1^5 - x_2 , \\ g_2(x) &= 2x_2 - x_1 . \end{aligned} \tag{15.2}$$

The interval Jacobian $J(X)$ (see Section 3) has elements

$$\begin{aligned} J_{11}(X) &= 4 - 12.6X_1^2 + 5X_1^4 , \\ J_{12}(X) &= J_{21}(X) = -1 , \\ J_{22}(X) &= 2 . \end{aligned} \tag{15.3}$$

As described in [4], a better formulation for $J(X)$ could be derived which would give smaller intervals, in general. However, we shall use the simpler form given here.

Suppose that the box we choose in step 1 has first component $X_1 = [1, 1.1]$. In step 2 we find that, whatever X_2 is,

$$h_{11}(X_1,X_2) = [-5.196,-2.55] .$$

Since this is strictly negative, we know that f does not have a minimum in the interval X_1 for any value of x_2. Hence if X does not contain a boundary point of $X^{(0)}$, we can delete all of X.

Now suppose the box chosen in step 1 has components $X_1 = [2,3]$ and $X_2 = [0,1]$. We find

$$h_{11}(X_1,X_2) = [-29.4,358.6] , \quad h_{22}(X_1,X_2) = 2 .$$

Since neither interval is negative, we cannot say that f is not convex in X. Hence we go to step 3.

In step 3, we evaluate $g(X)$ obtaining

$$g_1(X) = [-74.4,221.4] , \quad g_2(X) = [-3,0] .$$

We see that $g_2(x)$ is non-positive for all $x \in X$ and hence f is smallest in X for $x_2 = 1$. Thus we can replace X by the degenerate box X' with components $X_1' = [2,3]$ and $X_2' = [1,1]$.

If the box X had had components $X_1 = [0,1]$ and $X_2 = [2,3]$, we would have obtained

$$g_1(X) = [-7.2,3] \quad g_2(X) = [3,6] .$$

In this case $g_2(x)$ is strictly positive and we can eliminate all of X unless the boundary of X at $x_2 = 2$ contains a boundary point of $X^{(0)}$. Suppose $X_1^{(0)} = [0,1]$ and $X_2^{(0)} = [1,3]$. Then $X^{(0)}$ has boundary points at $x_2 = 2$ for $x_1 = 0$ and 1. We could thus delete all of X except the points $(0,2)$ and $(1,2)$. This is simple to do in this two-dimensional problem. In higher dimensions, it might be simpler to retain the entire boundary at $x_2 = 2$.

Now suppose that X is given by $X_1 = X_2 = [0,1]$. Then $g_1(X) = [-5.2,5]$ and $g_2(X) = [-1,2]$ so that we do not have monotonicity. Therefore, we do step 4 which involves the quadratic method of Section 7.

For this box, we obtain

$$H_{11}(X) = J_{11}(X) = [-8.6,9], \quad H_{21}(X) = 2J_{21}(X) = -2, \quad H_{22}(X) = J_{22}(X) = 2$$

The center of the box is at $x = (.5,.5)$. We wish to evaluate $f(x)$ and $g(x)$. We cannot obtain $f(x)$ exactly using finite precision decimal arithmetic. Let us use five significant decimal digits and evaluate $f(x)$ using interval arithmetic to bound rounding errors. Thus we replace the coefficient $1/6$ by $[.16666, .16667]$ and obtain

$$f(x) = [.43697, .43699].$$

We also obtain

$$g(x) = \begin{bmatrix} [1.0062, 1.0063] \\ .5 \end{bmatrix}.$$

Suppose we have previously obtained $\bar{f} = .2$ and that we choose $\varepsilon_1 = 0$. To do step 4, we wish to solve (7.2) for points $y \in X$ where we know that $f(y) > \bar{f}$ does not hold and hence $f(y) \leq \bar{f}$ might hold. If we first tried to solve for Y_1, we would rewrite (7.2) in the form (7.4). However, the left endpoint of $H_{11}(X)$ is less than zero. Hence, solving (7.4) would give rise to two semi-infinite intervals Therefore, we defer this operation until step 6 and first solve for Y_2 which will be a single interval since the "left endpoint" of $H_{22}(X)$ is positive.

