SOME EXPERIMENTS USING INTERVAL ARITHMETIC

Eric K. Reuter John P. