AN OPERATIONAL INTERVAL ARITHMETIC

R. E. Boche
Lockheed Missiles & Space Company
Palo Alto, Calif.

Submitted in abstract or outline form only to the IEEE Computing Devices Committee for presentation at the IEEE-Illinois Institute of Technology-Northwestern University-University of Illinois National Electronics Conference, Chicago, Ill., October 28-30, 1963. Manuscript submitted August 1, 1963; made available for printing August 8, 1963.

Paper No. **CP**
63-1431
CD-1

Interval Arithmetic, Error Analysis, Digital Computer, Programming, Numerical Methods, Number System, Numbers, Differential Equations, Matrix, Marco, Input, Output.

Paper No. CP
63-1431
CD-1

AN OPERATIONAL INTERVAL ARITHMETIC

R. E. Boche*

A scheme to perform automatic error analysis in digital computation has been described by R. E. Moore.[1,2] The basis of the scheme is a method for performing computations with intervals of real numbers. The computations may be performed on a digital computer in a manner that guarantees the containment of an infinite precision real number result in the computed interval.

We will first describe the arithmetic and present some results of numerical operations with it. We will then consider the implementation of the arithmetic itself on digital computers and finally discuss programming procedures developed and in development for interval analysis.

Interval Arithmetic

We will first describe an exact interval arithmetic. Later we will discuss the implementation of interval arithmetic on a digital computer to contain exact, or infinite precision, interval arithmetic and in turn infinite precision real arithmetic. We state in advance that the computations to be performed are such that the infinite precision interval is a subset of the computed interval.

We denote by $[a,b]$ the closed interval of real numbers x such that $a \leq x \leq b$. Thus any real number element x to be employed in computation may be represented by an interval whose end points are rational numbers of limited precision. We state the following definitions for interval operations restricting only to intervals whose end points are real.

For the interval numbers $[a,b]$ and $[c,d]$ we have:

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

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

$$[a,b] \cdot [c,d] = [\text{MIN}\ (ac, ad, bc, bd), \text{MAX}\ (ac, ad, bc, bd)]$$

and if the closed interval $[c,d]$ does not contain 0,

$$[a,b] \ / \ [c,d] = [a,b] \cdot [1/d\ ,\ 1/c]$$

The properties of interval numbers have been investigated extensively by Moore.[1,2] We state a few here for use in the following discussions.

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

$[a,b] < [c,d]$ iff $b < c$

$[a,b] > [c,d]$ iff $a > d$

*Lockheed Missiles and Space Company, Palo Alto, California

$[a,b] \approx [c,d]$ is a notation used here to denote that the intersection of the intervals $[a,b]$ and $[c,d]$ is non-null.

Interval arithmetic contains real arithmetic since for intervals of the form $[a,a]$ we have

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

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

$$[a,a] \cdot [b,b] = [ab , ab]$$

and if $b \neq 0$

$$[a,a] / [b,b] = [a/b , a/b]$$

The intervals $[0,0]$ and $[1,1]$ serve as additive and multiplicative identities respectively.

Computing with Interval Arithmetic

The description of the arithmetic and statement of its purpose immediately suggest some possible applications. The most obvious of these potential applications is the analysis of error in numerical computations involving large numbers of arithmetic operations on a computer to reach a result which with an infinite precision computation would be exact. Matrix operations performed with exact methods are in this category. In general any set of arithmetic operations for whatever purpose desired may be performed in interval arithmetic to determine the amount of error due to round-off or truncation and its growth. Such computations might apply to numerical approximations which are not exact. In such cases there is no intention to bound an infinite precision real result with a computed interval but only to analyse the contribution of computing error to result error.

The availability of interval arithmetic has led to the investigation of numerical methods developed specifically for computation with intervals. Two examples of such methods for the numerical solution of differential equations and the integration of functions are used in sample results to follow.

Another important area of usefulness is to reflect into a numerical computation a degree of uncertainty in the input data. Such an uncertainty might result from inaccuracies in experimental observation or even from data formating alone as in analog to digital conversion of data.

Examples and Results

Programs for numerical solution of differential equations using interval arithmetic have been prepared. The method used is derived from a truncated Taylor's series with remainder. The remainder is itself bounded by interval arithmetic computations.

For the equation

$\frac{dy}{dx} = y^2$ with initial condition $y(0) = 1$

the exact solution is $y(x) = \frac{1}{1 - x}$

At $x = \frac{1}{4}$, $y(\frac{1}{4}) = \frac{1}{1 - \frac{1}{4}} = 4/3$.

The floating point octal representation of 4/3 on the computer used is 201 525252525. The interval endpoints computed by the program using 12 terms and remainder are

$y(\frac{1}{4})$ = [201 525252524, 201 525252530] . (The 201 is a biased exponent.)

Thus, the computed interval has width 4 in the ninth significant octal place and bounds the exact solution.

The following 7th order system of differential equations was treated using the first nine terms of the Taylor's series with remainder.

$$\ddot{x} = -\mu x/(x^2 + y^2 + z^2)^{3/2}$$

$$\ddot{y} = -\mu y/(x^2 + y^2 + z^2)^{3/2}$$

$$\ddot{z} = -\mu z/(x^2 + y^2 + z^2)^{3/2}$$

$$\dot{t} = 1 \qquad \mu = 0.98$$

Programs were prepared in parallel using machine single precision floating point arithmetic and interval arithmetic. The results obtained from both programs after approximately one fourth of an orbit and the initial conditions used are given as Table II.

Twelve steps of uniform size were taken to reach the quarter orbit point. The interval arithmetic program ran at a rate of about 4 seconds per point while the machine arithmetic program took 2 seconds per point.

The above examples involve the use of a numerical method designed for computing with intervals. The following computations will serve to illustrate an interval method and result.

$$\int_1^2 (x - 1.5)^2 \, dx = \frac{1}{12}$$

Using the trapezoidal rule with one step we evaluate the function at 1 and 2 and get $(\frac{1}{4} + \frac{1}{4})/2$, or the result $\frac{1}{4}$. By an interval method[1] we evaluate the function over the interval [1,2] as follows:

$$([1,2] - [1.5,1.5])^2 = [-0.5,0.5]^2 = [0,\tfrac{1}{4}]$$

The interval result again contains the exact solution. The point here is only to suggest what an interval method might be like. Moore has developed rigorously this and many others.

The determinant of the matrix $a_{ij} = \frac{1}{i+j-1}$ was computed using the pivot method with no pivotal element searching. The use of this particular matrix serves as an illustration of uncertainty in input data. The element 1/3 must be entered as the interval [0.33333333, 0.33333334] . The interval results for the cases:

3 by 3 $[0.46296249 \times 10^{-3} , 0.46296365 \times 10^{-3}]$

4 by 4	$[0.16534366 \times 10^{-6}, 0.16536174 \times 10^{-6}]$
5 by 5	$[0.30721218 \times 10^{-12}, 0.31768488 \times 10^{-12}]$

Considering the degree of uncertainty in the input elements, the number of computations performed, and the lack of refinement in the method, the resulting interval widths seem reasonable.

An interval matrix inversion program has been completed and run for a limited number of cases using the Crout method with pivot searching. The tri-diagonal matrix with $a_{11} = 1$, $a_{ii} = 2$, $a_{i-1,i} = -1 = a_{i,\ i-1}$, $i = 2,3,\ldots,N$. All other elements zero was inverted for the cases 3 by 3 and 10 by 10. In both cases the results were exact to the eight decimal places printed. The inverse is the matrix with elements $a_{ij} = N - \mathrm{MAX}\ (i, j) + 1$. The 3 by 3 inverse matrix was re-inverted. The computed inverse elements had a median width of 3 in the last of the 8 decimal places printed and a maximum width of 11.

Comments and Conclusions

At this stage it should be pointed out that interval width does not necessarily grow with repeated computation. For verification of this see Moore[1], pg. 13, where an iterative square root process is shown to converge with simultaneously decreasing interval width. The following computation which converges to an interval of width zero also illustrates this point:

$$I_0 = [0,1]$$

$$I_{N+1} = I_N \left[\tfrac{1}{4},\tfrac{1}{2}\right]$$

The propagation and growth of interval width in some problems may prove so great as to give an interval from which no conclusion of meaningful impact may be drawn. Such a result certainly may not be taken to mean that the single precision result which is always contained in the interval is not valid. However, the opposite case presents a very useful result by assuring that a single precision result contained in a relatively narrow interval is indeed accurate to the number of places indicated.

Inapplicability to particular problems should not preclude the availability of interval arithmetic as a working and useful computation center tool. There exist classes of problems to which interval arithmetic gives a valid and useful bound.

Interval Arithmetic Programs

The remainder of this report will be devoted to our experience in attempting to implement interval arithmetic as a general purpose computation center problem analysis tool. To begin with, a set of interval arithmetic routines had been prepared by D. Thoe for the IBM 709 computer. These routines served as an excellent starting point and were extensively modified and augmented. The four basic arithmetic operations were available from the earlier programming efforts as were some of the logical instructions. An instruction repertoire is given as Appendix I.

A macro programming technique was employed using the "FAP" assembly program language.[3] An example and illustration of the macro form and its use is included as Appendix II. Although the use of the macro is not an essential concept, its usefulness and convenience warrants its prominence in the programming descriptions.

The underlying principle on which the interval arithmetic programs are based is that the computed interval must contain the infinite precision real number result. In order to be useful, this assurance must be given without undue sacrifice of interval width. Given two intervals whose end points are machine representable floating point single precision numbers, and given that we wish to operate on two real numbers, each contained in their respective intervals, we may operate on the end points of the intervals in the manner dictated by the operation to be performed and the definition for such an interval operation stated earlier. If we do so using single precision arithmetic unrounded we may assure containment of the infinite precision result by "extending up" the upper end of the result interval and "extending down" the lower end of the result interval. Although this process will provide our vital guarantee of containment, it will result in undue growth of interval width in a great number of cases. Therefore, we must refine our rules of procedure to minimize the growth of interval width without sacrificing containment.

We will consider a machine employing the binary number system and refer to bits in extending up or down. Clearly, rounding is not adequate because if, for example, the next two most significant bits beyond the single precision word length of the positive upper end of an interval are 0, 1 rounding would not take place and containment may fail. Thus, we speak of the process of adding (or subtracting) a low order bit to an interval end point as extension. Note also that automatic rounding by the computer if not suppressable by the programmer is a hindrance rather than a help. To implement floating point interval arithmetic we must make use of machine "left overs". Such "left overs" are the result of shifting operations performed automatically by the floating point hardware to arrive at compatible scaling, or of arithmetic operations that generate more bits than can be contained in the single precision word, e.g., multiply and divide. Fortunately, the leftovers are accessible to the programmer of many digital computers.

Now, getting back to interval widths and their growth, we may see that undue growth would result from extending downward a positive lower end point of a computed interval or extending upward a negative upper end point. Thus seperation by sign must be employed. Also note that on most digital computers, bumping the low order bit by one bit may result in a carry into the exponent field or an overflow. Therefore, the extension of intervals must be approached carefully.

In a typical digital computer floating point operation a single precision operation leaves a single precision result in one register and a less significant single precision result in another register. In many cases the mantissa of the low order result may be zero and may indicate that all bits of significance are contained in the high order single precision result. In such cases we have the required containment without interval extension. In that case extension would cause unnecessary interval width growth. But, testing for zero in the mantissa may not be enough. If the mantissa of the low order portion is not zero, extension may be warranted, depending on the sign. If extension is warranted by sign and the mantissa of the low order portion is zero, extension may be necessary anyway. If the scaling of the two interval end point numbers prior to the interval operation was different by some amount (a function of computer word length and other hardware characteristics) the smaller end point may have been shifted off the low order end of the low order result register. If this possibility arises, then there is no recourse but to separate the exponent portions of the intervals as they appeared before operation and check to see if extension should occur.

The above considerations are characteristic of a finite word length digital computer interval arithmetic. The rigorous containment required along with the conservation of interval width results in unfortunately slow program speeds when compared with an arithmetic performed by the hardware. This is not too surprising as the same may be said of other non-hardware arithmetics. Double precision before its implementation by hardware was also a slow painful process as are present day N-precision arithmetic programs.

There are some additional techniques which in certain circumstances may be employed to further conserve interval width without sacrificing containment.