We solve for Y_2 using (7.5). As we have not yet solved for Y_1, we use X_1 in its place. Substituting into (7.5), we obtain

$$[-1.2412,\ 1.9652] + [0,1]\tilde{y}_2 + \tilde{y}_2^2 \leq 0.$$

From equation (7.8), the solution set is

$$\tilde{Z}_2 = [-1.7212,\ 1.1141] .$$

Hence

$$Z_2 = x_2 + \tilde{Z}_2 = [-1.2212,\ 1.6141]$$

and

$$Y_2 = Z_2 \cap X_2 = X_2 .$$

Thus we have not deleted any of X_2.

Next we do step 5 which applies that part of the Newton method which generates a single interval. The interval Jacobian is

$$J(X) = \begin{bmatrix} [-8.6,\ 9] & -1 \\ -1 & 2 \end{bmatrix} .$$

The center of this interval matrix is

$$J_c = \begin{bmatrix} .2 & -1 \\ -1 & 2 \end{bmatrix}$$

whose inverse is (approximately)

$$B = \begin{bmatrix} -3.3333 & -1.6667 \\ -1.6667 & -.33333 \end{bmatrix} .$$

For simplicity of exposition, we shall compute $BJ(X)$ explicitly. We obtain from (6.2) (with ξ replaced by X),

$$\begin{bmatrix} [-4.1877, -4.1872] \\ [-1.844, -1.8436] \end{bmatrix} + \begin{bmatrix} [-28.334, 30.334] & -.0001 \\ [-14.668, 14.668] & [1, 1.0001] \end{bmatrix} (y-x) = 0 . \tag{15.4}$$

We try to improve X_2 first rather than X_1 because $[BJ(X)]_{11}$ contains zero while $[BJ(X)]_{22}$ does not. Thus the first equation of (15.4) gives rise to two new intervals while the second equation does not.

The second equation is

$$[-1.844, -1.8436] + [-14.668, 14.668](X_1-x_1) + [1, 1.0001](y_2-x_2) = 0$$

where we have replaced y_1 by X_1. Solving for y_2, we obtain the interval

$$Z_2 = [-4.9904, 8.678] .$$

The intersection $Y_2 = Z_2 \cap X_2$ equals X_2 so again no improvement has been made.

Step 6 prescribes that we use the quadratic method to try to improve those components of X not solved for in step 4. We solve (7.5) for points y_1 where $f(y) \le \overline{f}$. We would use Y_2 in place of X_2; but they are equal. Substituting into (7.5), we obtain

$$[.08697, .83699] + [.5062, 1.5063]\tilde{y}_1 + [-4.3,4.5]\tilde{y}_1^2 \le 0$$

Using equation (7.11), we find that this quadratic has the solution set

$$\tilde{Z}_1 = [-\infty, -.10093] \cup [.42555, \infty].$$

Thus

$$Z_1 = \tilde{Z}_1 + x_1 = [-\infty, .39907] \cup [.92555, \infty]$$

and

$$Y_1 = Z_1 \cap X_1 = [0, .39907] \cup [.92555, 1]$$

note we have eliminated a subinterval of length .52648 from X_1.

In step 7, we use the Newton method to try to improve X_1. We solve the first equation of (15.4),

$$[-4.1877, -4.1872] + [-28.334, 30.334](y_1 - x_1) - .0001(X_2 - x_2) = 0,$$

where we have now replaced Y_2 by X_2. Solving for y_1, we obtain the two semi-infinite intervals

$$Z_1^{(3)} = [-\infty, .35223], \quad Z_1^{(4)} = [.63803, \infty].$$

Their intersections with X_1 are (say) $Y_1^{(3)}$ and $Y_1^{(4)}$ where

$$Y_1^{(3)} = [0, .35223], \quad Y_1^{(4)} = [.63803, 1].$$

We wish to combine the results obtained using the quadratic method and the Newton method. Thus we retain the intersection of

$$Y_1^{(1)} \cup Y_1^{(2)} \quad \text{and} \quad Y_1^{(3)} \cup Y_1^{(4)} \quad \text{which is}$$

$$[0, .35223] \cup [.92555, 1] .$$

We have deleted a substantial portion of the original box X. The remaining points compose the two boxes

$$\begin{bmatrix} [0,.35223] \\ [0, 1] \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} [.92555, 1] \\ [0,1] \end{bmatrix} .$$

In subsequent steps, our method would be applied separately to each of these boxes.

16. Computational results

We now describe some computational results obtained using the algorithm described above. The computations were done on the Amdahl 470V/6-II computer. In each case, we assumed it was known that the global minimum occurred in the interior of the initial box. This speeds up the algorithm since boundary points need not get special treatment.

This is consistent with the fact that we are really considering the unconstrained case. We intend to treat the constrained case in a later paper.

We give results for only one example which typifies the problems in two and three dimensions which we have used. The example is the three hump camel function given by (15.1).

This function has its global minimum at the origin. It has two local minima at approximately $[\pm 1.75, \pm 0.87]$ and two saddle points at approximately $[\pm 1.07, \pm 0.535]$. Our initial box was defined by $X_1 = X_2 = [-2, 4]$ which contains all these points. We chose $\epsilon_1 = 0$ and $\epsilon_0 = \epsilon_2 = \epsilon_3 = 10^{-4}$.

We find that after eight steps of our algorithm, we have six sub-boxes in our list. In the next step a sub-box is entirely eliminated and after the fifteenth step, only one sub-box remains but its width exceeds ε_2. After an additional step, we obtain the final box

$$X = \begin{bmatrix} [-5.78 \times 10^{-7},\ 7.11 \times 10^{-6}] \\ [-2.91 \times 10^{-7},\ 3.56 \times 10^{-6}] \end{bmatrix} .$$

Here and in the following, we record results to only three decimals. This box satisfies the error criterion requiring its width to be less than ε_2. Evaluating f at the center of this box, we obtain $\overline{f} = 1.12 \times 10^{-10}$.

As prescribed in step 12, we evaluate $f(X)$ and obtain $[-1.24 \times 10^{-11},\ 7.57 \times 10^{-10}]$. Thus $\underline{f} = -1.24 \times 10^{-11}$ and

$$f^* \in [\underline{f}\ ,\ \overline{f}] = [-1.24 \times 10^{-11},\ 1.12 \times 10^{-10}].$$

Since $\overline{f} - \underline{f} < \varepsilon_0$, we have f^* bounded to the prescribed tolerance. If we approximate f^* by

$$(\underline{f} + \overline{f})/2 = 4.98 \times 10^{-11},$$

then we know that the error is at most 6.22×10^{-11} in magnitude.

If we approximate x^* by the center $(3.27 \times 10^{-6},\ 1.63 \times 10^{-6})$, then we know that the error in x_1^* is less than 3.85×10^{-6} and the error in x_2^* is less than 1.93×10^{-6}.

We have obtained x^* to far more accuracy than required because of the rapid rate of convergence of the interval Newton method used. The bound on f^* is much better than required simply because a given error bound on x^* automatically yields a much better bound on f^* .

We also used this example with an initial box of width 2×10^6. This case required 46 steps to run to completion. This illustrates that if we use a very large box to assure containment of x^*, the computing time need not increase drastically.

17. Conclusion.

We have presented an algorithm for solving the unconstrained minimization problem assuming we have an initial box which is known to contain the minimum.

It would certainly be possible to construct a highly oscillatory function for which our algorithm would be prohibitively slow. However, it has converged adequately rapidly for the test problems on which we have tried it. (See Section 16.)

