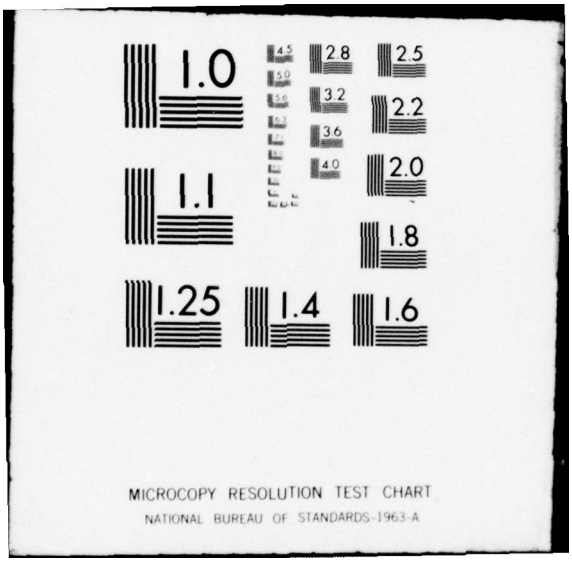
AD-A069 858 UNIVERSITY OF SOUTHWESTERN LOUISIANA LAFAYETTE DEPT O--ETC F/G 9/2
IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE. --ETC(U)
APR 79 S PODLASKA-LANDO, E K REUTER DACA39-76-M-0249

UNCLASSIFIED WES-TR-O-79-1-2 NL

| OF |
AD
A069858



END
DATE
FILMED
7 -79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A069858

LEVEL II

TECHNICAL REPORT O-79-1

IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE

Report 2

THE HONEYWELL MULTICS SYSTEM

by

S. Podlaska-Lando, Eric K. Reuter, Bruce D. Shriver

Computer Science Department
University of Southwestern Louisiana
Lafayette, La. 70504

DDC
RECEIVED
JUN 14 1979
C

April 1979
Report 2 of a Series

Approved For Public Release; Distribution Unlimited



Prepared for Office, Chief of Engineers, U. S. Army
Washington, D. C. 20314
Under Contract Nos. DACA39-76-M-0249 and DACA39-77-M-0101
Monitored by Automatic Data Processing Center
U. S. Army Engineer Waterways Experiment Station
P. O. Box 631, Vicksburg, Miss. 39180

DDC FILE COPY

79 06 11 067

Destroy this report when no longer needed. Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

This program is furnished by the Government and is accepted and used by the recipient with the express understanding that the United States Government makes no warranties, expressed or implied, concerning the accuracy, completeness, reliability, usability, or suitability for any particular purpose of the information and data contained in this program or furnished in connection therewith, and the United States shall be under no liability whatsoever to any person by reason of any use made thereof. The program belongs to the Government. Therefore, the recipient further agrees not to assert any proprietary rights therein or to represent this program to anyone as other than a Government program.

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Technical Report O-79-1	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): IMPLEMENTATION AND EVALUATION OF INTERVAL ARITHMETIC SOFTWARE, Report 2, The Honeywell MULTICS System.	5. TYPE OF REPORT & PERIOD COVERED: Report 2 of a series
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): S. Podlaska-Lando, Eric K. Reuter, Bruce D. Shriver	8. CONTRACT OR GRANT NUMBER(s): Contract Nos. DACA39-76-M-0249, DACA39-77-M-0101
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Southwestern Louisiana, Computer Science Department, Lafayette, La. 70504	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Intergrated Software Research & Development Program, AT11
11. CONTROLLING OFFICE NAME AND ADDRESS: Office, Chief of Engineers, U. S. Army, Washington, D. C. 20314	12. REPORT DATE: April 1979
	13. NUMBER OF PAGES: 86
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): U. S. Army Engineer Waterways Experiment Station, Automatic Data Processing Center, P. O. Box 631, Vicksburg, Miss. 39180	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

WES TR-O-79-1-2

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)

Algorithms
Computer systems programs
Evaluation
Honeywell MULTICS System
Interval arithmetic

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

This is Report 2 of a series entitled "Implementation and Evaluation of Interval Arithmetic Software." The series concerns implementation and evaluation of an interval arithmetic software package on six different computer systems. The other reports to be published in this series are:

Report 1: The State of the Interval: Evaluation and Recommendations

(Continued)

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

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

411 222

20. ABSTRACT (Continued)

Report 3: The Honeywell G635 System

Report 4: The IBM 370, DEC 10, and DEC PDP-11/70 Systems

Report 5: The CDC CYBER 70 System

Interval arithmetic can be used to determine the precision of the arithmetic required to guarantee a given precision in the results of an algorithm. In general, whether using interval or regular arithmetic, the greater the precision the longer the run time required for a given algorithm.

A 56 decimal digit version of the original MULTICS interval package was implemented on the MULTICS system.

It is concluded that the use of single precision and 56 decimal digit extended precision interval arithmetic can, at times, be extremely useful. The testing showed that, when using the 56 decimal digit data type, much better bounds were obtained for the results than when using the single precision interval data type.

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

PREFACE

In December 1975, the Automatic Data Processing (ADP) Center of the U. S. Army Engineer Waterways Experiment Station (WES), Vicksburg, Miss., submitted a proposal to implement and evaluate interval arithmetic, a software system for digital computer numerical analysis, on the Corps of Engineers' primary engineering computer--the WES Honeywell G635. The proposal was later expanded to include the implementation and evaluation of an interval arithmetic software package on six different computer systems. Engineering and scientific data problems were selected to be used on each of the six computers with the interval arithmetic software.

The work was funded by the Office, Chief of Engineers, U. S. Army, through the Integrated Software Research and Development (ISRAD) Program, AT11, Engineering Software Research.

This is Report 2 of a series entitled "Implementation and Evaluation of Interval Arithmetic Software." The other reports to be published in the series are:

Report 1: The State of the Interval: Evaluation and Recommendations

Report 3: The Honeywell G635 System

Report 4: The IBM 370, DEC 10, and DEC PDP-11/70 Systems

Report 5: The CDC CYBER 70 System

This report was written by Mr. S. Podlaski-Lando, Mr. Eric K. Reuter, and Dr. Bruce D. Shriver of the Computer Science Department, University of Southwestern Louisiana, Lafayette, La. Their work was performed under Contract No. DACA39-76-M-0249, dated 28 April 1976, and Contract No. DACA39-77-M-0101, dated 24 February 1977. The work concerned implementation and evaluation of an interval arithmetic software system on the Honeywell MULTICS system and the implementation and evaluation of a 56-decimal digit interval arithmetic system on the Honeywell MULTICS 66/80 system.

Dr. J. Michael Yohe, Director of Academic Computing Services, University of Wisconsin-Eau Claire, developed and wrote the interval arithmetic software package which was implemented on each of the six

computer systems. Dr. Fred D. Crary, formerly with the U. S. Army Mathematics Research Center, University of Wisconsin-Madison, developed and wrote the AUGMENT precompiler which was implemented on each computer system as a front-end to the interval arithmetic software package. Dr. Yohe and Dr. Crary are specially thanked and recognized for their technical contributions and assistance.

Mr. James B. Cheek, Jr., formerly with the ADP Center, WES, provided initial impetus and guidance for the project. Mr. Fred T. Tracy, ADP Center, WES, provided expert advice and technical guidance during the project. Dr. N. Radhakrishnan, Special Technical Assistant, ADP Center, furnished technical guidance and general project supervision. The project and the report were monitored by Mr. William L. Boyt under the general supervision of Mr. D. L. Neumann, Chief of the ADP Center.

Directors of WES during the project and the preparation of the report were COL G. H. Hilt, CE, and COL J. L. Cannon, CE. Technical Director was Mr. F. R. Brown.

Copies of the other reports of the series, computer listings of the interval program and of AUGMENT for each computer system, and runs of the benchmarks for each computer system may be obtained from the ADP Center, WES.

CONTENTS

	Page
PREFACE	1
MAIN TEXT	
1.0 General Considerations on the Computer Implementation of Interval Arithmetic	5
1.1 Interval Valued Functions	6
2.0 The Implementation of the MRC Interval Arithmetic Package for the MULTICS System	7
2.1 Interval Arithmetic Package Subroutines	8
2.1.1 Arithmetic Operations	9
2.1.2 Exponentiation Operations	11
2.1.3 Conversion Functions	11
2.1.4 Comparisons	12
2.1.5 Basic External Functions	12
2.1.5.1 Accuracy testing	13
2.1.5.2 Conditions for errors	15
2.1.5.3 Domain extention for functions exp, sinh, cosh	16
2.1.6 Supporting Functions	17
2.1.7 Input/Output Routines	18
2.1.8 Miscellaneous	19
3.0 Writing Interval Fortran Programs on MULTICS Using Interval Arithmetic	20
3.1 Definition and Use of the Interval Data Type	21
3.2 Interval Input/Output Routines	23
4.0 56-Decimal Digit Interval Fortran Work	30
5.0 Summary Tables	31
5.1 Gaussian Elimination	31
5.2 BANSOL Program	34
5.3 SESOL Program	37
5.4 SPLINE	40
5.5 Matrix Inversion	44
5.6 FFT Program	51
5.7 SLOPE Program Work	53
5.8 STRESS	58
6.0 Conclusions and Recommendations	58

CONTENTS

		Page
REFERENCES		61
APPENDIX A:	MATHEMATICAL BASIS FOR INTERVAL ARITHMETIC	A1
APPENDIX B:	DESCRIPTION OF INTERVAL FAULTS AND LIST OF ERROR MESSAGES	B1
APPENDIX C:	SUMMARY OF INTERVAL SUBROUTINE MODIFICATIONS	C1
APPENDIX D:	SAMPLE OF INTERVAL FORTRAN PROGRAM	D1
APPENDIX E:	USE OF intfor COMMAND ON MULTICS	E1
APPENDIX F:	SUMMARY STATEMENT ON RUNNING OF 'AUGMENT' ON MULTICS	F1

1.0 General Considerations on the Computer Implementation of Interval Arithmetic

The floating number system used on computers is an approximation to the real number system. In interval arithmetic, real numbers are approximated by intervals which contain the number. A brief introduction to interval arithmetic is given in Appendix A. We will represent an interval as a pair of floating point numbers stored in consecutive storage locations. The first number will be the left endpoint and the second number will be the right endpoint of the interval. Since the floating point system used on computers is an approximation to the real number system, there are many intervals whose endpoints do not have an exact representation in a particular floating point system. In this case the endpoints of the interval have to be approximated by the floating point system.

We will regard intervals as bounds on an exact but unknown real number. We would like the computer approximation to the interval to also bound the same real number. This means that if the interval $[a',b']$ is a computer approximation to the interval $[a,b]$, then we would like $[a,b] \subseteq [a',b']$. In order to insure that the preceeding set inclusion always holds, a' must be a lower bound for a and b' must be an upper bound for b. Since we want the best computer approximation to the interval, we want a' to be the greatest lower bound for a and b' to be the least upper bound for b. In this way the interval $[a',b']$ will be the smallest computer representable interval that contains $[a,b]$.

In order to obtain the smallest computer representable interval for the result of arithmetic operations on intervals, directed roundings on the computer artihmetic operations must be defined. If x is a real number and M1 and M2 are two consecutive machine representable numbers such that $M1<x<M2$ and if r is a rounding function, then r is downward directed if $r(x) = M1$ and r is upward directed if $r(x) = M2$. M1 and M2 will be the machine representable numbers that are respectively the greatest lower bound and the least upper bound for the real number x. If x is a machine representable number, then $r(x) = x$.

In general a op b, where a and b are machine representable numbers and op is one of the machine arithmetic operations, is not a machine representable number and must be rounded into a machine representable number. Algorithms for performing the machine arithmetic operations with directed roundings can be found in Yohe [2]. These operations are used to compute the endpoints of the resultant interval for a particular arithmetic operation performed on two intervals. A downward directed rounding is performed on the left endpoint and an upward directed rounding is performed on the right endpoint.

For example, in Appendix A, interval addition is defined as follows:

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

We assume now that a, b, c, and d are machine representable numbers. The computer approximation to the resultant interval is defined as follows:

$$[a,b] \bullet [c,d] = [r1(a \bullet c), r2(b \bullet d)]$$

where $\bullet$ is the machine addition operation and r1 is a downward directed rounding and r2 is an upward directed rounding.

Since the exponent range is bounded, certain faults may occur during an arithmetic operation. If the exponent becomes too small, underflow has occurred. If the exponent becomes too large, then overflow has occurred. If underflow occurs, then the true result is between zero and the smallest positive or negative representable number. In this case a directed rounding can give a valid bound. In the case of overflow, if rounding away from zero is wanted, then there is no machine representable number which can be used as a correct bound. This type of fault is known as an infinity fault.

1.1 Interval Valued Functions

A real-valued function, f, which is defined and continuous on an interval [a,b] can be extended to an interval-valued function, F, of an interval variable by defining

$$F([a,b]) = \{f(x) : x \in [a,b]\}.$$

When f is evaluated on a digital computer using machine representable approximations to the real numbers, a computer approximation, f', to f results. If F([a,b]) is an interval valued function of an interval (where a and b are machine representable numbers), then the computer approximation, F'([a,b]) is defined as an interval that contains F([a,b]).

If f' is the computer approximation of a real valued function f and f is monotonic increasing on [a,b], then

$$F'([a,b]) = [r1(f'(a)), r2(f'(b))]$$

where r1 is a downward directed rounding into a machine representable number such that $r1(f'(a)) \leq f(a)$ and r2 is an upward directed rounding into a machine representable number such that $r2(f'(b)) \geq f(b)$. Ideally we would like r1(f'(a)) to be the largest machine representable number such that $r1(f'(a)) \leq f(a)$

(i.e., a greatest lower bound) and r2(f'(b)) to be the smallest machine representable number such that r2(f'(b)) $\geq$ f(b) (i.e. a least upper bound).

If f is monotonic decreasing on [a,b], then

$$F'([a,b]) = [r1(f'(b)),r2(f'(a))]$$

If f is not monotonic on [a,b], then the interval [a,b] can be divided into disjoint subintervals; X'(i), i = 1,2,3,...,n; where the endpoints of each X'(i) are machine representable numbers and U X(i) contains all the machine representable numbers in the interval [a,b] and f is monotonic on each X'(i). In this case F'([a,b]) = U F (X'(i)).

It may not be possible to obtain the best bounds for the result of the computer approximation to the function f. The problem will be illustrated in the next section when describing the interval counterparts of the MULTICS basic external functions.

2.0 The Implementation of the MRC Interval Arithmetic Package for the MULTICS System

The interval arithmetic package and the input/output routines for interval numbers which are implemented on the MULTICS system follow the design of an interval arithmetic package implemented on the UNIVAC 1108 computer located at the Mathematics Research Center of the University of Wisconsin [1,3]. This section mainly presents the difficulties encountered when the interval arithmetic package was implemented on the MULTICS system and also the changes that were made to the original interval package implemented at MRC. Most of the changes dealt with machine dependencies.

Before the routines are described, a description of the representation of interval numbers on MULTICS will be given along with a description of the MULTICS double precision floating point format and how it impacted the realization of the interval package. The endpoints of the intervals are represented as a pair of floating point numbers stored in consecutive storage locations. The MULTICS single precision floating point format uses a 36 bit word which consists of an 8 bit 2's complement exponent, with the high order bit the sign bit, followed by a 28 bit normalized 2's complement fraction, with the high order bit the sign bit.

In the original interval package implemented on the UNIVAC 1108, the type double precision in Fortran was used extensively to trap underflow and overflow fault conditions. This could be done because the exponent range of the double precision floating point

format on the UNIVAC 1108 is greater than the single precision floating point format. Therefore, with certain precautions, results could be computed in double precision without fear of machine underflow or overflow. The underflow or overflow could then be trapped when the conversion is made to single precision. The MULTICS double precision floating point format uses a 72 bit double word which consists of an 8 bit 2's complement exponent, with the high order bit the sign bit, followed by a 64 bit normalized 2's complement fraction, with the high order bit the sign bit. The double precision floating point format has the same exponent range as the single precision format. Therefore, it is much more difficult to trap certain faults. The problem is amply illustrated in the section describing the implementation of the interval basic external functions.

2.1 Interval Arithmetic Package Subroutines

The subroutines of the MRC interval package can be divided into eight categories. These categories are arithmetic operations, exponentiation operations, conversion functions, comparison, basic external functions, supporting functions, input/output routines and miscellaneous. All of the routines in each category except the input/output category were written in Fortran at the upper level. Several of the Fortran subroutines call routines that are written in PL/I. These PL/I routines correspond for the most part to the assembler routines that were written for the UNIVAC 1108 version of the interval package and are written specifically for the MULTICS implementation. Most of the input/output routines were written in PL/I.

Each subroutine in the package will be described briefly except when changes had to be made to a particular subroutine because of some machine dependencies. In that case a more detailed description will be given. After the description of each routine a code will be given that specifies whether that particular routine was implemented in its original form from the UNIVAC 1108 version, or it was implemented with some changes from the UNIVAC 1108 version, or it was written in PL/I specifically for the MULTICS system. The codes are [MRC] for the original form with no changes, [MRC/M] for the original form modified, and [MUL] for the routines written specifically for the MULTICS system. Appendix C provides a summary of the routines in each of the above categories. A complete source listing of the MULTICS interval package can be found in [8].

2.1.1 Arithmetic Operations

The routines in this category perform the four basic arithmetic operations of addition, subtraction, multiplication, and division on interval numbers. Since we want the best computer approximation to the results of computer arithmetic operations on intervals, directed roundings on the computer arithmetic operations must be performed. The floating point hardware on the MULTICS system does not perform directed roundings. Therefore the four basic single precision floating point computer arithmetic operations of addition, subtraction, multiplication, and division had to be simulated in order to provide the correct roundings. A description of the routines that simulated the floating point computer arithmetic operations and provided the proper directed roundings and a description of the routines that perform the basic computer arithmetic operations on intervals follows. These routines perform the "best possible arithmetic" computer operations with directed roundings as described by Yohe [2]. All the routines are written in PL/I for the MULTICS system.

bpaadd: Performs single precision floating point addition. [MUL]

bpasub: Performs single precision floating point subtraction. A problem can occur in this routine because the subtraction is realized by negating the second operand and calling the bpaadd routine. The problem is negating the smallest positive representable number, because underflow will occur using the MULTICS floating point hardware. This problem can occur in general and the solution to the problem is described below in the description of the pack routine. [MUL]

bpamul: Performs single precision floating point multiplication. [MUL]

bpadiv: Performs single precision floating point division. [MUL]

brounding: Performs bounds checking and rounding on the results from the above four arithmetic operations. The rounding strategies employed are toward zero, away from zero, downward directed, upward directed, and optimal. [MUL]

unpack: This routine unpacks the floating point number from MULTICS format into a format that the bpa routines will handle. The number is made positive because the bpa routines perform their operations on signed magnitude fractions. [MUL]

normalize: This routine is used by the bpa routines to normalize the fraction. [MUL]

shift_right: This routine is used by the bpa routines to shift the fraction right when fraction overflow occurs. [MUL]

s_mgn_add: This routine performs a signed magnitude addition of two 36 bit binary integers. It is used by the bpaadd routine to add the two fractions. [MUL]

pack: This routine packs the floating point result produced by the bpa routines into the MULTICS floating point format. A problem can occur in this routine when the number to be packed is the negative of the smallest positive representable number. The bpa routines perform signed magnitude arithmetic on the fractions and a negative result is obtained in MULTICS format by negating the positive result. The negation of the smallest positive representable number will cause an underflow using the MULTICS floating point hardware because the normalized form of that number has an exponent of -129 and is therefore not representable. If the number is to be negated and it is the smallest positive representable number, then the bit pattern that represents the negative of the smallest positive representable number is assigned to the result. This number is not in true 2's complement normalized form, but represents the true value. [MUL]

The following two Fortran subroutines perform the arithmetic operations of addition, subtraction, multiplication, and division on intervals. The routines call the bpa routines described above to perform their operations on the endpoints of the intervals.

arith1: This routine performs the operations of addition and subtraction on intervals. There is an entry point for addition and an entry point for subtraction. The operations are performed on the endpoints as described in Sections 1 and 2. A slight change was made in this routine from the original routine implemented in the UNIVAC 1108 version of the interval package. The original routine only made calls to the bpaadd routine. In the case of interval subtraction the endpoints of the second interval operand were negated and the bpaadd routine was then called. This negation could cause the same problem as described above in the bpasub and pack routines. Since the problem had been taken care of in the bpasub routine, it was decided to call the bpasub routine directly if an interval subtraction was to be performed. [MRC/M]

arith2: This routine performs the operations of multiplication and division on intervals. There is an entry point for multiplication and an entry point for division. As is stated in Appendix A, the signs of the endpoints of the intervals being multiplied or divided are examined in order to determine in advance which products or quotients will be the maximum and minimum. [MRC]

2.1.2 Exponentiation Operations

The routines in this category perform various exponentiation operations involving interval, double precision, real, and integer numbers. The exact nature of the exponentiation performed will be described in the description of each Fortran routine that follows:

expon1: This routine calculates the value of an interval raised to an integer power. [MRC]

bpaxp4: This routine computes the best value of a real number raised to an integer power. It is used by expon1 to calculate the value of an interval raised to an integer power. Changes were made in this routine to correct the situation in which the fault flag may not be set correctly and to take care of the problem of negating the smallest positive representable number. [MRC/M]

expon2: This routine calculates base ** power where base is an interval number and power is real, double precision, or interval. There is an entry point for each type of exponentiation. [MRC]

2.1.3 Conversion Functions

The routines in this category perform conversions from the standard types to type interval and from interval to the standard types. The following routines are written in Fortran.

convrt: This routine has entry points to convert from integer to interval, complex to interval, real to interval, and double precision to interval. [MRC]

intc84: This routine converts from interval to integer. A change was made in the routine to set the sign of the result correctly when the maximum result needs to be set. [MRC/M]

intc85: This routine converts from interval to real. Changes were made in this routine to check for underflow

differently than in the original due to the fact that the exponent range of a double precision floating point number is not greater than the exponent range of a single precision floating point number on the H68/80 computer. [MRC/M]

intc86: This routine converts from interval to double precision. Changes were made in this routine to check for underflow differently than the original because of the same exponent range of the double precision and single precision floating point formats. [MRC/M]

intc87: This routine converts from interval to complex. [MRC]

funct3: This routine computes an interval with integer endpoints (in floating point form) which contains the interval of the argument. [MRC]

2.1.4 Comparisons

The routines in this category are the relational intrinsic functions for type interval. The following routine was written in Fortran.

relatn: This routine has entry points for the relational functions of equal, not equal, less than, less than or equal, greater than, and greater than or equal. [MRC]

2.1.5 Basic External Functions

Included in the interval package are the interval counterparts of the MULTICS basic external functions atan2, exp, alog, alog10, sin, cos, tan, asin, acos, atan, sinh, cosh and sqrt. The general method of calculation of the interval functions involves bounding the results of the corresponding double precision basic external function. For functions that are monotonic over an interval, the endpoints of the resultant interval are the result of the double precision function evaluated at the endpoints of the input interval and then properly bounded. If the function is not monotonic over the interval, then a case analysis is done by dividing the input interval into subintervals over which the function is monotonic.

The result obtained from the double precision functions must be bounded before it can be used as the endpoint of an interval. Therefore, the accuracy of the results of the double precision basic external functions are required by determining a lower bound on the number of bits of the fraction that the result is

guaranteed to have. the number of bits of the fraction that the result is guaranteed to have are required. This can be illustrated by the following example. Suppose a result is accurate to 35 bits of fraction and a 27 bit lower bound for the result is required. Assume that the 27th through 37th bits of the fraction were 10000000000. If the result were just truncated to 27 bits the 27th bit would be a 1. If however the 37th bit was one unit too large, then bits 27 through 37 would be 01111111111 and the 27th bit of the correct lower bound would be 0. It cannot be determined which case is correct.

The following general bounding technique is performed which will produce correct bounds in all cases, but not necessarily optimal bounds. If a lower bound is sought for the double precision result, then the fraction is decremented by one at or before the last bit known to be accurate. If an upper bound is sought, then the fraction is incremented by one at or before the last bit known to be accurate. The same bounding technique used in bounding the results of the arithmetic operations is then used to obtain the 27 bit fraction of the result.

The following Fortran routines compute the basic external functions for the interval type.

funct2: This routine has entry points for the following interval functions: sqrt, log, exp, log10, atan, asin, acos, tanh, sinh, cosh. Changes made to the asin and acos functions are described in Section 2.1.5.2 and changes made to the functions sinh, cosh and tanh are described in Section 2.1.5.3. [MRC/M]

funct4: This routine calculates sin(arg) and cos(arg) where arg is an interval. There is an entry point for the sine and an entry point for the cosine function. The cosine routine scales the argument so that the left endpoint is in the interval [0,2pi], and then performs a case analysis. The sine routine performs the same scaling as the cosine routine and performs a case analysis. [MRC/M]

funct5: This routine calculates tan and atan2 of an interval. The entry point is provided for either function. The tangent is calculated as follows: the argument is reduced so that the left endpoint is in the interval: [-pi/2, pi/2] and then a case analysis is performed. The atan2 routine takes two interval arguments, x and y and computes atan(x/y). [MRC/M]

2.1.5.1 Accuracy testing

To our knowledge, there is no documentation concerning the implementation of the basic external functions on MULTICS

accessible either by PL/I or Fortran. We considered three approaches to determining the accuracy of the required external function:

1) rigorous error analysis of current implementation
2) rewriting of the required routines
3) comparison of accuracy with known test data

First, the error analysis of the mathematical library routines seemed to be impossible due to the a) lack of description of the algorithms employed, b) low readability of the source programs (much of which was written in ALM - Assembly Language of MULTICS). The second possibility had to be eliminated due to the time constraints of the project and therefore the third approach had to be taken.

The testing itself was done in two stages:

stage1 - generation of input test data and evaluation of the given function

stage2 - comparison of significant digits of the result and corresponding value in the tables

"Driver" programs were written which generated test data and called the routines which were to be tested. The standard tables of functions, i.e. Handbook of Mathematical Functions by Milton Abramowitz and Irene A.Stegun [7] were used. The output was generated in decimal form and then a check was made as to the first digit that was different from the result given in the table. All digits of function values which were tested proved to be identical with corresponding digits in the Handbook. The only exception being the last digit in the Abramovitz's tables. However, the analysis of the very next digit in our results showed that in each case the error was caused by an upward rounding.

The test data had been restricted to the decimal values that can be represented exactly in the floating binary notation. Thus, we avoided the input conversion error and the function value could be obtained for the true argument. Also, we have to warn that the accuracy estimated in this way must be somewhat pessimistic. We were able to check only as many digits as were given in the standard tables. Thus, the tan function is assumed to have only 8 accurate decimal digits even though there are reasons to believe that accuracy is much greater than that.

The list of the number of decimal digits (and binary estimates as well) that are assumed to be accurate is given below.

	Accuracy	
Function	decimal	binary
sqrt	10	33
log	16	52
log10	10	33
exp	16	52
sin	17	56
cos	17	56
tan	8	27
asin	12	39
acos	12	39
atan	12	39
sinh	9	29
cosh	10	33
tanh	8	27

2.1.5.2 Conditions for errors

The Univac 1108 double precision word has an 11 bit exponent field vs. an 8 bit exponent in the single precision word. This allowed the checking for overflow and underflow faults to be done during the conversion from double to single precision format, as was stated in section 2.0. This strategy was not applicable for MULTICS due to the same size of the exponent in both the double and single precision format. In conclusion, the check for eventual fault conditions had to be made prior to the calls to the double precision functions.

The following functions could produce overflows: exp, sinh, cosh and tanh. In the MULTICS implementation, the overflow was prohibited by restricting the allowable domain of the argument to the interval [-88.028,88.028]. From this it followed that for arguments x such that abs(x)>88.028, a special action had to be taken. At this point it turned out, that the magnitudes of results produced at the endpoints were much smaller than the largest representable number. This implied that the actual domain should be extended beyond [-88.028,88.028]. The attempt was made to compute the new endpoints and either compute (if possible) or estimate the proper bounds for the left and right endpoints of the interval result. The detailed discussion of these cases will be given later on.

The occurence of underflow was detected in the double precision functions asin and acos. Analysis of the source programs revealed that an underflow condition was raised at the point of the internal function call to the atan routine. Namely,

```
asin(x) = atan(x,sqrt(-x*x+1))
acos(x) = atan(sqrt(-x*x+1),x)
```

and for x very small, the multiplication operation caused an underflow. The overflow and underflow fault conditions have been tested with a number of programs.

2.1.5.3 Domain extension for functions exp, sinh, cosh.

As we mentioned before, the MULTICS implementation of the functions exp, sinh and cosh restricts the domain to the interval [-88.028,88.028] . Let MIN denote the smallest positive , and MAX the largest positive machine representable number. The endpoints that could actually cause overflow or underflow were obtained from:

$$\log(MIN) = -89.415$$
$$\log(MAX) = 88.029$$

Thus, the domain of exp could be extended to the interval [-89.415,88.029] with exp evaluating to MIN or MAX at the endpoints. The value of exp outside of this interval and in the intervals [-88.415,-88.028] and [88.028,88.029] was evaluated as follows:

lep or rep	explep	exprep
x < -89.415	MIN, underflow	MIN
x = -89.415	MIN	MIN,round up
-89.415 < x < -88.028	MIN	exp(-88.028)
-88.028 <=x<= 88.028	exp(x),round down	exp(x),round up
88.028 < x < 88.029	exp(88.028)	MAX
x = 88.029	MAX,round down	MAX
x > 88.029	MAX, overflow	MAX, infinity

where "round down" means round to the next smaller machine representable number and "round up" means round to the next larger machine representable number.

The largest values of the functions hyperbolic sine, cosine, and tangent could be computed for the argument x=88.029 (since exp(88.029)=MAX). However, even then they were much smaller than the largest representable number. Let x denote the left or the right endpoints of the argument. For x very large we have

$$\sinh(x) = \exp(x)/2$$
$$\sinh(-x) = -\exp(x)/2$$
$$\cosh(x) = \exp(x)/2$$

The smallest positive argument that would cause an overflow was obtained from:

$$\exp(x)/2 = MAX \Rightarrow \exp(x)/2 = \exp(88.029) \Rightarrow x = 88.029+\log(2)$$

The optimal bounds for the left and right endpoint are shown in the table below. "lep" and "rep" denote the left and right endpoints of the interval argument.

hyperbolic_sine

lep or rep = x	sinhlep	sinhrep
x > 88.029+ln(2)	MAX, overflow	MAX, infinity
88.029 < x<=88.029+ln(2)	MAX/2	MAX
x = 88.029	MAX/2, round down	MAX/2
88.028 < x <88.029	sinh(88.028)	MAX/2
-88.029< x <-88.028	-MAX/2	sinh(-88.028)
x = -88.029	-MAX/2	-MAX/2, round u
-88.029-ln(2)<=x<-88.029	-MAX	-MAX/2
x < -88.029-ln(2)	-MAX, overflow	-MAX, infinity

hyperbolic_cosine

lep or rep = x	coshlep	coshrep
x >= 88.029+ln(2)	MAX, overflow	MAX, infinity
88.029<x < 88.029+ln(2)	MAX/2	MAX
x = 88.029	MAX/2	MAX/2,round up
88.028 < x < 88.029	cosh(88.028)	MAX/2

hyperbolic_tangent

lep or rep = x	tanhlep	tanhrep
x < -88.028	-1.0	-1.0,round up
x > 88.028	1.0,round down	1.0

2.1.6 Supporting Functions

The following routines perform some useful functions involving intervals. All the routines are written in Fortran.

funct1: This routine has entry points to calculate the absolute value of an interval, to store the value of one interval into another and to store the negative of one interval into another. A change was made to this routine to take care of the problem of negating the smallest positive representable number. [MRC/M]

supinf: This routine has an entry point that returns the left endpoint of an interval and an entry point that returns

the right endpoint of an interval. [MRC]

unints: This routine has an entry point that returns the union of two intervals and an entry point that returns the intersection of two intervals. [MRC]

length: This routine returns the length of an interval. A change was made in this routine to compute the length differently than in the original. The length is computed by performing a single precision subtract with an away from zero rounding strategy. This change was made in order to trap underflow or overflow. [MRC/M]

intbnd: This routine returns an interval which bounds a double precision value to a specified accuracy. [MRC]

dist: This routine computes the distance between two intervals. [MUL]

compos: This routine returns an interval that consists of the two real arguments as endpoints. [MUL]

2.1.2 Input/Output Routines

The routines in this section were designed to some extent after the I/O routines implemented for the UNIVAC 1108 version of the interval package [3]. Additional routines were included in the MULTICS version to handle scalar interval variables and a matrix of interval variables. A brief description of each I/O routine is given here. In Section 3, which describes writing interval Fortran programs, a more detailed description of the routines is provided giving the calling sequence and several examples for each routine. All of the following routines are written in PL/I.

intrdv: This routine reads interval numbers into any number of interval scalar variables. [MUL]

intrdf: This routine reads interval numbers into an interval vector. [MUL]

intrdm: This routine reads interval numbers into an interval matrix. [MUL]

intprv: This routine outputs interval numbers from any number of interval scalar variables. [MUL]

intpr: This routine outputs interval numbers from an interval vector. [MUL]

intprm: This routine outputs interval numbers from an interval matrix. [MUL]

The following routines are the supporting routines needed by the above routines in order to perform interval I/O. All the routines are written in PL/I except where noted.

convert_to_binary: This routine converts from fixed decimal to floating binary performing a specified rounding. [MUL]

convert_to_decimal: This routine converts from fixed binary to floating decimal. [MUL]

convert_fb_dec: This routine converts from floating binary to floating decimal performing a specified rounding. [MUL]

get_next_int_number: This routine reads the next interval number in the input stream making a syntax check of the number. [MUL]

round_dec: This routine performs a specified rounding of a decimal number. [MUL]

get_char: This routine returns the next character in the input stream. This routine is used by the get_next_int_number routine. It is written in Fortran. [MUL]

set_input_pointer: This routine sets the input pointer for the get_char routine in order to start it off. [MUL]

2.1.8 Miscellaneous

The following routines either fit no other category and/or are used by routines in more than one category.

intrap: This PL/I routine traps all faults that can occur during an operation involving interval operands. It is called by practically all the routines in the package after an operation is performed involving interval numbers. The intrap routine displays an appropriate error message and any arguments and then takes some action depending on the type of fault that occurred. The action taken by intrap when a fault occurs can be specified by the user or a set of default actions can be taken. Appendix B lists all the faults that can occur and the default action taken by intrap after a fault has occurred. Appendix B also

provides a description of how the user can change the action that intrap takes after a particular fault has occurred. [MUL]

bpac68: This PL/I routine converts a double precision number to a single precision number using a specified rounding strategy. The double precision number is taken to be exact. [MUL]

bpac98: This PL/I routine converts a double precision number to a single precision number where the accuracy in number of bits of the fraction of the double precision number is given. [MUL]

intas: This PL/I routine is used to allow the assignment of interval constants to interval variables in a Fortran program. A description of how this is done can be found in Section 3 describing the writing of interval Fortran programs. [MUL]

set_common: This Fortran routine is used to set up the default actions taken by the intrap routine after a fault has occurred. [MUL]

finish: This Fortran routine closes the standard Fortran input and output files and stops the program. It is called by intrap when the action specified for a particular fault is to halt the program. [MUL]

comput: This Fortran routine is used to compute the interval result of an interval function. It is called by the interval basic external function routines. [MRC]

aidint: This PL/I routine is used to return the double precision integer portion of its double precision argument. [MUL]

3.0 Writing Interval Fortran Programs on MULTICS using Interval Arithmetic

This section provides the user of the interval package with a guide to writing interval Fortran programs. An interval Fortran program is a Fortran program in which the extended data type interval is used. After an interval Fortran program has been written, it must be processed by the AUGMENT precompiler [4,5]. The AUGMENT precompiler will generate the necessary calls to the routines in the interval package. Fortran programs which contain interval variables are compiled on MULTICS by use of the intfor command which will automatically invoke the AUGMENT precompiler for the user, see Appendix E. A description is given first of how to define the interval data type in Fortran and how to use it in a program. Next a detailed description of the I/O routines

for interval numbers will be given. A sample Fortran program illustrating the interval data type can be found in Appendix D.

3.1 Definition and Use of the Interval Data Type

A variable is declared in a Fortran program as having the interval data type through the use of an interval type declaration statement. The key word for the interval type declaration statement is "INTERVAL". For example, if the statement

```
INTERVAL A,B,C(10)
```

appeared in the Fortran program, then the scalar variables A and B would have the type interval and C would be an interval vector of 10 elements. The key word "INTERVAL" can appear in any context that the standard type key words (i.e. REAL, DOUBLE PRECISION, etc.) can appear. For example, if the statement

```
IMPLICIT INTERVAL (A-Z)
```

appeared in the Fortran program, then all variables beginning with the letters A-Z would default to type interval.

All of the standard arithmetic operators are defined for the interval data type. All implicit conversions from interval to the standard types and from the standard types to interval are defined except for conversion from logical to interval and interval to logical. The interval data type always takes precedence in an implicit conversion. All relational operators are defined between interval data types. All cases of exponentiation between the interval data type and the standard types are defined except for the standard types logical and complex. There are two new operators that act on interval operands. These operators are .INSCT. which finds the intersection of two intervals and .UNION. which finds the union of two intervals. For example, if A, B, and C are of type interval, then the statement

```
A = B .INSCT. C
```

finds the intersection of B and C and assigns the result to A. If the intersection is empty then an error message is displayed.

Most of the builtin functions of standard Fortran have been implemented for the type interval. The following are the builtin functions available for use with the interval data type.

ABS - Absolute value
ACOS- Arccosine

AINT - Integer
ALOG - Log base e
ALOG10 - Log base 10
ASIN - Arcsine
ATAN - Arctangent
ATAN2 - Arctangent of x/y
COS - Cosine
COSH - Hyperbolic cosine
DBLE - Converts to double precision
EXP - Exponential
FLOAT - Converts to real
IFIX - Converts to integer
SIN - Sine
SINH - Hyperbolic sine
SQRT - Square root
TAN - Tangent
TANH - Hyperbolic tangent

All of the above builtin functions take interval arguments and return an interval result except for the IFIX, FLOAT, and DBLE functions which return respectively type INTEGER, REAL, and DOUBLE PRECISION. The FLOAT and DBLE functions return the midpoint of the interval argument as either a real or double precision number. The IFIX function returns the integer portion of the midpoint of the interval argument. The AINT function computes an interval with integer endpoints (in floating point form) which contains the interval argument.

Several builtin functions were added for the type interval. These functions are listed and described below.

COMPOS: This function takes two real arguments and returns an interval with the first argument as the left endpoint and the second argument as the right endpoint.

DIST: This function takes two interval arguments and returns the distance between the intervals as a real number.

INF: This function takes one interval argument and returns its left endpoint as a real number.

INTBND: This function takes two arguments with the first argument being double precision and the second argument integer and returns an interval that contains the double precision number. The integer specifies the number of bits of accuracy of the double precision number.

INTSCT: This function takes two interval arguments and returns their intersection. Note that INTSCT can also be used as a binary operator as described earlier.

LENGTH: This function takes one interval argument and returns its length as a real number.

SUP: This function takes one interval argument and returns its right endpoint as a real number.

UNION: This function takes two interval arguments and returns their union. Note that UNION can also be used as a binary operator as described earlier.

If a constant is to be assigned to an interval variable, then in some cases the following type of assignment statement should be used.

```
A = "interval constant"
```

where an interval constant is defined in Section 3.2. This type of assignment statement should be used if 1) the interval being assigned is not degenerate or 2) a degenerate interval is being assigned, but there will be conversion error when converting from floating decimal to floating binary. An example of this type of assignment is contained in the sample interval Fortran program in Appendix D even though it was not necessary in that case to perform that type of assignment.

3.2 Interval Input/Output Routines

An interval constant consists of 1) a pair of floating point or fixed point numeric constants enclosed in parenthesis or square brackets and separated by a comma or 2) a single floating point or fixed point numeric constant that represents a degenerate interval. The form of the floating point or fixed point numeric constant is any number acceptable as a floating point decimal numeric constant in PL/I with a maximum of 59 decimal digits. Examples of interval constants are shown below:

```
(1,2)
[3,4]
1
( 1.0e15, 3.49 )
234.08e15
0.1
(-5,-4)
[-0.1,.6]
(-328.42, 2.3e-14)
( .1,.2 )
```

An interval number enclosed in parenthesis or square brackets may have any number of blanks before or after the parenthesis or square brackets and comma. The numbers representing the

endpoints may not have any embedded blanks. An interval number that represents a degenerate interval may not have any embedded blanks. Note that a decimal number that represents a degenerate interval may not be converted into a degenerate interval internally. This is because not all fractional decimal numbers have an exact representation in floating binary. For example, the decimal number 0.1 does not have a finite representation in floating binary. Therefore the endpoints of the resultant machine representable interval will not be equal because the left endpoint must be rounded downward and the right endpoint must be rounded upward when the decimal number is converted to floating binary.

Input_Routines

The following three routines are used to read interval numbers. The numbers are input from Fortran file number 5 which is the standard Fortran input file. Any number of interval numbers can appear on each input line with 1 or more blanks separating the interval numbers. The interval numbers can be input from the terminal or from a segment through an appropriate operating system I/O attach statement.

Routine: intrdv

Purpose: Read interval numbers into any number of interval scalar variables.

Calling sequence: call intrdv (a,b,c,...,eof)

a,b,c,... (output) are interval scalar variables into which the interval numbers are to be read.

eof (output) is a logical variable that is true if end of file is encountered and is false if not.

Examples:

In the following examples assume a, b, c, and d are interval scalar variables and eof is a logical variable.

Example 1:

call intrdv (a,b,eof)

The next two interval numbers in the input stream will be read into the interval variables a and b.

Example 2:

```
call intrdv (a,b,c,d,eof)
```

The next four interval numbers in the input stream are read into the interval scalar variables a, b, c, and d.

In both of the above examples if end of file was detected, then the variable eof would be set to true, otherwise it is set to false.

Routine: intrdf

Purpose: Read interval numbers into an interval vector.

Calling sequence: call intrdf (x,i,j,eof)

x (output) is an interval vector.

i (input) is an integer variable or constant specifying the starting index of where in x the interval numbers will be placed.

j (input) is an integer variable or constant specifying the ending index of where in x the interval numbers will be placed.

eof (output) is a logical variable that is true if end of file is encountered and is false if not.

Examples:

In the following examples assume x is an interval vector that can contain a maximum of 10 interval numbers and eof is a logical variable.

Example 1:

```
call intrdf (x,1,10,eof)
```

The next ten interval numbers in the input stream will be read into the entire vector x.

Example 2:

```
call intrdf (x,2,7,eof)
```

The next 6 interval numbers in the input stream will be read into the interval locations x(2) to x(7).

In the both examples above, if end of file is encountered eof will be set to true, otherwise eof will be false.

Routine: intrdm

Purpose: Read interval numbers into an interval matrix in a row by row fashion.

Calling sequence: call intrdm (x,n,k,eof)

x (output) is an interval matrix.

n (input) is an integer variable or constant specifying the number of rows to be considered.

k (input) is an integer variable or constant specifying the number of columns to be considered.

eof (output) is a logical variable that is true if end of file is encountered and is false if not.

Examples:

In the following examples assume x is an interval matrix that can contain a maximum of 5 rows and 6 columns of interval numbers and eof is a logical variable.

Example 1:

```
call intrdm (x,5,6,eof)
```

The next 30 numbers in the input stream will be read into the entire matrix x row by row with 6 numbers per row and 5 total rows.

Example 2:

```
call intrdm (x,3,4,eof)
```

The next 12 interval numbers in the input stream will be read into the interval matrix x row by row with 4 numbers per row and 3 total rows.

In both examples above, if end of file is encountered, then eof is set to true, otherwise it is set to false.

Output_Routines

The following three routines are used to output interval numbers. The numbers are output to the PL/I standard output file, sysprint. The output can be directed to a segment either by an I/O attachment or through a file_output command.

Routine: intprv

Purpose: Output interval numbers from any number of interval scalar variables.

Calling sequence: call intprv (cc,nod,nob,width,a,b,c,....)

cc (input) is a single character representing the carriage control character. The carriage control characters are the same as in standard fortran and are listed below:

blank - single space
0 - double space
+ - suppress spacing
1 - skip to top of page

nod (input) is an integer variable or constant specifying the number of interval numbers to output per line.

nob (input) is an integer variable or constant specifying the number of blanks to insert between each interval number on each line.

width (input) is an integer variable or constant specifying the total width that each interval number will occupy in the output line. The interval number is output in the form [±.XX..XX±YY,±.XX..XX±YY] The number of significant digits output for each endpoint will be (width-13)/2.

a,b,c,... (input) are interval scalar variables to be output.

Examples:

In the following examples assume a, b, c, and d are interval variables.

Example 1:

call intprv (1h ,3,1,25,a,b,c)

The interval numbers in the interval variables a, b, and c will be output with single spacing. All three numbers will be on the same line. One blank will be between each interval number. Each interval number will occupy 25 columns providing 6 significant digits for each endpoint.

Example 2:

```
call intprv ("0",3,5,33,a,b,c,d)
```

The interval numbers in the interval variables a, b, c, and d will be printed with double spacing. a, b and c will be on one line with d on the next line. There will be five spaces between each interval number. Each interval number will occupy 33 columns providing 10 significant digits for each endpoint.

Routine: intpr

Purpose: Output interval numbers from an interval vector.

Calling sequence: call intpr (cc,nod,nob,width,x,i,j)

cc, nod, nob, and width are the same as for intprv.

x (input) is an interval vector.

i (input) is an integer variable specifying the starting index in the interval vector x from where output is to start.

j (input) is an integer variable specifying the ending index in the interval vector x where output is to stop.

Examples:

In the following examples assume x is an interval vector that can contain a maximum of 10 interval numbers.

Example 1:

```
call intpr (" ",2,10,35,x,1,10)
```

The interval numbers in the entire interval vector x will be output with single spacing. There will be two interval numbers per line. Ten blanks will be between each interval number. Each interval number will occupy 35 columns providing 11 significant digits for each endpoint.

Example 2:

```
call intpr (1h0,5,1,25,x,3,9)
```

The 7 interval numbers in interval locations x(3) to x(9) will be printed with double spacing. There will be five interval numbers per line. One blank will be between each interval number. Each interval number will occupy 25 columns providing 6 significant

digits for each endpoint.

Routine: intprm

Purpose: Print interval numbers from an interval matrix.

Calling sequence: call intprm (cc,nod,nob,width,x,n,k)

cc, nod, nob, and width are the same as for intprv.

x (input) is an interval matrix.

n (input) is an integer variable or constant specifying the number of rows of the interval matrix to output.

k (input) is an integer variable or constant specifying the number of columns of the interval matrix to output.

Examples:

In the following examples assume x is an interval matrix that can contain a maximum of 7 rows and 5 columns of interval numbers.

Example 1:

```
call intprm (" ",5,1,25,x,7,5)
```

The interval numbers in the entire interval matrix x will be printed with single spacing. There will be five interval numbers per line. One blank will be between each interval number. Each interval number will occupy 25 columns providing 6 significant digits for each endpoint.

Example 2:

```
call intprm ("0",4,5,27,x,6,4)
```

The interval numbers in the first 6 rows and 4 columns of the interval matrix x will be printed with double spacing. There will be four interval numbers per line. Five blanks will be between each interval number. Each interval number will occupy 27 columns providing 7 significant digits for each endpoint.

4.0 56-Decimal Digit Interval Fortran Work

A 56 decimal digit version of the original Multics interval package has been implemented on the Multics system. This version uses the decimal arithmetic hardware available on the Multics system. The Multics decimal arithmetic unit performs both fixed and floating point 59 decimal digit arithmetic. Fixed decimal arithmetic was used to implement the decimal interval package. The floating point decimal arithmetic was not used because of the lack of control the user has over both the rounding strategy used and the detection of faults (overflow, underflow and divide by zero). The end points of the intervals are represented by a 56 decimal digit fraction and a 17 binary digit exponent. A 59 decimal digit fraction was not used was because in the implementation of the "best possible arithmetic" routines, two digits were needed for guard digits and one digit was reserved for overflow.

The implementation of the 56 decimal digit interval package followed the implementation of the original Multics interval package as closely as possible [10]. In this way the logic of the original interval package was used and the number of errors encountered in the implementation of the 56 decimal digit interval package could be reduced. The entire 56 decimal digit interval package was written in PL/I as Fortran does not support decimal arithmetic. Just the number of words required to carry the PL/I representation of the interval was declared in Fortran. The Fortran routines would carry the interval to be passed to the PL/I routines.

The first step in the implementation of the 56 decimal digit interval package was the implementation of the "Best Possible Arithmetic" or "BPA" routines as proposed by Yohe [2]. The already existing procedures for doing BPA for the original interval package were modified to perform 56 decimal "BPA". The implementation was fairly straightforward.

In the single precision interval package supplied by MRC [1] the implementation of the I/O package proved to be one of the most difficult operations. The reason was because of the conversion from floating decimal to floating binary and vice versa. The correct roundings had to be done for the conversions in either direction and the algorithms for the conversions got rather involved. The implementation of the 56 decimal digit interval I/O presented no problem with conversion since the internal representation of the interval was already in decimal. The only rounding is on output if the user requests less than 56 decimal digits on output.

In the initial interval effort [10] the interval counterparts of the basic external functions were implemented through the double precision floating binary routines in the Multics library. This obviously would not be sufficient for the 56 decimal

implementation. The basic external functions had to be calculated to a precision of greater than 56 decimal digits. To achieve this, we are using the Fortran Multiple Precision Package, MPP, developed by Richard Brent [9]. The basic external functions could be calculated using the MPP to an arbitrary number of decimal digits. One problem that was to be solved was the interface between Brent's routines written in Fortran and the 56 decimal digit interval package written in PL/I. The interface was just a conversion from the data representation in the MPP to the data representation used in the interval package after a correct rounding was made. Another problem was in the implementation of the SIN and COS routines. These routines required the greatest amount of work to implement. The argument had to be reduced to between 0 and 2pi. A case analysis then had to be made for each endpoint to determine the correct interval evaluation of the SIN or COS function. The case analysis depended on the correct 56 decimal digit bounds on the numbers pi/2, pi, 3pi/2, 2pi, 5pi/2, 3pi and 7pi/2. These constants had to be computed using the MPP.

5.0 Summary Tables

5.1 Gaussian Elimination

An example is shown below of a simple 4 by 4 linear system. The results for single precision, double precision, single precision interval, and extended interval are shown. For the interval results, the width of the intervals appear below each interval. The widths for the single precision intervals are expressed as a single precision value and the widths for the extended intervals are expressed as an extended interval. The widths of the single precision intervals are of the magnitude 10**-4 to 10**-2 while the widths of the extended intervals are of the magnitude 10**-51 to 10**-50. The reduced interval widths for the extended intervals is due to the extra precision of the extended intervals. The price that had to be paid for the increase in precision was an increase in central processing unit time from .44 central processing unit seconds to 12.64 central processing unit seconds for the extended interval results, but a good deal of precision was gained.

Matrix of Order 4:

```
 5   7   6   5
 7  10   8   7
 6   8  10   9
 5   7   9  10
```

Vector of Constants:

```
23 32 33 31
```

Single_Precision

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =      0.011512 seconds; Page faults =          1

THE SOLUTION IS AS FOLLOWS:

        0.999999449          1.000000328
        1.000000179          0.999999903
```

Double_Precision

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =      0.032315 seconds; Page faults =          7

THE SOLUTION IS AS FOLLOWS:

      1.00000000000000003        0.99999999999999998
      0.99999999999999999        1.00000000000000000
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =      0.444765 seconds; Page faults =          3

THE SOLUTION IS AS FOLLOWS:

   [ .99886015+00, .10011449+01]   [ .99929769+00, .10006993+01]
         0.1142337918e-02                 0.7008016109e-03

   [ .99991752+00, .10000813+01]   [ .99995276+00, .10000480+01]
         0.8184090257e-04                 0.4761666059e-04
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =     12.645348 seconds; Page faults =          5

THE SOLUTION IS AS FOLLOWS:

      [ .9999999999+00000, .1000000001+00001]
      [ .2646500000-00050, .2646500000-00050]

      [ .9999999999+00000, .1000000001+00001]
      [ .1623500000-00050, .1623500000-00050]

      [ .9999999999+00000, .1000000001+00001]
```

[.1895800000-00051, .1895800000-00051]

[.9999999999+00000, .1000000001+00001]
[.1101000000-00051, .1101000000-00051]

Another example, shown below, of a 7 by 7 linear system shows that the single precision interval version has broken down, but the extended interval version was able to compute the results. The interval widths were of the magnitude 10**-45 to 10**-43.

Matrix of Order 7:

180180	120120	90090	72072	60060	51480	45045
120120	90090	72072	60060	51480	45045	40040
90090	72072	60060	51480	45045	40040	36036
72072	60060	51480	45045	40040	36036	32760
60060	51480	45045	40040	36036	32760	30030
51480	45045	40040	36036	32760	30030	27720
45045	40040	36036	32760	30030	27720	25740

Vector of Constants:

1 1 1 1 1 1 1

Single_Precision

TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time = 0.019547 seconds; Page faults = 0

THE SOLUTION IS AS FOLLOWS:

0.000106439	-0.003075291	0.026859137
-0.101941098	0.188414240	-0.166749334
0.056536387		

Double_Precision

TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time = 0.023362 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

0.00015540015540038	-0.00419580419580964
0.03496503496507612	-0.12820512820526593
0.23076923076945858	-0.20000000000018252
0.0666666666672321	

Single_Precision_Interval

```
THE GAUSS ELIMINATION PROCESS HAS BROKEN DOWN
BECAUSE NO PIVOT GREATER THAN THE INPUT TOLERANCE
COULD BE FOUND FOR THE  6TH STEP.
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR THE GAUSSIAN ELIMINATION ARE AS FOLLOWS:

CPU time =     25.043362 seconds; Page faults =          11

THE SOLUTION IS AS FOLLOWS:

      [ .1554001554-00003, .1554001555-00003]
      [ .3468206177-00043, .3468206178-00043]

      [-.4195804196-00002,-.4195804195-00002]
      [ .3357609101-00043, .3357609102-00043]

      [ .3496503496-00001, .3496503497-00001]
      [ .1802686229-00043, .1802686230-00043]

      [-.1282051283+00000,-.1282051282+00000]
      [ .6297511335-00044, .6297511336-00044]

      [ .2307692307+00000, .2307692308+00000]
      [ .1835758441-00044, .1835758442-00044]

      [-.2000000001+00000,-.1999999999+00000]
      [ .4241149309-00045, .4241149309-00045]

      [ .6666666666-00001, .6666666667-00001]
      [ .1298809545-00045, .1298809546-00045]
```

5.2 BANSOL Program

The following example shows the results for the BANSOL routine which solves a banded system of equations using Gaussian elimination with no pivoting. The matrix of coefficients is assumed to be symmetrical and only the upper triangular banded matrix of coefficients is stored. The example uses an Hilbert matrix of order 4 as the matrix of coefficients. The single precision interval results have an interval width of the magnitude 10**-1 while the extended interval results have an interval width of the magnitude 10**-48.

```
Hilbert Matrix of Order 4

Vector of Constants:

                1  0  0  0
```

Single_Precision

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time =      0.029758 seconds; Page faults =          1

THE SOLUTION IS AS FOLLOWS:

      0.1600010896e+02      -0.1200011921e+03
      0.2400028324e+03      -0.1400018272e+03
```

Double_Precision

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time =      0.026642 seconds; Page faults =          1

THE SOLUTION IS AS FOLLOWS:

   0.160000000000000562d+02      -0.120000000000000581d+03
   0.240000000000001346d+03      -0.140000000000000855d+03
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time =      0.484731 seconds; Page faults =          3

THE SOLUTION IS AS FOLLOWS:

 [ .1596911+02, .1603087+02]    [-.1200379+03,-.1199619+03]
       0.3088021278e-01                0.3800058365e-01

 [ .2399760+03, .2400238+03]    [-.1400156+03,-.1399843+03]
       0.2390098572e-01                0.1564788818e-01
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time =      1.959542 seconds; Page faults =          4

THE SOLUTION IS AS FOLLOWS:

      [ .1599999999+00002, .1600000001+00002]
      [ .2227200000-00048, .2227200000-00048]

      [-.1200000001+00003,-.1199999999+00003]
      [ .2740700000-00048, .2740700000-00048]
```

```
[ .2399999999+00003, .2400000001+00003]
[ .1725100000-00048, .1725100000-00048]

[-.1400000001+00003,-.1399999999+00003]
[ .1127100000-00048, .1127100000-00048]
```

The next example uses an Hilbert matrix of order 10 for the matrix of coefficients. The solution for the single precision interval case cannot be found. the solution for the extended interval case could be found with the interval widths ranging from 10**-29 to 10**-25. Also note that the single precision results are meaningless.

Hilbert Matrix of Order 10

Vector of Constants:

1 0 0 0 0 0 0 0 0 0

Single_Precision

TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time = 0.065116 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

```
  0.6804037952e+02     -0.2041981995e+04      0.1802587769e+05
 -0.6609998340e+05      0.1038329834e+06     -0.3899770605e+05
 -0.5230199170e+05      0.9572640381e+04      0.6173250244e+05
 -0.3380692432e+05
```

Double_Precision

TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time = 0.099313 seconds; Page faults = 1

THE SOLUTION IS AS FOLLOWS:

```
  0.1000000023578835879 3d+03     -0.4950000202446120795 2d+04
  0.7920000429216765519 4d+05     -0.6006000388819666227 4d+06
  0.2522520134935564124 3d+07     -0.6306300507208783912 6d+07
  0.9609600830559078902 3d+07     -0.8751600801311593106 2d+07
  0.4375800420030998475 0d+07     -0.9237800922669556988 5d+06
```

Single_Precision_Interval

SOLUTION CANNOT BE FOUND

Extended_Interval

```
TIME AND PAGE FAULTS FOR BANSOL METHOD ARE AS FOLLOWS:

CPU time =     13.190437 seconds; Page faults =          16

THE SOLUTION IS AS FOLLOWS:

      [ .9999999999+00002, .1000000001+00003]
      [ .2607111740-00025, .2607111741-00025]

      [-.4950000001+00004,-.4949999999+00004]
      [ .3216633306-00025, .3216633307-00025]

      [ .7919999999+00005, .7920000001+00005]
      [ .2078321976-00025, .2078321977-00025]

      [-.6006000001+00006,-.6005999999+00006]
      [ .9134022350-00026, .9134022351-00026]

      [ .2522519999+00007, .2522520001+00007]
      [ .3044631515-00026, .3044631516-00026]

      [-.6306300001+00007,-.6306299999+00007]
      [ .8177182762-00027, .8177182763-00027]

      [ .9609599999+00007, .9609600001+00007]
      [ .1839461895-00027, .1839461896-00027]

      [-.8751600001+00007,-.8751599999+00007]
      [ .3552142351-00028, .3552142352-00028]

      [ .4375799999+00007, .4375800001+00007]
      [ .5810584606-00029, .5810584607-00029]

      [-.9237800001+00006,-.9237799999+00006]
      [ .1288532381-00029, .1288532382-00029]
```

5.3 SESOL Program

The SESOL program solves a banded system of linear equations using the LU decomposition technique. Operations with zero elements are not performed. The matrix of coefficients is symmetrical and only the upper triangular banded matrix of coefficients is stored. The first example uses an Hilbert matrix of order 4 for the matrix of coefficients. The single precision interval results had interval widths of the magnitude 10**-1, while the extended interval results had interval widths of the magnitude 10**-48.

Hilbert Matrix of Order 4

```
Vector of Constants:

                    1 0 0 0

                        Single_Precision

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time =      0.223950 seconds; Page faults =          5

THE SOLUTION IS AS FOLLOWS:

      0.1600008774e+02      -0.1200009499e+03
      0.2400022411e+03      -0.1400014400e+03

                        Double_Precision

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time =      0.225749 seconds; Page faults =          6

THE SOLUTION IS AS FOLLOWS:

   0.15999999999999999542d+02      -0.11999999999999999407d+03
   0.23999999999999998501d+03      -0.13999999999999998998d+03

                    Single_Precision_Interval

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time =      0.657901 seconds; Page faults =          9

THE SOLUTION IS AS FOLLOWS:

   [ .15966054+02, .16033943+02]    [-.12004178+03,-.11995823+03]
         0.3394412994e-01                  0.4177045822e-01

   [ .23997377+03, .24002631+03]    [-.14001725+03,-.13998282+03]
         0.2626705170e-01                  0.1720905304e-01

                        Extended_Interval

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time =      2.108744 seconds; Page faults =         49

THE SOLUTION IS AS FOLLOWS:

      [ .1599999999+00002, .1600000001+00002]
      [ .2479670000-00048, .2479670000-00048]

      [-.1200000001+00003,-.1199999999+00003]
```

[.3051400000-00048, .3051400000-00048]

[.2399999999+00003, .2400000001+00003]
[.1920100000-00048, .1920100000-00048]

[-.1400000001+00003,-.1399999999+00003]
[.1255600000-00048, .1255600000-00048]

The next example use an Hilbert matrix of order 10 for the matrix of coefficients. The single precision and single precision interval cases could not find a solution. The extended interval results had interval widths of the magnitude 10**-29 to 10**-25.

Hilbert Matrix of Order 10

Vector of Constants:

1 0 0 0 0 0 0 0 0 0

Single_Precision

STOP *** ZERO DIAGONAL ENCOUNTERED DURING EQUATION SOLUTION
EQUATION NUMBER = 9

Double_Precision

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time = 0.243050 seconds; Page faults = 6

THE SOLUTION IS AS FOLLOWS:

0.1000000007188466035 8d+03	-0.4950000062517137749 3d+04
0.7920000133783211504 1d+05	-0.6006000122056285881 6d+06
0.2522520058382703342 4d+07	-0.6306300160861385393 2d+07
0.9609600264425674991 8d+07	-0.8751600255945903822 1d+07
0.4375800134553289543 9d+07	-0.9237800296253089276 2d+06

Single_Precision_Interval

STOP *** ZERO DIAGONAL ENCOUNTERED DURING EQUATION SOLUTION
EQUATION NUMBER = 6

Extended_Interval

TIME AND PAGE FAULTS FOR SESOL METHOD ARE AS FOLLOWS:

CPU time = 12.364521 seconds; Page faults = 93

THE SOLUTION IS AS FOLLOWS:

[.9999999999+00002, .1000000001+00003]

```
[ .2961612465-00025, .2961612466-00025]

[-.4950000001+00004,-.4949999999+00004]
[ .3654013424-00025, .3654013425-00025]

[ .7919999999+00005, .7920000001+00005]
[ .2360920776-00025, .2360920777-00025]

[-.6006000001+00006,-.6005999999+00006]
[ .1037601650-00025, .1037601651-00025]

[ .2522519999+00007, .2522520001+00007]
[ .3458623775-00026, .3458623776-00026]

[-.6306300001+00007,-.6306299999+00007]
[ .9289071131-00027, .9289071132-00027]

[ .9609599999+00007, .9609600001+00007]
[ .2089581813-00027, .2089581814-00027]

[-.8751600001+00007,-.8751599999+00007]
[ .4035143144-00028, .4035143145-00028]

[ .4375799999+00007, .4375800001+00007]
[ .6600675959-00029, .6600675960-00029]

[-.9237800001+00006,-.9237799999+00006]
[ .1463739935-00029, .1463739936-00029]
```

5.4 SPLINE

The spline program solves a system of linear equations using an iterative technique to calculate the moments at a set of data points in order to fit a cubic spline to those data points. The first example uses 4 (X,Y) data points. The single precision interval widths were of the magnitude 10**-5. The extended interval widths were of the magnitude 10**-53.