A particular case resulted in the provision of the instruction "SQR" in the instruction repertoire. This instruction squares an interval number rather than multiplying the number times itself. If A is the interval number $[A', A'']$ then using the "SQR" instruction,

$$A^2 = [y \mid y = x^2 , \ A' \leq x \leq A'']$$

is the square of A. If this computation is performed by multiplication,

$$A \times A = [y \mid y = xz , \ A' \leq x \leq A'' , \ A' \leq z \leq A'']$$

which may result in a wider interval than necessary if A spans zero. Thus for computed intervals,

$$A^2 \subset A \times A.$$

The sequence in which computations are performed also influence the resulting interval widths just as it influences accuracy in finite precision real arithmetic. For example:

$$\left[\frac{1}{3}, \frac{1}{3}\right] [0, 2]^N$$

will result in a narrower interval if $[0, 2]^N$ is computed first and then multiplied by $\left[\frac{1}{3}, \frac{1}{3}\right]$. Another example is:

$$[0, 1] \cdot \left([1, 2] + [-1, 0]\right) = [0, 1] \cdot [0, 2] = [0, 2]$$
$$[0, 1] \cdot [1, 2] + [0, 1] \cdot [-1, 0] = [0, 2] + [-1, 0] = [-1, 2]$$

where

$$[0, 2] \subset [-1, 2]$$

The testing of relative sizes of range numbers is not adaptable to a single absolute rule of procedure. However, the user has available a number of instructions and alternative rules and procedures which should prove adequate for implementing his definition of size.

To be absolutely rigorous you may say: $A > B$ iff $x \in A, y \in B \Rightarrow x > y$. This definition is readily applied by use of the single range instruction "CAS", but the separation is not exhaustive when $A \cap B$ is non-null. Then, in order to preserve the infinite precision result, you must fail to arrive at a result in a number of cases.

If instead we consider $A > B$ to mean $\exists x \in A \ni \quad y \in B \Rightarrow x > y$, then the division of cases is exhaustive and may be accomplished by successive application of range instructions CAS, MEET, and SUBINT.

The latter definition is not completely defensible as, for example, $[0, 1000]$ is taken as being greater than $[999.9, 999.9]$. Although it is true that based on the interval computation, the first interval may indeed contain the greater real number, it is not intuitively appealing if the spread actually characterises the degree of uncertainty. In such a circumstance, the mid-point of the interval may be a better measure. In such a case as searching for Pivotal elements in a matrix, it may prove adequate to examine only the upper ends of the absolute values of the interval elements.

The Input/Output Problem

In order to provide a complete interval arithmetic programming system, it will be necessary to provide a set of interval input and output programs. Many different input and output programs already exist. The mechanics of such programs are not of interest here. The decimal-octal conversion is an important problem. The arithmetic uses internally octal numbers in floating point format. We could therefore, make use of an existing program to input in floating octal format both the end points of an interval. Since our number system is decimal, the conversion by hand to octal would be inconvenient. If instead, we make use of a standard program and input the interval end points in floating decimal format, the conversion from decimal to octal may of necessity be an approximate one and containment may fail. We must, therefore, convert from a floating decimal input to a floating octal machine representable number and then determine whether or not interval extension is necessary. Care will still be required to assure that the floating decimal interval entered contains the infinite precision real number.

An additional problem may arise. If a floating decimal number to be input contains more digits of significance than can be accomodated by the input program, then that floating decimal number must be represented by an interval in which the least significant figures of the end points differ by one. The resulting conversions to octal and the necessary extensions may result in an octal representation of the interval differing by more than one in its least significant figure.

A better procedure would seem to be the inputting of the floating decimal number to a larger number of significant figures than may be retained. This could be followed by truncation of the end point of least absolute value and extension of the other end point to form an interval when the truncated portion is not zero. The result would be an interval of width no greater than one in its least significant figure. However, the number of significant figures which must be entered to assure containment for a computer of some particular word length is not determined. If this proves to be a large number (perhaps infinite) the procedure must be rejected.

In any event, the interval input program must allow for input of intervals to reflect a degree of uncertainty in the data by the user. For convenience, provision should be made for input of a single number with automatic conversion and formation of an octal interval.

The output problem is much less difficult. The interval end points to be output are floating octal numbers of fixed length. The conversion from octal to decimal may generate more decimal digits than provided for by the output formats. Extension must then be performed on the binary representations of the decimal digits to be output in much the same manner as described for the arithmetic extensions. The procedure, though tedious, is straight forward.

Execution Time

Table I gives execution times for interval instructions in cycles for the IBM 700 series computers. The times vary depending on the different extension procedures necessary as described previously. The zero cases are not included, but in this particular set of programs, such cases are treated separately and much more rapidly.

As in most programming efforts, there is room for improvement. A set of arithmetic programs is under development which on final checkout should be significantly faster. For compatibility, the machine instruction repertoire has been restricted to that of the IBM 709. Additional savings of cycles may be realized with the expanded instruction repertoire of the IBM 7094. Cycle times in microseconds are;

Model IBM	709	7090	7094
Microseconds	12	2.18	2

The elapsed time for the IBM 7094 reflects the machine's instruction look ahead feature. This feature may contribute a saving of up to a single cycle for each machine instruction executed.

We arrive at an over-all timing ratio for interval versus machine floating point instructions of around 4 or 5 to one. This is with the presently far from perfect version of the interval programs. We conjecture a minimum ratio of around two to one since we must, of course, perform the specified arithmetic operation on both interval end points. It is realistic to assume that in a great majority of computation center problems to which interval arithmetic might prove applicable, the proportion of total program running time which is devoted to actual arithmetic computation is quite low. Even if we conjecture a proportion of as much as one half total run time for arithmetic, we arrive at program run time ratios of approximately two to one. This seems a reasonable price to pay for a result which may be unattainable with other methods.

Programming Procedures

The above discussion has dealt primarily with considerations and characteristics inherent to the design of the arithmetic itself. We now wish to consider the preparation of programs employing the arithmetic. It will be necessary to make reference to the particular programming language involved[3,4] in order to adequately describe these procedures.

In preparing these interval arithmetic programs an important underlying philosophy was to place as few restrictions as possible or reasonable on the user and, further, to attempt to document alternatives and changes which the user could make to suit his particular objectives. This seemed particularly important since we feel that the surface has been barely scratched in finding problem areas to which interval arithmetic may be meaningfully applied. However, there is nothing so sure to discourage a potential user as a huge document of alternatives from which he must select, with inadequate knowledge, to guide his selection. In order to provide a workable approach it was necessary to document a single selected version of the programs and usage techniques. It is recognised and noted that considerable efficiency may be gained by modification of technique in some particular direction for a particular application. But, the user may still follow a set of step by step instructions to convert his normal assembly program code to an interval arithmetic code by substituting the appropriate macro instructions for the normal floating arithmetic and transmissive instructions in the computational segments of his program.

This procedure has proven to be a very useful one. A number of existing FORTRAN programs were selected from the computation center. In advance of the selection, a small deck was prepared which contained the macro definitions to be employed and the binary decks of interval arithmetic. The FORTRAN source programs were next obtained and examined. A set of rules of procedure (also prepared in advance) was applied. The following steps were taken:

1.) The FORTRAN program contained a dimensioned array. Two choices were indicated; either write the dimension statement to correspond with those selected in the macros or change a few cards in the macros to conform to the required array sizes. The second of these is warranted only under extremes of storage limited programs. The dimension statement was changed.

2.) A "COMMON" storage specification card was added to provide working area for the arithmetic and arrange storage allocation for interval numbers.

3.) The program was recompiled to obtain an assembly program listing of the compiled program.

4.) The resulting listing was examined to find the sections of the code where computation was actually performed. In all the cases selected, this portion of the code was quite small relative to the indexing, book-keeping and other portions. The computation region was lined out and translated to coding sheets in a line by line translation process. This process had been thoroughly described in advance. The operation codes were the only parts of each instruction requiring alteration in order to perform an interval operation as opposed to the indicated single precision machine operation. It is of great significance that no changes in either the address or index designating portion of the instructions were required. (The only exception to this was to make any numerical constants employed interval numbers.)

5.) The storage map of the compilation was then consulted. "Variables not appearing in common or dimension statements" were listed on the coding sheets to allocate storage of two cells for each. A list of "symbols not appearing in source program" must also be compared against the assembly listing and storage allocated to any that do not appear in the listing.

6.) The translated coding and the augmenting storage allocation was keypunched. The entire listing with translated segments lined out was then keypunched.

7.) The resulting decks were merged and the macro definition deck was added. The resulting program was assembled and run with the binary decks of the previously assembled interval arithmetic.