We have assumed $f(x) \in C^2$. The global minimization problem can also be solved for $f(x) \in C^1$. In this case, the Newton method cannot be used. The quadratic method can be replaced by a corresponding linear method in which we find points y at which $f(y) \leq \bar{f}$. This is done by noting that if $x \in X$ and $y \in X$, then

$$f(y) = f(x) + (y - x)^T g(\xi)$$

for some $\xi \in X$. Thus we can solve for the approximate points y from

$$f(x) + (y - x)^T g(X) \leq \bar{f}$$

For the problems of low dimension on which we have used this method, it was less efficient then the quadratic method described in Section 7. We do not know the relative efficiencies for large n.

It is possible to solve the global optimization case when $f(x)$ is only continuous but not differentiable. However, our algorithm is very slow. It entails a different approach that we hope to describe in another paper.

The nonlinear constrained optimization problem can also be solved by interval methods. An extension of our algorithm is required. Our experience in this case is for hand calculations only. A difficulty exists (currently) when it is difficult to find a point in the neighborhood of x^* which is without question, feasible.

One of the virtues of interval arithmetic is that it is usually possible to formulate an iterative algorithm in such a way that it stops automatically when the best possible result has been obtained for the finite precision arithmetic used. We plan to do this for our algorithm and thus preclude the need for specifying ε_0, ε_1, ε_2, and ε_3.

Acknowledgment

The author is greatly indebted to Thomas Kratzke and Saumyendra Sengupta for their assistance in programming and debugging the computer program with which the data in this paper was obtained.

REFERENCES

1. Dixon, L. C. W., and G. P. Szegö: Towards Global Optimization. North Holland, (1975)

2. Dixon, L. C. W., and G. P. Szegö: Towards Global Optimization 2. North Holland, (1977)

3. Hansen, Eldon: On solving systems of equations using interval arithmetic. Math. Comp., 22, 374-384, (1968)

4. Hansen, Eldon: Interval forms of Newton's method. Computing, 20, 153-163, (1978)

5. Hansen, Eldon: Global optimization using interval analysis - the one-dimension case. Submitted to Jour. Optimiz. Theo. Applic.

6. Hansen, Eldon: Bounding solutions of systems of equations using interval analysis. To be submitted.

7. Hansen, Eldon, and Roberta Smith: Interval arithmetic in matrix computations, II. SIAM J. Num. Anal., 4, 1-9, (1967)

8. Krawczyk, R: Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. Computing, 4, 187-201, (1969)

9. Moore, R. E.: Interval Analysis. Prentice-Hall, (1966)

10. Moore, R. E.: On computing the range of a rational function of n variables over a bounded region. Computing, 16, 1-15, (1976)

11. Moore, R. E.: A test for existence of solutions to nonlinear systems. SIAM J. Num. Anal., 14, 611-615, (1977)

12. Moore, R. E.: A computational test for convergence of iterative methods for nonlinear systems. SIAM J. Num. Anal., 15, 1194-1196, (1978)

13. Nickel, Karl: On the Newton method in interval analysis. Mathematics Research Center Report 1136, University of Wisconsin (1971)

14. Skelboe, S.: Computation of rational interval functions, BIT, 14, 87-95, (1974)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) Unclassified

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR- 79-1227	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): GLOBALMIN - A COMPUTER PROGRAM FOR GLOBAL OPTIMIZATION	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Eldon R. Hansen	8. CONTRACT OR GRANT NUMBER(s): -F49620-76-C-0003
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Lockheed Palo Alto Research Laboratory, 3251 Hanover Street, Palo Alto, CA 94304	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A3
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: 15 November 1979
	13. NUMBER OF PAGES: 122
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Global optimization
Interval arithmetic
Unconstrained optimization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The implementation of an algorithm that provides infallible bounds on the global minimum of any twice continuously differentiable real function of n real variables on a closed, bounded domain is described. The algorithm also provides infallible bounds on the location of the global minimum. The algorithm uses interval arithmetic and requires the availability of several fundamental interval arithmetic processors for its operation.

DD FORM 1 JAN 73 1473

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)