(X,Y) DATA POINTS:

X	Y
1.6	1
5.4	2
7	1
8.2	1

Single Precision

TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time = 0.003969 seconds; Page faults = 1

```
INTERPOLATED VALUES

                X                              Y

        0.1000000000e+01          0.6069527492e+00
        0.3000000000e+01          0.1842634425e+01
        0.5000000000e+01          0.2160504758e+01
        0.7000000000e+01          0.1000000000e+01
        0.9000000000e+01          0.1135431752e+01
```

Double_Precision

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      0.007115 seconds; Page faults =          1

INTERPOLATED VALUES

                X                                      Y

  0.1000000000000000000d+01      0.6069527477460501562 5d+00
  0.3000000000000000000d+01      0.1842634435511974666 7d+01
  0.5000000000000000000d+01      0.2160504781961309110 0d+01
  0.7000000000000000000d+01      0.1000000000000000000 0d+01
  0.9000000000000000000d+01      0.1135431773019077610 9d+01
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      0.800636 seconds; Page faults =          4

INTERPOLATED VALUES

X = [ .1000000000000+01, .1000000000000+01]
                    0.0000000000e+00
Y = [ .6069516241550+00, .6069538742304+00]
                    0.1125037670e-05

X = [ .3000000000000+01, .3000000000000+01]
                    0.0000000000e+00
Y = [ .1842632219195+01, .1842636644841+01]
                    0.2212822437e-05

X = [ .5000000000000+01, .5000000000000+01]
                    0.0000000000e+00
Y = [ .2160503506660+01, .2160506069661+01]
                    0.1281499863e-05

X = [ .7000000000000+01, .7000000000000+01]
                    0.0000000000e+00
Y = [ .1000000000000+01, .1000000000000+01]
```

```
                    0.0000000000e+00

X = [ .9000000000000+01, .9000000000000+01]
                    0.0000000000e+00
Y = [ .1135428726673+01, .1135434836150+01]
                    0.3054738045e-05
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      5.429209 seconds; Page faults =          0

INTERPOLATED VALUES

X = [ .1000000000+00001, .1000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .6069527477+00000, .6069527478+00000]
    [ .1750000000-00053, .1750000000-00053]

X = [ .3000000000+00001, .3000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1842634435+00001, .1842634436+00001]
    [ .3700000000-00053, .3700000000-00053]

X = [ .5000000000+00001, .5000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .2160504781+00001, .2160504782+00001]
    [ .2200000000-00053, .2200000000-00053]

X = [ .7000000000+00001, .7000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1000000000+00001, .1000000000+00001]
    [ .0000000000+00000, .0000000000+00000]

X = [ .9000000000+00001, .9000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1135431773+00001, .1135431774+00001]
    [ .3400000000-00053, .3400000000-00053]

The next example uses 11 (X,Y) data points.  The single precision
interval  version of the program could not find a solution, while
the extended interval version could find a solution with interval
widths of the magnitude 10**-51 to 10**-50.

(X,Y) DATA POINTS:

 X          Y

1.0        1.008
10         20.183
19         39.096
28         58.69
37         85.48
```

```
46          106.7
55          132.91
64          156.9
73          180.88
82          207.21
91          231
```

Single_Precision

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      0.009885 seconds; Page faults =          1

INTERPOLATED VALUES

                X                           Y

        0.1000000000e+01          0.1008000001e+01
        0.5000000000e+01          0.9510347366e+01
        0.1000000000e+02          0.2018300009e+02
        0.1500000000e+02          0.3085654688e+02
        0.2000000000e+02          0.4108214760e+02
        0.2500000000e+02          0.5135257101e+02
```

Double_Precision

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:

CPU time =      0.032777 seconds; Page faults =          2

INTERPOLATED VALUES

                X                                   Y

 0.10000000000000000000d+01      0.10079999999999999999d+01
 0.50000000000000000000d+01      0.95103474691975531663d+01
 0.10000000000000000000d+02      0.20182999999999999999d+02
 0.15000000000000000000d+02      0.30856546767257317009d+02
 0.20000000000000000000d+02      0.41082147846906934087d+02
 0.25000000000000000000d+02      0.51352571259161417030d+02
```

Single_Precision_Interval

```
SINGLE PRECISION INTERVAL HAS BROKEN DOWN DUE TO BOUNDS FAULTS
(See attachment)
```

Extended_Interval

```
TIME AND PAGE FAULTS FOR SPLINE ARE AS FOLLOWS:
```

```
CPU time =     31.215525 seconds; Page faults =          5

INTERPOLATED VALUES


X = [ .1000000000+00001, .1000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .1008000000+00001, .1008000000+00001]
    [ .0000000000+00000, .0000000000+00000]

X = [ .5000000000+00001, .5000000000+00001]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .9510347469+00001, .9510347470+00001]
    [ .1518000000-00051, .1518000000-00051]

X = [ .1000000000+00002, .1000000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .2018300000+00002, .2018300000+00002]
    [ .0000000000+00000, .0000000000+00000]

X = [ .1500000000+00002, .1500000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .3085654676+00002, .3085654677+00002]
    [ .6490000000-00051, .6490000000-00051]

X = [ .2000000000+00002, .2000000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .4108214784+00002, .4108214785+00002]
    [ .3790000000-00051, .3790000000-00051]

X = [ .2500000000+00002, .2500000000+00002]
    [ .0000000000+00000, .0000000000+00000]
Y = [ .5135257125+00002, .5135257126+00002]
    [ .1165000000-00050, .1165000000-00050]
```

5.5 Matrix Inversion

The matrix inversion program finds the inverse of a square matrix. The first example finds the inverse of an Hilbert matrix of order 4. The single precision interval widths were of the magnitude from 10**-3 to 10**0. The extended interval widths were of the magnitude from 10**-50 to 10**-47.

Single Precision

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:


CPU time =      0.006392 seconds; Page faults =          0


INVERSE OF HILBERT MATRIX OF ORDER  4
```

```
ROW     1

  0.16000119e+02  -.12000130e+03  0.24000305e+03  -.14000196e+03

ROW     2

  -.12000130e+03  0.12000141e+04  -.27000332e+04  0.16800212e+04

ROW     3

  0.24000308e+03  -.27000333e+04  0.64800785e+04  -.42000503e+04

ROW     4

  -.14000198e+03  0.16800214e+04  -.42000505e+04  0.28000323e+04
```

Double_Precision

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =      0.010209 seconds; Page faults =           2

INVERSE OF HILBERT MATRIX OF ORDER 4


ROW     1

  0.160000000000000020d+02   -.120000000000000021d+03
  0.240000000000000049d+03   -.140000000000000031d+03

ROW     2

  -.120000000000000021d+03   0.120000000000000023d+04
  -.270000000000000053d+04   0.168000000000000034d+04

ROW     3

  0.240000000000000051d+03   -.270000000000000054d+04
  0.648000000000000127d+04   -.420000000000000081d+04

ROW     4

  -.140000000000000033d+03   0.168000000000000035d+04
  -.420000000000000082d+04   0.280000000000000052d+04
```

Single_Precision_Interval

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:
```

```
CPU time =      0.363970 seconds; Page faults =          2

INVERSE OF HILBERT MATRIX OF ORDER 4


ROW  1

 [ .15999083+02, .16000938+02]   [-.12001072+03,-.11998957+03]
        0.9272098541e-03                 0.1057004929e-01

 [ .23997471+03, .24002608+03]   [-.14001772+03,-.13998284+03]
        0.2568149567e-01                 0.1743507385e-01

ROW  2

 [-.12001029+03,-.11998998+03]   [ .11998861+04, .12001176+04]
        0.1014995575e-01                 0.1156997681e+00

 [-.27002862+04,-.26997238+04]   [ .16798125+04, .16801943+04]
        0.2811279297e+00                 0.1908416748e+00

ROW  3

 [ .23997635+03, .24002433+03]   [-.27002782+04,-.26997313+04]
        0.2398204803e-01                 0.2733917236e+00

 [ .64793482+04, .64806767+04]   [-.42004596+04,-.41995576+04]
        0.6642456055e+00                 0.4509582520e+00

ROW  4

 [-.14001562+03,-.13998484+03]   [ .16798277+04, .16801786+04]
        0.1538562775e-01                 0.1753845215e+00

 [-.42004345+04,-.41995821+04]   [ .27997164+04, .28002950+04]
        0.4261474609e+00                 0.2892761230e+00
```

Extended_Precision_Interval

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =      3.018572 seconds; Page faults =          3

INVERSE OF HILBERT MATRIX OF ORDER 4
```

```
ROW  1

   [ .1599999999+00002, .1600000001+00002]
   [ .4854000000-00050, .4854000000-00050]

   [-.1200000001+00003,-.1199999999+00003]
   [ .5527000000-00049, .5527000000-00049]

   [ .2399999999+00003, .2400000001+00003]
   [ .1341800000-00048, .1341800000-00048]

   [-.1400000001+00003,-.1399999999+00003]
   [ .8760000000-00049, .8760000000-00049]

ROW  2

   [-.1200000001+00003,-.1199999999+00003]
   [ .5258000000-00049, .5258000000-00049]

   [ .1199999999+00004, .1200000001+00004]
   [ .5990000000-00048, .5990000000-00048]

   [-.2700000001+00004,-.2699999999+00004]
   [ .1453800000-00047, .1453800000-00047]

   [ .1679999999+00004, .1680000001+00004]
   [ .9493000000-00048, .9493000000-00048]

ROW  3

   [ .2399999999+00003, .2400000001+00003]
   [ .1249800000-00048, .1249800000-00048]

   [-.2700000001+00004,-.2699999999+00004]
   [ .1423300000-00047, .1423300000-00047]

   [ .6479999999+00004, .6480000001+00004]
   [ .3455500000-00047, .3455500000-00047]

   [-.4200000001+00004,-.4199999999+00004]
   [ .2256400000-00047, .2256400000-00047]

ROW  4

   [-.1400000001+00003,-.1399999999+00003]
   [ .8089000000-00049, .8089000000-00049]

   [ .1679999999+00004, .1680000001+00004]
   [ .9213000000-00048, .9213000000-00048]

   [-.4200000001+00004,-.4199999999+00004]
   [ .2236900000-00047, .2236900000-00047]

   [ .2799999999+00004, .2800000001+00004]
```

[.1460700000-00047, .1460700000-00047]

The next example inverts an Hilbert matrix of order 10. The single precision interval version could not find a solution. To conserve space the extended interval results are not shown but may be found in the attachment to this letter. The extended interval widths ranged from 10**-36 to 10**-28. Also not that the single precision results are meaningless.

Single_Precision

TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time = 0.065202 seconds; Page faults = 1

INVERSE OF HILBERT MATRIX OF ORDER 10

ROW 1

0.66628764e+02 -.19763846e+04 0.17260560e+05 -.62370884e+05
0.95071291e+05 -.29811917e+05 -.52568417e+05 0.12496660e+04
0.68708063e+05 -.35645043e+05

ROW 2

-.19894290e+04 0.75388504e+05 -.72811225e+06 0.28178170e+07
-.46580271e+07 0.20127897e+07 0.23197084e+07 -.10075493e+07
-.21445800e+07 0.13152878e+07

ROW 3

0.17619120e+05 -.73787954e+06 0.76123086e+07 -.31378729e+08
0.56769210e+08 -.32712767e+08 -.24285263e+08 0.27813182e+08
0.54030478e+07 -.85081013e+07

ROW 4

-.65350269e+05 0.29248154e+07 -.32090695e+08 0.14215210e+09
-.28589484e+09 0.21212882e+09 0.10197568e+09 -.24367823e+09
0.10984572e+09 -.72728300e+07

ROW 5

0.10526792e+06 -.50690238e+07 0.60455295e+08 -.29635527e+09
0.68104005e+09 -.63099936e+09 -.21897011e+09 0.94184239e+09
-.71255193e+09 0.18048439e+09

ROW 6

-.42870195e+05 0.25873252e+07 -.38849799e+08 0.23780274e+09
-.67411450e+09 0.75598362e+09 0.35379525e+09 -.17238461e+10
0.15614960e+10 -.47486910e+09

```
ROW    7

  -.58890916e+05  0.25351663e+07  -.25165329e+08  0.94883239e+08
  -.16748904e+09  0.25520953e+09  -.65576277e+09  0.11986492e+10
  -.10311218e+10  0.32840154e+09

ROW    8

  0.35385499e+05  -.24513788e+07  0.41530602e+08  -.28713407e+09
  0.96178677e+09  -.15891991e+10  0.98973335e+09  0.49803286e+09
  -.97000460e+09  0.35773168e+09

ROW    9

  0.36014969e+05  -.71491391e+06  -.91393845e+07  0.16378166e+09
  -.77517486e+09  0.14881434e+10  -.79604687e+09  -.11365808e+10
  0.16805552e+10  -.61501253e+09

ROW   10

  -.25268021e+05  0.85312601e+06  -.36485157e+07  -.26486693e+08
  0.20763149e+09  -.46059757e+09  0.24735363e+09  0.43887222e+09
  -.64172422e+09  0.23783952e+09
```

Double Precision

```
TIME AND PAGE FAULTS FOR THE INVERSION ARE AS FOLLOWS:

CPU time =      0.104900 seconds; Page faults =          3


INVERSE OF HILBERT MATRIX OF ORDER 10

ROW    1

  0.10000000225334438 5d+03   -.49500001924608660 7d+04
  0.79200004064343256 9d+05   -.60060003670371123 0d+06
  0.25225201741354272 6d+07   -.63063004765925095 9d+07
  0.96096007790543368 7d+07   -.87516007504928902 2d+07
  0.43758003929292513 0d+07   -.92378008620549492 9d+06

ROW    2

  -.49500001923406703 2d+04   0.32670001642319687 6d+06
  -.58806003467440037 4d+07   0.47567523130774615 6d+08
  -.20810791485140684 1d+09   0.53513464064219981 6d+09
  -.83243166642866319 8d+09   0.77014086398799279 0d+09
  -.38988381349927746 1d+09   0.83140207349016914 5d+08

ROW    3

  0.79200004059794671 0d+05   -.58806003465721720 1d+07
  0.11290752731596679 2d+09   -.95135046604764127 4d+09
```

```
  0.42810771132759859 8d+10   -.11237827457233148 5d+11
  0.17758542201024441 1d+11   -.16635042629465951 6d+11
  0.85065559064410232 3d+10   -.18290845549707656 6d+10

ROW     4

  -.60060003664776851 0d+06   0.47567523127960147 4d+08
  -.95135046602099166 2d+09   0.82450373959680658 9d+10
  -.37875640626551995 6d+11   0.10100170853391650 9d+12
  -.16160273391926267 0d+12   0.15290796737354902 1d+12
  -.78843170772551986 1d+11   0.17071455797886197 1d+11

ROW     5

  0.25225201738114935 9d+07   -.20810791483306953 5d+09
  0.42810771130443794 5d+10   -.37875640625602423 4d+11
  0.17675298980035697 5d+12   -.47723307294362368 2d+12
  0.77128577511534554 9d+12   -.73586959210546783 1d+12
  0.38208613460639784 1d+12   -.83223346825902715 8d+11

ROW     6

  -.63063004755702328 4d+07   0.53513464058045500 2d+09
  -.11237827456355421 5d+11   0.10100170852911614 3d+12
  -.47723307293317961 0d+12   0.13015447446793373 3d+13
  -.21210358806694989 4d+13   0.20377927146205238 8d+13
  -.10643828020102334 3d+13   0.23302537068048908 1d+12

ROW     7

  0.96096007771925972 1d+07   -.83243166631149129 7d+09
  0.17758542199246992 4d+11   -.16160273390832188 7d+12
  0.77128577508359503 5d+12   -.21210358806293253 5d+13
  0.34806742645102340 3d+13   -.33639752722214532 2d+13
  0.17660870175121363 9d+13   -.38837561720107403 1d+12

ROW     8

  -.87516007485395712 0d+07   0.77014086336150143 6d+09
  -.16635042627466375 3d+11   0.15290796736041537 4d+12
  -.73586959206257442 2d+12   0.20377927145481384 4d+13
  -.33639752721663891 7d+13   0.32678616907994671 6d+13
  -.17232864375341711 2d+13   0.38044958369994492 2d+12

ROW     9

  0.43758003918330178 0d+07   -.38988381342678519 9d+09
  0.85065559052618844 9d+10   -.78843170764481638 3d+11
  0.38208613457827492 4d+12   -.10643828019568273 7d+13
  0.17660870174579553 0d+13   -.17232864375097479 0d+13
  0.91232811319661604 2d+12   -.20211384111426626 6d+12

ROW    10
```

-.923780085950684804d+06	0.831402073318937322d+08
-.182908455468634513d+10	0.170714557958832975d+11
-.832233468186303462d+11	0.233025370665754467d+12
-.388375617184253555d+12	0.380449583689833549d+12
-.202113841111777620d+12	0.449141868706854188d+11

Single_Precision_Interval

SINGLE PRECISION INTERVAL HAS BROKEN DOWN DUE TO DIVISION BY ZERO (See attachment)

Extended_Precision_Interval

The extended interval results are not shown here in order to conserve space. See the attachment for the results.

5.6 FFT Program

The FFT program supplied by WES, once interfaced and running correctly on Multics, was modified to print out the central processor time used and page faults generated during various stages of the overall program. These stages were initialization, the FFT subroutine, and output.

The FFT program was then converted to double precision. The only difficulty encountered during this stage of the work was that complex arithmetic is not supported in double precision in Multics fortran and, therefore, had to be simulated. Similarly, during the conversion to interval, complex arithmetic again had to be simulated. Further, in the driver routine, a call to ATAN2 with arguments 1.0 and 0.0 was replaced by the value such a call returns, one half pi. The value returned by ATAN2 was not the minimum interval representation and, when used in subsequent calls to the SIN function, resulted in unacceptably large intervals being returned. The only modification made to the interval version of the program before executing it utilizing the 56 decimal package was to insert a call to a subroutine, GENPI, to obtain a 56 decimal digit precision value for pi.

The single precision, double precision, interval, and 56 decimal interval versions of the program were executed using the original real coefficients supplied by WES and also using real coefficients in the form of a square wave as suggested by WES. Other real coefficients were also tried but the results added little additional information for analysis.

Using the real coefficients supplied by WES, all complex coefficients but one should have, theoretically, been zero. The following table has example values computed in each of the four different runs.

type of arithmetic	typical zero value computed
single precision	0.2491e-05
double precision	0.2102941d-05
standard interval	[-.391942e-05,.841782e-05]
56 decimal interval	[-.1803163-00052,.2545842-00052]

The non-zero coefficient computed should have theoretically been 0.0 - 0.5i. The following table gives the values actually computed.

type of arithmetic	value computed
single precision	-0.77297e-04 - 0.50000e+00i
double precision	0.19775601d-08 - 0.50000000d+00i
standard interval	[-.631673-05,.131955-04] + [-.500000+00,-.499999+00]i
56 decimal interval	[-.54200231-00052,.7330509-00053] + [-.50000000+00000,-.49999999+00000]i

The non-zero values computed in the above runs are in close agreement. This is true of the values computed using the "square wave" real coefficients also. The agreement of the values is demonstrated in the following sample.

type of arithmetic	value computed
single precision	0.62378e-01 + 0.15625e-01i
double precision	0.62378497d-01 + 0.15625000d-01i
standard interval	[.623784-01,.623786-01] + [.156249-01,.156251-01]i
56 decimal interval	[.6237849-00001,.6237850-00001] + [.1562500-00001,.1562501-00001]i

The number of page faults and the amount of CPU time required are dependent to some degree upon the load on the system. The following tables, however, still indicate a trend as to relative speed and overhead.

Page Faults (Original Input Data)

	initialization	fft subroutine	output	total
single precision	10	1	1	23
double precision	34	1	1	36
standard interval	23	2	14	39
56 decimal interval	2742	339	114	3195

CPU Time in Seconds (Original Input Data)

initialization	fft subroutine	output	total

single precision	0.0265	0.0428	0.2930	0.3623
double precision	0.3113	0.0460	0.3105	0.6678
standard interval	0.8222	2.5133	11.1639	14.4994
56 decimal interval	260.8131	180.6730	25.3920	466.8781

5.7 SLOPE Program Work

A large portion of time had been spent in an attempt to understand the applications programs. The logic was carefully followed using the given data. Throughout the program, the language of the program was updated (FORTRAN IV as opposed to FORTRAN II), detected inefficiencies removed (for example, unnecessary variables, GO TO's to GO TO's, unused labels, GO TO's to the next executable statement, etc.), and, in general, made more readable. This time proved beneficial not only in facilitating the conversion of the routines to interval arithmetic but also in that it exposed some errors in the program (double initialization of some variables, an integer function which should have been a real function, etc.) These errors were corrected and duly reported to WES. It should be noted that all testing was done using corrected version of the original program rather than the heavily modified version. It was felt that testing conditions should approximate the working conditions at WES as closely as possible.

Once the program was successfully interfaced with Multics and reproducing the desired output, a program of testing was outlined. This consisted of varying a set of three inputs (cohesion, unit weight and phi) for the program plus or minus ten percent, singly and in conjunction with each other. A comprehensive analysis was run consisting of 81 separate runs with 27 comparisons of 3, and were submitted to WES in July, 1977. The largest problem encountered at this stage was the production of summary reports which presented the analysis in a readable form. The report was finally configured to consist of 27 comparisons of three runs each in regard to central processing unit time, paging and fluctuation of the output data (Table 1).

TABLE 1

Configuration of the 27 comparisons of the test runs. For example, run one compares the three runs cohesion+10% phi+10% unit weight-10%, cohesion+10% phi+10% unit weight-00%, cohesion+10% phi+10% unit weight+10%.

cohesion +10% phi +10% unit weight [-10%, no flux, +10%]
cohesion +10% phi -00% unit weight [-10%, no flux, +10%]
cohesion +10% phi -10% unit weight [-10%, no flux, +10%]
cohesion -00% phi +10% unit weight [-10%, no flux, +10%]

cohesion -00% phi -00% unit weight [-10%, no flux, +10%]
cohesion -00% phi -10% unit weight [-10%, no flux, +10%]
cohesion -10% phi +10% unit weight [-10%, no flux, +10%]
cohesion -10% phi -00% unit weight [-10%, no flux, +10%]
cohesion -10% phi -10% unit weight [-10%, no flux, +10%]
phi +10% unit weight +10% cohesion [-10%, no flux, +10%]
phi +10% unit weight -00% cohesion [-10%, no flux, +10%]
phi +10% unit weight -10% cohesion [-10%, no flux, +10%]
phi -00% unit weight +10% cohesion [-10%, no flux, +10%]
phi -00% unit weight -00% cohesion [-10%, no flux, +10%]
phi -00% unit weight -10% cohesion [-10%, no flux, +10%]
phi -10% unit weight +10% cohesion [-10%, no flux, +10%]
phi -10% unit weight -00% cohesion [-10%, no flux, +10%]
phi -10% unit weight -10% cohesion [-10%, no flux, +10%]
unit weight +10% cohesion -00% phi [-10%, no flux, +10%]
unit weight +10% cohesion -10% phi [-10%, no flux, +10%]
unit weight -00% cohesion +10% phi [-10%, no flux, +10%]
unit weight -00% cohesion -00% phi [-10%, no flux, +10%]
unit weight -00% cohesion -10% phi [-10%, no flux, +10%]
unit weight -10% cohesion +10% phi [-10%, no flux, +10%]
unit weight -10% cohesion -00% phi [-10%, no flux, +10%]
unit weight -10% cohesion -10% phi [-10%, no flux, +10%]
unit weight +10% cohesion +10% phi [-10%, no flux, +10%]

At this time the applications programs were converted to double precision and the same testing procedure as outlined above was applied. No significant problems were encountered during the conversion. The output of the two tests were given to the same precision as that given in the report supplied by WES. The summary report along with each run and data was sent to WES for further evaluation.

During the testing it was noted that one of the inputs to be varied, cohesion of the first soil, was zero. As 0-10% = 0+10% = 0 no fluctuation was produced by this parameter. A non-zero value was received from WES and the testing procedure applied again. The output was increased to 8 digits of accuracy, the maximum for single precision FORTRAN on Multics, for this set of tests.

An analysis of the test output discerned no significant difference in accuracy between the single and double precision (Table 2).

TABLE 2

Sample values from the case phi-00%, unit weight-00%, cohesion[-10%, no flux, +10%] for single precision (SP), double precision (DP), single precision interval (SPI), and extended interval (EI). The values shown are for CP.

=10%

SP	-1.6056320	-1.3048941	439.65
DP	-1.60563196598609845	-1.30489416517190121	439.65
SPI	[-2.0212812,-1.2585650]	[-1.6226176,-1.0324112]	[435.14,444.21]
EI	[-1.6056320,-1.6056320]	[-1.3048741,-1.3048943]	[439.65,439.65]

no_flux

SP	-1.6069299	-1.3060394	439.63
DP	-1.60692993603933327	-1.30603947523241611	439.63
SPI	[-2.0217068,-1.2604411]	[-1.6231307,-1.0340142]	[435.13,444.18]
EI	[-1.6069299,-1.6069299]	[-1.3060194,-1.3060396]	[439.63,439.63]

±10%

SP	-1.6082275	-1.3071844	439.60
DP	-1.60822753256086878	-1.30718440917743123	439.60
SPI	[-2.0220372,-1.2623842]	[-1.6235718,-1.0356705]	[435.11,444.14]
EI	[-1.6082275,-1.6082275]	[-1.3071643,-1.3071845]	[439.60,439.60]

Better than 50% of the output agreed to the full eight digits, the rest differed by no more that ±3 in the eighth digit. This was attributed to the fact that the Honeywell 68/80 does floating point computations in double precision.

Concurrent with the above testing was the transformation of the applications programs into interval arithmetic. Several problems occurred during this period which greatly hampered progress. The first problem was that, apparently, the given interval routines to read in data were designed to read from one file without interruption, i.e. file 5 in FORTRAN. The applications programs on the other hand used a scratch file writing and then reading from it. The second problem, which interacted with the first to create a much larger problem, was that a bug in the Multics FORTRAN I/O was struck upon. Normally when a record of, say, 256 characters is read from a file containing 80 characters, the remaining 176 characters are padded with blanks. In this case they were not; an error message, "record too short", was produced instead. These problems were solved and work begun on the interval version of the testing.

Testing procedures for the interval version were the same as those for single and double precision. During the testing of the interval version two problems worth noting were encountered. The first problem concerned the manner in which the algorithm was coded. The second problem involved the identification of data sensitivity. This was much more difficult to handle, taking some

effort even to determine the nature of the source. Once the nature of the problem was discovered it took an even larger portion of time to track down the source of the problem owing to the near impossibility of following the logic of the programs. The first problem was a result of the way in which the interval package evaluates the test value in an arithmetic IF statement. When an arithmetic IF is encountered with an interval test value the interval is converted to a real (i.e. the midpoint is taken) and the branch evaluated as normal. The difficulty was occurring at a particular branch in the subroutine WGHT which was to be taken only if the test value was positive. Certain intervals were passing along this branch whose midpoint was indeed positive but whose left endpoint was negative. The interval was subsequently used as a divisor; as it contained zero a zero-divide error was flagged by the interval arithmetic routines. The problem was solved simply by recoding the branch as a logical IF statement which is evaluated in a different manner and avoids this problem.

The second problem, as stated before, was much more difficult to handle. During the testing it was noted that some of the runs contained intervals which were "blowing up", that is, the width of the intervals were becoming quite large. After a perusal of the output a correlation was discovered between the blownup intervals and the varying of unit weight by -10%: all runs which varied unit weight by -10% contained blownup intervals! Desiring to know with some certainty at what point the intervals would start blowing up the following strategy was devised: each run would start out with unit weight varied by zero; unit weight would then be decremented by units of .25% of the initial value until the intervals would blowup (Table 3). Once this strategy was carried out a value (2.25%) was found at which unit weight could be decreased without generating large interval widths. The testing procedure was redone using this value to decrease unit weight. In the summary report generated for this set of testing a note is made indicating those runs in which the unit weight is decreased by this value rather than the normal 10%.

TABLE 3

Sample values from one of the graduated runs (cohesion-00%, phi-10%). T3 = FS1 - FSL. For this run the intervals became unstable at unit weight-3.25%. (* indicates infinity)

```
UNIT WEIGHT - 2.50%
   T3 = [-.382388e-01,-.380500e-01]
   CP = [-1.786,-1.094]  [-1.43,-.897]  [435,444]

UNIT WEIGHT - 2.75%
   T3 = [-.38430e-01,-382241e-01]
   CP = [-1.786,-1.093]  [-1.43,-.895]  [435,444]
```

```
UNIT WEIGHT - 3.00% (bounds faults occurred)
   T3 = [-.139087e-03,0.826716e-04]
   CP = [-1.977,-.9748]  [-1.57,-.801]  [434,446]

UNIT WEIGHT -3.25% (bounds faults occurred)
   T3 = [-.240356e-03,-.183582]
   CP = [-*,+*]  [-*,+*]  [34,5731]
```

While these runs were being made a large amount of time and effort was spent in the tracking down of the source of the expanding intervals. After an extended effort the source was traced with a large degree of confidence to one statement, "T3 = FS1 - FSL". It seems that as unit weight is decreased the difference between FS1 and FSL becomes increasingly small. After a time the subtraction has the effect of stripping off the significant digits of accuracy of the resultant interval. The problem was compounded by using T3 as a divisor in a subsequent computation thus exploding the interval during the next few computations.

One other unexpected benefit was reaped by during this procedure. While tracing through the routines it was noted that in the subroutine WGHT several computations could be combined and an interval consistently of less than optimal width could be factored out producing a more accurate algorithm. While its disturbing influence could not altogether be avoided it was minimized.

The 56 decimal digit version of the interval package was then made available. Testing was done as before. Three outstanding features of the testing were noted. The first was that the data dependency of the algorithm as noted above disappeared, the interval widths getting no larger than 10**-4. The second was the overly large amount of central processor unit time required up processing each run (55 minutes plus or minus 6 minutes) (Table 4). It was noted, however, that most of this time was spent in just a few of the interval routines, principally the trigometric functions which take considerable amounts of time to evaluate when the argument is greater than one. The fast fourier transform routines did not encounter this large an increase in the amount of central processing unit time per run.

TABLE 4

This table shows the maximum and minimum central processor unit times which were encountered during each set of testing runs for single precision, double precision, single precision interval, extended interval.

SINGLE PRECISION
max -- 2.80 (cohesion+10%, phi-10%, unit weight-10%)
min -- 2.48 (cohesion-00%, phi-10%, unit weight-00%)

DOUBLE PRECISION
max -- 3.08 (cohesion+10%, phi-10%, unit weight-00%)
min -- 2.58 (cohesion-00%, phi+10%, unit weight-00%)

SINGLE PRECISION INTERVAL
max -- 26.30 (cohesion+10%, phi-10%, unit weight+10%)
min -- 22.35 (cohesion-10%, phi-10%, unit weight-10%)

EXTENDED INTERVAL
max -- 3719 (cohesion-00%, phi-00%, unit weight+10%)
min -- 2951 (cohesion-00%, phi-10%, unit weight-00%)

The third and most pleasant of the notable features was the complete lack of problems in the bringing up and testing of the 56 decimal digit version of the algorithm. This was a benefit of the absence of any needed large modifications to be made to the single precision interval to convert it to the 56 decimal digit version. The only required modification was that of a slight adjustment to the output parameters to widen the output field. This was required as the exponent field supplied by the 56 decimal digit routines was somewhat larger.

5.8 STRESS

The STRESS program finds the stress on a plane at particular nodal points. This program was provided to us as an extra program from WES to analyze. The program was run in single precision and double precision. The program terminated abnormally in both cases. The output produced by WES also indicated that the program terminated abnormally. The output produced at USL matched the output provided by WES up to the point of termination. The program was also run in single precision interval and extended interval and also terminated abnormally. We are currently waiting to receive from WES corrections to the STRESS program and additional input to run the program again.

6.0 Conclusions and Recommendations

We have concluded that the use of single precision and 56 decimal digit extended precision interval arithmetic can, at times, be extremely useful. It can be used to show the limits of precision of an algorithm. From the testing it was shown that when using the 56 decimal digit data type much better bounds were obtained

for the results than when using the single precision interval data type. This was expected for two reasons: 1) 56 decimal digits carry more precision than 27 binary digits (approximately equivalent to 8 decimal digits) used for single precision and 2) there was no conversion error on input and output. The price paid for this increase in precision is a decrease in runtime efficiency. The testing indicated that single precision interval arithmetic resulted in, at most, one order of magnitude increase in execution time over that required for regular single or double precision execution. 56 decimal digit interval operation resulted in a further increase of more than one to more than two orders of magnitude.

One application for 56 decimal interval arithmetic would be to validate existing routines. Any data sensitivity discovered could be included in a description of the algorithm and directions on its use. Although 56 decimal interval arithmetic is expensive, its cost must be balanced against possible consequences of using invalid results. A defective dam or the moving of 100,000 tons of dirt unnecessarily would cost considerably more than a few hours of computer time.

A more cost effective technique might be to first test the algorithm using single precision interval arithmetic. Its relatively small decrease in runtime efficiency indicates that its use is more than justified as an economical means in identifying possible trouble areas in an algorithm for the data under consideration. The more expensive 56 decimal interval arithmetic could then be used to investigate those cases with possible problem areas.

Interval arithmetic can also be used to determine the precision of the arithmetic required to guarantee a given precision in the results of an algorithm. In some of the benchmarks executed in 56 decimal digit interval arithmetic, the results were good only to 40 or so digits. This represents a considerable loss of precision. It also points out why arbitrarily picking a given precision for arithmetic does not guarantee results in which absolute confidence can be placed. How great an increase in precision is obtained, if any, by going from a machine with 32 bit words to one with 60 bit words?

In general, whether using interval or regular arithmetic, the greater the precision the longer the run time required for a given algorithm. Having variable precision interval arithmetic would allow the validation of algorithms for which single precision interval arithmetic is insufficient without having to go all the way to 56 decimal digit precision. There will also be instances where it might be desirable or necessary to go beyond 56 decimal digits of precision. In any case, the overhead associated with execution in interval arithmetic will only be as great as required for the necessary precision.

The execution speed of interval arithmetic can be increased in several ways. One would be to decrease the number of levels of interpretation required in the current implementation. The optimum solution would be to have a hardware or firmware module which could execute variable precision interval arithmetic. (Many existing minicomputer systems have undefined opcodes and ports for just such requirements). As a side effect, an arithmetic unit that can execute variable precision interval arithmetic can also execute variable precision regular arithmetic. This means that interval arithmetic, of the necessary precision, could be used to determine the required arithmetic precision required for the results of the algorithm. Then, the algorithm could be executed using only the required precision on the special arithmetic module.

REFERENCES

[1] Ladner, T. D. and Yohe, J. M., "An interval arithmetic package for the UNIVAC 1108," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1055, May 1970.

[2] Yohe, J. M., "Best possible floating point arithmetic," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1054, Mar 1970.

[3] Binstock, W., Hawkes, J., and Hsu, N. T., "An interval input/output package for the UNIVAC 1108," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1212, Sep 1973.

[4] Crary, F. D., "The AUGMENT precompiler; I: User information," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1469, Dec 1974.

[5] Crary, F. D., "The AUGMENT precompiler; II: Technical documentation," The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report No. 1470, Dec 1975.

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

[7] Abramowitz, M. and Stegun, I. A., ed., Handbook of Mathematical Functions, National Bureau of Standards Applied Mathematics Series, Jun 1964.

[8] Reuter, E. K., "MULTICS interval package listings," Computer Science Department Report No. 76-7-3, University of Southwestern Louisiana, Lafayette, La., Aug 1976.

[9] Brent, R. P., "A fortran multiple-precision arithmetic package," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., May 1976.

[10] Reuter, E. K., Podlaska-Lando, S., and Shriver, B. D., "The implementation of the Mathematics Research Center's interval arithmetic package on the MULTICS system," Computer Science Department Report No. 76-7-2.A, University of Southwestern Louisiana, Lafayette, La., Aug 1976.

The following are additional computer listings and runs available from the Automatic Data Processing Center, U. S. Army Engineer Waterways Experiment Station, P. O. Box 631, Vicksburg, Miss. 39180:

Source Listing for the MULTICS Interval Arithmetic Package, by Eric K. Reuter and S. Podlaska-Lando, Computer Science Department Report No. 76-7-3, University of Southwestern Louisiana.

Linear System Solvers, Binder 2-5, by Dr. Bruce D. Shriver, University of Southwestern Louisiana, Sep 1976.

Matrix Operations, Binder 3-5, by Dr. Bruce D. Shriver, University of Southwestern Louisiana, Sep 1976.

FFT, Binder 4-5, by Dr. Bruce D. Shriver, University of Southwestern Louisiana, Sep 1976.

Error Tests and Elementary Functions, Binder 5-5, by Dr. Bruce D. Shriver, University of Southwestern Louisiana, Sep 1976.

APPENDIX A: MATHEMATICAL BASIS FOR INTERVAL ARITHMETIC

The details of the mathematical basis for interval arithmetic are developed in Moore [6]. The set of interval numbers is the set of all closed intervals on the real number line. An interval may be represented by an ordered pair of real numbers [a,b] where $a \leq b$. If $a = b$, then the interval is said to be degenerate.

The operations of addition, subtraction, multiplication, and division between two intervals (except for the division of one interval by an interval containing zero) are defined as follows where \$ is one of the above operations:

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

Each of the operations of addition, subtraction, multiplication, and division may be defined as follows:

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

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

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

$$[a,b] / [c,d] = [\min\{a/c,a/d,b/c,b/d\}, \max\{a/c,a/d,b/c,b/d\}] \quad \text{if } 0 \notin [c,d]$$

In the cases of multiplication and division, by examining the signs of the endpoints of the intervals being multiplied or divided; a determination in advance can be made of which products or quotients will be the maximum and the minimum.

The following real single valued functions of intervals may be useful:

The midpoint of an interval, mid ([a,b]), is defined to be the real number (a+b)/2.

The length of an interval, length ([a,b]), is defined to be the real number b-a.

The supremum of an interval, sup ([a,b]), is the real number a.

The infimum of an interval, inf ([a,b]), is the real number b.

The distance from interval [a,b] to interval [c,d], dis ([a,b],[c,d]), is defined to be the real number $\max\{|c-a|, |d-b|\}$.

The following interval single valued functions of intervals may also be useful:

The union of intervals [a,b] and [c,d], union ([a,b],[c,d]), is defined to be the smallest interval containing both [a,b] and [c,d] and is given by [min{a,c},max{b,d}]. The intersection of intervals [a,b] and [c,d], intsct ([a,b],[c,d]), is defined to be the largest interval contained in each of [a,b] and [c,d] or is empty if [a,b] and [c,d] are disjoint intervals and is given by [max{a,c},min{c,d}].

The relational operations may be defined on intervals as follows:

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

The above definition means that two intervals are equal if and only if they both are degenerate and represent the same real number. This definition is employed instead of the more general definition of testing for a = c and b = d. The reason the more general definition is not used is because we will regard intervals as bounds on an exact but unknown real number. If two intervals were not degenerate and if both intervals had the same endpoints, then the intervals may not represent the same exact real number. The only way for the two intervals to represent the same exact real number is for both intervals to be degenerate with their endpoints equal to the real number. We also say that

$$[a,b] \neq [c,d] \text{ if and only if } [a,b] \text{ intersection } [c,d] = \emptyset$$

This definition means that two intervals are not equal if and only if they are disjoint intervals and cannot represent the same exact real number.

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

The above definition means that two intervals are ordered by the $\leq$ relational operator if and only if $\forall\, x \in [a,b]$ and $\forall\, y \in [c,d]$, $x \leq y$.

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

The above definition means that two intervals are ordered by the > relational operator if and only if $\forall\, x \in [a,b]$ and $\forall\, y \in [c,d]$, $x > y$.

Interval valued functions of interval variables are defined in terms of real valued functions of real variables. If f is a real valued function of a real variable, then f may be extended to an interval valued function, F, of an interval variable by defining

$$F([a,b]) = \{f(x) : x \in [a,b]\}$$

If f is defined and continuous on [a,b], then F([a,b]) will be an interval. If intervals are to represented as pairs of real numbers, then the above definition is not operational. Some

means is needed for deriving the endpoints of the image of [a,b] under the function F. The endpoints of the image interval will be the image under f of points of [a,b].

For functions, f, that are monotonic on the interval [a,b], the endpoints of the image of [a,b] under F can be expressed as the result of the function f evaluated at the endpoints of [a,b]. If f is monotonic increasing on [a,b], then F([a,b]) = [f(a), f(b)]. If f is monotonic decreasing on [a,b], then F([a,b]) = [f(b), f(a)]. If f is not monotonic over [a,b], then [a,b] can be divided into disjoint subintervals; X(i), i = 1,2,3,...n; where U X(i) = [a,b] and f is monotonic on each X(i). In this case F([a,b]) = U f(X(i)).

APPENDIX B: DESCRIPTION OF INTERVAL FAULTS AND LIST OF ERROR MESSAGES

The following table lists the possible fault conditions that can arise during an interval operation along with the value of the fault flag and the default action code that specifies the action taken by intrap after it is called. The action code is explained after the table.

Fault Flag	Fault Condition		Default Response
	Left Endpoint	Right Endpoint	
0	no faults	no faults	-
1	no faults	overflow	3
2	no faults	infinity	2
3	no faults	underflow	0
4	overflow	no faults	3
5	overflow	overflow	3
6	overflow	infinity	2
7	overflow	underflow	3
8	infinity	no faults	2
9	infinity	overflow	2
10	infinity	infinity	2
11	infinity	underflow	2
12	underflow	no faults	0
13	underflow	overflow	3
14	underflow	infinity	2
15	underflow	underflow	0
16	division by zero		2
17	zero to the zero power		1
18	square root of a negative number		2
19	log of a non-positive number		2
20	underflow during interval-to-real		0
21	overflow during interval-to-real		2
22	intersection of disjoint intervals		2
23	argument out of range		2
24	underflow during interval-to-double		2
25	underflow		2
26	overflow		2

The action codes are as follows:

0 - Exit
1 - Print error message and arguments
2 - Print error message, arguments and trace stack
3 - Print error message, arguments, trace stack and stop

The arguments that are displayed are the arguments of the calling program. Three arguments are always passed to intrap. If the calling program had only two arguments, then the first two arguments passed to intrap have the same value as the first argument of the calling routine. If the calling routine has only one argument, then all three arguments passed to intrap have the same value as that argument.

intrap_Modification

Under certain circumstances the user may wish to change the default action taken by intrap when a fault occurs. The user may also wish to change the value assigned as the result of an operation in order to be more mathematically consistent with the problem to be solved. The user can modify the action taken by intrap by including the following statements in the user's Fortran program.

```
common /intflt/ ifault,routin,type(3),itgarg(3),rarg(3),darg(3),
                itvagr(2,3),montor(32)
integer type
character*6 routin
real itvarg
double precision darg
```

"ifault" will contain the fault flag after each operation.

"routin" will contain a character string which is the name of the last routine to call intrap.

"type" will contain the types of the last three arguments passed to intrap. If type(i) is zero, then that particular argument was not present in the call to intrap. The type codes are as follows:

1 - integer
2 - real
3 - double precision
4 - interval

"itgarg", "rarg", "darg", and "itvarg" will contain the arguments passed to intrap. They contain respectively either the integer, real, double precision, or interval arguments. For example, if type(1) = 3, type(2) = 1, and type(3) = 4 then darg(1) will

contain the double precision argument, itgarg(2) will contain the integer argument, and itvarg(1,3) will contain the interval argument.

"montor" contains the action codes for each type of fault listed in the table. If the user wants to change the action for a particular fault, then the user changes the location in the montor array that corresponds to the particular fault. For example, if the user wishes to change the action for a divide by zero fault to a fatal error, then the following statement is included in the user's program.

```
montor(16) = 3
```

In this case when a zero divide occurs the program will stop.

List_of_Error_Messages

The following is a list of the error messages produced by intrap.

BOUNDS FAULT DURING "routine name" LEFT ENDPOINT-- "fault" RIGHT ENDPOINT-- "fault"

DIVISION BY ZERO DURING "routine name"

ZERO TO THE ZERO POWER DURING "routine name"

SQUARE ROOT OF A NEGATIVE NUMBER DURING "routine name"

LOG OF A NON-POSITIVE NUMBER DURING "routine name"

UNDERFLOW DURING CONVERSION FROM INTERVAL TO REAL IN "routine name"

OVERFLOW DURING CONVERSION FROM INTERVAL TO INTEGER IN "routine name"

INTERSECTION OF DISJOINT INTERVALS DURING "routine name"

ARGUMENT OUT OF RANGE IN "routine name"

UNDERFLOW DURING CONVERSION FROM INTERVAL TO DOUBLE PRECISION IN "routine name"

UNDERFLOW IN "rouitne name"

OVERFLOW IN "routine name"

UNKNOWN ERROR DURING "routine name"

APPENDIX C: SUMMARY OF INTERVAL SUBROUTINE MODIFICATIONS

The following is a list of the routines written for the interval package divided into the categories with the codes [MRC], [MRC/M], and [MUL].

[MRC] - Original form from the Mathematics Research Center with no modifications.

arith2	expon1
expon2	convrt
intc87	relatn
supinf	unints
intbnd	funct3
comput	

[MRC/M] - Original form with modifications.

arith1	bpaxp4
intc84	intc85
intc86	funct1
length	funct2
funct4	funct5

[MUL] - Written specifically for the MULTICS System.

bpaadd	bpasub	bpamul
bpadiv	brounding	unpack
normalize	shift_right	s_mgn_add
aidint	pack	dist
intrdv	intrdf	intrdm
intprv	intpr	intprm
convert_to_binary	convert_to_decimal	convert_fb_dec
get_next_int_number	round_dec	get_char
set_input_pointer	intrap	bpac68
bpac98	intas	set_common
finish		

APPENDIX D: SAMPLE OF INTERVAL FORTRAN PROGRAM

The following Fortran program illustrates the use of the interval data type. The program solves a set of linear equations using Gaussian elimination with partial pivoting. The program is just illustrative of the interval data type and may not be the best method of solving a set of linear equations using interval arithmetic. A listing of the interval program is given followed by a listing of the translated program produced by AUGMENT with the calls to the interval package routines. Following that is a sample of the output produced by the program. In one case the error trapping capability of the interval package is illustrated. In this case it was set up so that if an error occurred the program continued.

```
      INTERVAL A(10,11),X(10),BIG,TERM,PIVOT,CONST,Y,TEMP
      LOGICAL EOF
      INTEGER YES
      DATA YES /3Hyes/
910   WRITE(6,710)
710   FORMAT(/," ENTER NUMBER OF EQUATIONS")
      READ(5,10)NN
10    FORMAT(V)
      M=NN
      N=M+1
      WRITE(6,750)
750   FORMAT(/," ENTER MATRIX")
      CALL INTRDM(A,M,N,EOF)
      LAST=M-1
C
C     START OVERALL LOOP FOR M-1 PIVOTS
C
      DO 200 I=1,LAST
C
C     FIND LARGEST REMAINING TERM IN I-TH COLUMN FOR PIVOT
C
      BIG="0"
      DO 50 K=I,M
      TERM=ABS(A(K,I))
      IF(TERM.LE.BIG) GO TO 50
      BIG=TERM
      L=K
50    CONTINUE
C
C     CHECK WHETHER A NON-ZERO TERM HAS BEEN FOUND
C
      IF(BIG.EQ."0") STOP
C
C     L-TH ROW HAS THE BIGGEST TERM -- IS I=L
C
      IF(I.EQ.L) GO TO 120
```

```
C
C       I IS NOT EQUAL TO L, SWITCH ROWS I AND L
C
        DO 100 J=1,N
        TEMP=A(I,J)
        A(I,J)=A(L,J)
100     A(L,J)=TEMP
C
C       NOW START PIVOTAL REDUCTION
C
120     PIVOT=A(I,I)
        NEXTR=I+1
C
C       FOR EACH OF THE ROWS AFTER THE I-TH
C
        DO 200 J=NEXTR,M
C
C       CONST IS MULTIPLYING CONSTANT FOR THE J-TH ROW
C
        CONST=A(J,I)/PIVOT
C
C       NOW REDUCE EACH TERM OF THE J-TH ROW
C
        DO 200 K=I,N
200     A(J,K)=A(J,K)-CONST*A(I,K)
C
C       END OF PIVOTAL REDUCTION - PRINT REDUCED MATRIX
C
        WRITE(6,501)
501     FORMAT(/," THE REDUCED MATRIX IS AS FOLLOWS:",/)
        CALL INTPRM(1H ,3,1,25,A,M,N)
C
C       PERFORM BACK SUBSTITUTION
C
        DO 500 I=1,M
C
```

```
C       IREV IS THE BACKWARD INDEX, GOING FROM M BACK TO 1
C
        IREV=M+1-I
C
C       GET Y IN PREPARATION
C
        Y=A(IREV,N)
        IF(IREV.EQ.M) GO TO 500
C
C       NOT WORKING ON LAST ROW, I IS 2 OR GREATER
C
        DO 450 J=2,I
C
C       WORK BACKWARD FOR X(N), X(N-1) ..., SUBSTITUTING PREVIOUSLY FOUND
C       VALUES
C
        K=N+1-J
450     Y=Y-A(IREV,K)*X(K)
C
C       FINALLY COMPUTE X
C
500     X(IREV)=Y/A(IREV,IREV)
C
C       PRINT VALUES OF X
C
        WRITE(6,502)
502     FORMAT(/," THE SOLUTION IS AS FOLLOWS:",/)
        CALL INTPR(1H ,1,0,25,X,1,M)
        WRITE(6,950)
950     FORMAT(/," DO YOU WANT TO CONTINUE?")
        READ(5,975)IRESP
975     FORMAT(A3)
        IF(IRESP.EQ.YES) GO TO 910
        ENDFILE 5
        ENDFILE 6
        STOP
```

END