A number of comments on the results of such a procedure are in order. The first and foremost comment is that a useful and usable program may be created in this fashion. In order to gain perspective with regard to the resulting effectiveness of the process a particular program will be referenced. The Scientific Computation Center was requested to provide a program for matrix inversion. The program received used the Crout method with pivot searching.

The program was in subroutine form. Steps one through seven as detailed above were followed by the author. The total time spent, exclusive of keypunching, was one hour and fifteen minutes. The result is a working matrix inversion program employing interval arithmetic. The program listing as generated by the compiler consisted of 637 lines of code. Of these, only 56 were arithmetic instructions requiring translation. The translated instructions employing macros result in an expansion in ratio of about five to one of the original arithmetic instructions after assembling. In addition, 49 lines of storage allocation code were written. The resulting program is about 920 lines of assembly coding.

It may be argued that the resulting code is inefficient. To a point, this is true. However, it must be argued that the resulting code is proportionately less inefficient than the initial FORTRAN code. The effect on computer running time of the inherent inefficiencies of the compiler language is retained but its contribution to over-all program running time is decreased in proportion as the running time for essential computation increases because of the slower arithmetic. Thus, we may say that the resulting interval arithmetic program is in a sense more efficient than the original FORTRAN program.

This suggests that programs to be coded in interval arithmetic may be coded first in FORTRAN and then converted to interval arithmetic. Such a procedure should appeal to a computation center aware of the disparate amount of time required for coding in a compiler language as opposed to an assembly language, and further, the level of experience required for the translation process is much less than that required for direct programming.

The next and inevitable step for an operational interval arithmetic is to automate the translation process. If, indeed, it may be performed as readily as described above, it should prove adaptable to an interpretive programming process, and it does. At the time of this writing, the automatic translation process for interval arithmetic coding is not yet operational, but work is progressing most promisingly.

APPENDIX I

Instruction Repertoir

In the following list $A = [A', A'']$ will represent the interval number contained in the interval accumulator. $I = [I', I'']$ will denote the interval number specified by the instruction address. $Z = [Z', Z'']$ will denote the interval number zero, i.e. $Z' = Z'' = 0$. The symbol $\emptyset$ will denote the null set. NI refers to the next sequential instruction in the program. The standard set theory notation for unions, intersections, and subsets is employed. Additionally, the symbol, $\approx$, will be used as follows: $A \approx I$ implies that the intersection of A and I is non-null, i.e. $A \cap I \neq \emptyset$. The symbol "$\longrightarrow$" may be read as "replaces".

ADD	$A + I \longrightarrow A$
SUB	$A - I \longrightarrow A$
MULT	$A \times I \longrightarrow A$
DIV	If $I \cap Z = \emptyset$, $A/I \longrightarrow A$ If $I \cap Z \neq \emptyset$, error stop occurs
LOAD	$I \longrightarrow A$
STO	$A \longrightarrow I$
NLOAD	$-I \longrightarrow A$
NSTO	$-A \longrightarrow I$
SQR	$I^2 \longrightarrow A$
ABS	$\lvert I \rvert \longrightarrow A$
INVS	$[1,1] / I \longrightarrow A$
CAP	If $A \cap I \neq \emptyset$, $A \cap I \longrightarrow A$, take NI + 1 If $A \cap I = \emptyset$, take NI
CUP	If $A \cap I \neq \emptyset$, $A \cup I \longrightarrow A$, take NI + 1 If $A \cap I = \emptyset$, take NI

CAS	If A $>$ I, take NI If A $\approx$ I, take NI + 1 If A $<$ I, take NI + 2
MEET	If I $\subset$ A, take NI If I $\equiv$ A, take NI + 1 If A $\subset$ I, take NI + 2 If A $\cap$ I $\neq \emptyset$ and none of above is true, take NI + 3 If A $\cap$ I = $\emptyset$, error stop occurs
SUBINT	If A' = I", take NI If I' < A' < I" < A", take NI + 1 If A' < I' < A" < I", take NI + 2 If A" = I', take NI + 3 If none of above is true, error stop occurs If A $\cap$ I = $\emptyset$, error stop occurs
SQRT	If I' $\geqslant$ 0, $\sqrt{I} \longrightarrow$ A If I' $<$ 0, error stop occurs

APPENDIX II

Sample Macro Definitions

Common storage of an interval array of dimension 50 by 50 is assumed.

```
RNG       MACRO     Y,T,OP
          SXA       *+6,4
          CLA       Y,T
          STO       OPER
          CLA       Y-2500,T
          STO       OPER+1
          TSX       $'OP',4
          AXT       **,4
RNG       END

LOAD      MACRO     Y,T
          CLA       Y,T
          STO       ACC
          CLA       Y-2500,T
          STO       ACC+1
LOAD      END

STOR      MACRO     Y,T
          CLA       ACC
          STO       Y,T
          CLA       ACC+1
          STO       Y-2500,T
STOR      END
```

Common storage of four locations for interval arithmetic pseudo registers, ACC and OPER is also required.

Sample Line by Line Translation

```
Assembly listing of           Translated program to
FORTRAN program               use interval arithmetic

•                             •
•                             •
•                             •
SXD     C)102,4               SXD     C)102,4
LXD     C)105,2               LXD     C)105,2
LXD     I1,1                  LXD     I1,1
LDQ     A+1,1                 LOAD    A+1,1
FMP     A+1,2                 RNG     A+1,2,MULT
FAD     A+1,4                 RNG     A+1,4,ADD
STO     A+1,4                 STOR    A+1,4
TXI     *+1,1,50              TXI     *+1,1,50
TXI     *+1,2,1               TXI     *+1,2,1
TXL     31A,2                 TXL     31A,2
•                             •
•                             •
•                             •
```

ACKNOWLEDGMENTS

R. E. Moore aided and supported this work in all stages of its development. An earlier arithmetic program by Dale Thoe, a differential equations program by Ann Davison, and suggestions from S. Shayer and W. Lowney have been of great assistance.

REFERENCES

1. Interval Arithmetic and Automatic Error Analysis in Digital Computing, R. E. Moore. Technical Report No. 25, Applied Mathematics and Statistics Laboratory, Stanford University, Stanford, Calif., 15 Nov. 1962.

2. Automatic Error Analysis in Digital Computation, R. E. Moore. Technical Report No. 48421, Lockheed Missiles and Space Co., Sunnyvale, Calif., 28 Jan. 1959.

3. IBM 709/7090 Programming Systems: FORTRAN Assembly Program (FAP). Reference Manual, Form C28-6235, IBM Corp., Poughkeepsie, N. Y., September 1962.

4. IBM 709/7090 FORTRAN Programming System. Reference Manual, Form C28-6054-2, IBM Corp., Poughkeepsie, N. Y., January 1961.

TABLE I. ARITHMETIC TIMING

Interval Instruction	Add & Sub.	Mult.	Div.
Cycles, single precision, IBM 709 & 7090	6 - 15	2 - 13	3 - 13
Cycles, interval, IBM 709 & 7090	82 - 104	51 - 85	42 - 74
Cycles, single precision, IBM 7094	2 - 12	2 - 5	3 - 9
Cycles, interval, IBM 7094	74 - 98	51 - 69	42 - 66
Number of machine instructions in interval program	44	30	24
Elapsed time in cycles for IBM 7094 interval program	30 - 54	21 - 39	18 - 42
Approximate times in microseconds:			
Single precision IBM 709	180	156	156
Single precision IBM 7090	33	28	28
Single precision IBM 7094	22	8	16
Interval IBM 7094	84	60	60
Approximate ratio, Int./S. P.	4 to 1	7 to 1	4 to 1

TABLE II. DIFFERENTIAL EQUATIONS EXAMPLE

Variable	f(0)	After 1/4 Orbit Machine Arithmetic	Interval Arithmetic
X	0.04	3.7193×10^{-3}	$[3.7173 \times 10^{-3}, 3.7212 \times 10^{-3}]$
Y	0	3.9097×10^{-2}	$[3.9096 \times 10^{-2}, 3.9099 \times 10^{-2}]$
Z	0	0	$[0, 0]$
$\dot{X}$	0	-4.9775	$[-4.9777, -4.9773]$
$\dot{Y}$	4.9	3.7350×10^{-1}	$[3.7312 \times 10^{-1}, 3.7388 \times 10^{-1}]$
$\dot{Z}$	0	0	$[0, 0]$
t	0	1.1891×10^{-2}	$[1.1890 \times 10^{-2}, 1.1891 \times 10^{-2}]$