```
C                =====  PROCESSED BY AUGMENT VERSION ,4I =====
C                -----  TEMPORARY STORAGE LOCATIONS -----
C                       INTERVAL
      REAL INTTMP(2,1)
C                -----  LOCAL VARIABLES -----
      LOGICAL EOF
      INTEGER I, IRESP, IREV, J, K, L, LAST, M, N, NEXTR, NN, YES
C                       INTERVAL
      REAL A(2,10,11), BIG(2), CONST(2), PIVOT(2), TEMP(2), TERM(2), X(2
     *     ,10), Y(2)
C                -----  SUPPORTING PACKAGE FUNCTIONS -----
      LOGICAL INTEQ, INTLE
C                =====  TRANSLATED PROGRAM =====
C ===== DATA STATEMENTS ARE NOT PROCESSED BY AUGMENT =====
      DATA YES /3Hyes/
910   WRITE(6,710)
710   FORMAT(/," ENTER NUMBER OF EQUATIONS")
      READ(5,10)NN
10    FORMAT(V)
      M=NN
      N=M+1
      WRITE(6,750)
750   FORMAT(/," ENTER MATRIX")
      CALL INTRDM(A,M,N,EOF)
      LAST=M-1
C
C     START OVERALL LOOP FOR M-1 PIVOTS
C
      DO 30000 I=1,LAST
C
C     FIND LARGEST REMAINING TERM IN I-TH COLUMN FOR PIVOT
C
      CALL INTAS ("0",BIG)
      DO 50    K=I,M
      CALL INTABS (A(1,K,I),TERM)
      IF (INTLE (TERM,BIG))  GO TO 50
```

```
      CALL INTSTR (TERM,BIG)
      L=K
50    CONTINUE
C
C     CHECK WHETHER A NON-ZERO TERM HAS BEEN FOUND
C
C ===== MIXED MODE OPERANDS ACCEPTED =====
      CALL INTAS ("0",INTTMP(1,1))
      IF (INTEQ (BIG,INTTMP(1,1)))  STOP
C
C     L-TH ROW HAS THE BIGGEST TERM -- IS I=L
C
      IF(I.EQ.L) GO TO 120
C
C     I IS NOT EQUAL TO L, SWITCH ROWS I AND L
C
      DO 100   J=1,N
      CALL INTSTR (A(1,I,J),TEMP)
      CALL INTSTR (A(1,L,J),A(1,I,J))
100   CALL INTSTR (TEMP,A(1,L,J))
C
C     NOW START PIVOTAL REDUCTION
C
120   CALL INTSTR (A(1,I,I),PIVOT)
      NEXTR=I+1
C
C     FOR EACH OF THE ROWS AFTER THE I-TH
C
      DO 30000 J=NEXTR,M
C
C     CONST IS MULTIPLYING CONSTANT FOR THE J-TH ROW
C
      CALL INTDIV (A(1,J,I),PIVOT,CONST)
C
C     NOW REDUCE EACH TERM OF THE J-TH ROW
C
```

```
      DO 30000 K=I,N
200   CALL INTMUL (CONST,A(1,I,K),INTTMP(1,1))
      CALL INTSUB (A(1,J,K),INTTMP(1,1),A(1,J,K))
30000 CONTINUE
C
C     END OF PIVOTAL REDUCTION - PRINT REDUCED MATRIX
C
      WRITE(6,501)
501   FORMAT(/," THE REDUCED MATRIX IS AS FOLLOWS:",/)
      CALL INTPRM(1H ,3,1,25,A,M,N)
C
C     PERFORM BACK SUBSTITUTION
C
      DO 500   I=1,M
C
C     IREV IS THE BACKWARD INDEX, GOING FROM M BACK TO 1
C
      IREV=M+1-I
C
C     GET Y IN PREPARATION
C
      CALL INTSTR (A(1,IREV,N),Y)
      IF(IREV.EQ.M) GO TO 500
C
C     NOT WORKING ON LAST ROW, I IS 2 OR GREATER
C
      DO 30001 J=2,I
C
C     WORK BACKWARD FOR X(N), X(N-1) ..., SUBSTITUTING PREVIOUSLY FOUND
C     VALUES
C
      K=N+1-J
450   CALL INTMUL (A(1,IREV,K),X(1,K),INTTMP(1,1))
      CALL INTSUB (Y,INTTMP(1,1),Y)
30001 CONTINUE
C
```

```
C       FINALLY COMPUTE X
C
500     CALL INTDIV (Y,A(1,IREV,IREV),X(1,IREV))
C
C       PRINT VALUES OF X
C
        WRITE(6,502)
502     FORMAT(/," THE SOLUTION IS AS FOLLOWS:",/)
        CALL INTPR(1H ,1,0,25,X,1,M)
        WRITE(6,950)
950     FORMAT(/," DO YOU WANT TO CONTINUE?")
        READ(5,975)IRESP
975     FORMAT(A3)
        IF(IRESP.EQ.YES) GO TO 910
        ENDFILE 5
        ENDFILE 6
        STOP
        END
```

```
ENTER NUMBER OF EQUATIONS
2

ENTER MATRIX
5 3 6
1 8 4

THE REDUCED MATRIX IS AS FOLLOWS:

 [ .500000+01, .500000+01] [ .300000+01, .300000+01] [ .600000+01, .600000+0
 [-.149012-07, .745059-08] [ .739999+01, .740001+01] [ .279999+01, .280001+0

THE SOLUTION IS AS FOLLOWS:

[ .972972+00, .972973+00]
[ .378378+00, .378379+00]

DO YOU WANT TO CONTINUE?
yes

ENTER NUMBER OF EQUATIONS
3

ENTER MATRIX
5 3 6 8
2 8 3 1
9 4 6 2

THE REDUCED MATRIX IS AS FOLLOWS:

 [ .900000+01, .900000+01] [ .400000+01, .400000+01] [ .600000+01, .600000+0
 [ .200000+01, .200000+01]
 [-.298024-07, .149012-07] [ .711111+01, .711112+01] [ .166666+01, .166667+0
 [ .555555+00, .555556+00]
 [-.596047-07, .596047-07] [-.447035-07, .447035-07] [ .248437+01, .248438+0
 [ .682812+01, .682813+01]
```

```
THE SOLUTION IS AS FOLLOWS:

[-.135850+01,-.135849+01]
[-.566038+00,-.566037+00]
[ .274842+01, .274843+01]

DO YOU WANT TO CONTINUE?
yes

ENTER NUMBER OF EQUATIONS
2

ENTER MATRIX
[1,2] [5,6] [8,9]
[12,13] [3,4] [15,16]

THE REDUCED MATRIX IS AS FOLLOWS:

 [ .120000+02, .130000+02] [ .300000+01, .400000+01] [ .150000+02, .160000+(
 [-.116667+01, .107693+01] [ .433333+01, .576924+01] [ .533333+01, .784616+(

THE SOLUTION IS AS FOLLOWS:

[ .596722+00, .110223+01]
[ .924444+00, .181066+01]

DO YOU WANT TO CONTINUE?
yes

ENTER NUMBER OF EQUATIONS
2

ENTER MATRIX
1 1 1
2 2 2
```

```
THE REDUCED MATRIX IS AS FOLLOWS:

 [ .200000+01, .200000+01] [ .200000+01, .200000+01] [ .200000+01, .200000+(
 [ .000000+00, .000000+00] [ .000000+00, .000000+00] [ .000000+00, .000000+(

**********************************************************************
DIVISION BY ZERO DURING intdiv
ARGUMENT 1 = [ .00000000000000000000000000+00, .00000000000000000000000000+(
ARGUMENT 2 = [ .00000000000000000000000000+00, .00000000000000000000000000+(
RESULT =     [-.17014118219281863150345791+39, .17014118219281863150345791+3
**********************************************************************

**********************************************************************
BOUNDS FAULT DURING intmul LEFT ENDPOINT--INFINITY RIGHT ENDPOINT--INFINITY
ARGUMENT 1 = [ .20000000000000000000000000+01, .20000000000000000000000000+(
ARGUMENT 2 = [-.17014118219281863150345791+39, .17014118219281863150345791+3
RESULT =     [-.17014118219281863150345791+39, .17014118219281863150345791+3
**********************************************************************

**********************************************************************
BOUNDS FAULT DURING intsub LEFT ENDPOINT--NO FAULTS RIGHT ENDPOINT--INFINITY
ARGUMENT 1 = [ .20000000000000000000000000+01, .20000000000000000000000000+(
ARGUMENT 2 = [-.17014118219281863150345791+39, .17014118219281863150345791+3
RESULT =     [-.17014118219281863150345791+39, .17014118219281863150345791+3
**********************************************************************

THE SOLUTION IS AS FOLLOWS:

[-.850706+38, .850706+38]
[-.170142+39, .170142+39]

DO YOU WANT TO CONTINUE?
no
STOP
```

APPENDIX E: USE OF intfor COMMAND ON MULTICS

Function: translates interval fortran programs into standard Fortran and compiles the translated segment if requested.

Syntax: intfor path -control_args-

Arguments: path is the pathname of an interval Fortran source segment; a suffix of ".interval" is assumed and need not be given.

Control arguments: -no_translated_source, -nts does not create the translated Fortran segment in the current working directory; default is to create a translated source segment with the suffix ".fortran". Any error messages produced during translation will be in this segment (see Notes below).

-convert_real_to_interval, -cri all variables of type real in the source will be considered to have the type interval.

-no_compile, -nc the translated source will not be compiled; default is to compile.

-force_compile, -fc there will be an attempt to compile the translated source even if there are errors during the translation.

-augment_list, -als a segment is produced by the AUGMENT precompiler (see Notes below) that consists of a listing of the input source segment passed to AUGMENT. Any error messages produced during translation will also be in this segment. The segment will have the suffix "agm.list"

The rest of the control arguments are any arguments acceptable to the Fortran compiler. These arguments will be passed to the Fortran compiler if a compilation is to be performed.

Notes: The intfor command uses the AUGMENT precompiler to produce the translated source. AUGMENT will display how many errors there were in the processing or translation phase. The error messages will be in the translated source segment next to the statement that caused the error. Therefore if a program is being translated for the first time, a translated source segment should be created in case there are errors. The error messages can be located in the translated source segment by searching for the character string "*****" which is attached to each error message.

Currently the input segment must be in the standard Fortran 80 column format using all uppercase letters.

In order to run an interval program the user must have the search rules set to search the directory >udd>beta>r&d>a.r&d>int_routines. This setting of the search rules can be done by a "ssr >udd>beta>r&d>a.r&d>isr" before running the interval program.

APPENDIX F: SUMMARY STATEMENT ON RUNNING OF 'AUGMENT' ON MULTICS

Computer Science Department
University of Southwestern Louisiana
USL P. O. Box 4-4330
Lafayette, Louisiana 70504
May 14, 1976

Fred D. Crary
Mathematics Research Center
University of Wisconsin - Madison
610 Walnut St.
Madison, Wisconsin 53706

Dear Fred,

Thank you for sending us the source code of AUGMENT. AUGMENT is now up and running on MULTICS. The problems encountered when installing AUGMENT and the changes that were made to the source of AUGMENT are outlined below.

There were four problems encountered when AUGMENT was being installed. These problems are outlined below in the order in which they occurred:

1. In the function trefnd an assumption is made that the minimum number of words required to store a string cell is 2. This assumption is valid only if the length of the string and the string cell identifier of 1 in the low order 5 bits are both stored in the same word. When the pack routine (which prepares the string cells) was first written, the string cell identifier of 1 and the length of the string were stored in separate words. Also the characters of the string were placed in separate words. Therefore the minimum number of words required to store a string cell was 3. When the trefnd function determined that it should store a blank in two words, what really was used was 3 words. Therefore, some data was overwritten when the string cell was created. The problem was solved when the length and string cell identifier were packed into the same word. Also, the characters of the string were packed 4 characters to a word. In this case the minimum number of words required to store a string cell was 2.

2. In the subroutines oprcrd and tstcrd, a statement function is used that returns a logical result. In both subroutines, the statement function was not returning the correct results. It has not been determined at this time why the statement functions were working incorrectly. The statement function was changed to an external function and it worked properly thereafter. This problem dealt with the MULTICS system and is not an error in AUGMENT.

3. In the function ntrlin, the return result of the function is assigned in the first statement of the function. The first argument passed to the function was being overwritten by the results of the function. It is not now known for sure what really happens, but the first argument was definitely being overwritten. Since the first argument is used later on, erroneous results occurred. The reason why the result of the function could be assigned in the first statement is that the result is not a function of the arguments. The problem was solved by moving the statement assigning the result of the function to a location closer to the return statement and after the use of the first argument. This problem dealt with the MULTICS system and is not an error in AUGMENT.

4. In the subroutine stdo a reference is made to the function smmake with a wrong type argument. The reference is as follows:

```
smmake (begin, last, -indtyp, 1, 0, .true., .false.)
```

The 6th argument, ".true.", should not be a logical argument, but an integer argument. From the context of the reference it appears that the argument should be -1. Apparently on a machine that represents the value ".true." as an integer -1, this function reference would work correctly, but not on MULTICS. The argument should be -1 and was thus changed.

The machine dependent primitive routines were first written in PL/I and Fortran, but were later written entirely in PL/I. ALM (Assembly Language of MULTICS) was not chosen at this time for the primitive routines because its use on MULTICS is discouraged and documentation of ALM is hard to come by.

The only other changes made to the source code of AUGMENT, besides those changes outlined above, had to do with Fortran common. The two block data subprograms (maindata and processdata) were changed to subroutines with the common areas being initialized by assignment statements. These two subroutines were then called at the beginning of the driver routine. MULTICS Fortran supports block data subprograms, but they still had to be called from the driver routine. MULTICS Fortran only supports block data subprograms in order to be compatible with other Fortrans. The reason why the block data subprograms were changed to subprograms is that when the common area is initialized by a block data subprogram, it cannot be reinitialized to its original values, even if the block data subprogram is called again.

An additional routine written in PL/I was needed to initialize pointers to the common blocks needed by the machine dependent primitive routines. This routine is called at the beginning of the driver routine and after the calls to the routines that initialize common.

In the maindata subroutine a dummy assignment had to be made to a variable in the common blocks /CARD/ and /STOR/ in order for those common

blocks to be created. MULTICS creates a common block only when a reference is made to a variable in the common block. The above two common blocks are needed by the primitive routines and therefore they must be created before a pointer to them can be obtained.

A routine was written in PL/I that sets up the I/O files and actually invokes the AUGMENT driver routine. This routine is the one invoked by the user when using AUGMENT.

If any additional problems occur in using AUGMENT we will inform you. We will be glad to send you a copy of the primitive routines that we wrote along with the additional routines that were also written. All of these routines are written in PL/I and are dependent on the MULTICS system. Please inform us how you would like the routines sent to you. Thank you again for sending us the source code and thank you for your time and trouble in preparing the tape for us.

Sincerely,

Bruce D. Shriver
Eric K. Reuter

In accordance with letter from DAEN-RDC, DAEN-ASI dated 22 July 1977, Subject: Facsimile Catalog Cards for Laboratory Technical Publications, a facsimile catalog card in Library of Congress MARC format is reproduced below.

Podlaska-Lando, D
Implementation and evaluation of interval arithmetic software; Report 2: The Honeywell MULTICS System / by S. Podlaska-Lando, Eric K. Reuter, Bruce D. Shriver, Computer Science Department, University of Southwestern Louisiana, Lafayette, La. Vicksburg, Miss. : U. S. Waterways Experiment Station ; Springfield, Va. : available from National Technical Information Service, 1979.
62, [24] p. ; 27 cm. (Technical report - U. S. Army Engineer Waterways Experiment Station ; 0-79-1, Report 2)
Prepared for Office, Chief of Engineers, U. S. Army, Washington, D. C., under Contract Nos. DACA39-76-M-0249 and DACA 39-77-M-0101.
References: p. 61-62.

1. Algorithms. 2. Computer systems programs. 3. Evaluation. 4. Honeywell MULTICS System. 5. Interval arithmetic. I. Reuter, Eric K., joint author. II. Shriver, Bruce D., joint

(Continued on next card)

Podlaska-Lando, D
Implementation and evaluation of interval arithmetic software; Report 2: The Honeywell MULTICS System ... 1979.
(Card 2)

author. III. Louisiana. University of Southwestern Louisiana, Lafayette. Dept. of Computer Science. IV. United States. Army. Corps of Engineers. V. Series: United States. Waterways Experiment Station, Vicksburg, Miss. Technical report ; 0-79-1, Report 2.
TA7.W34 no.0-79-1 Report 2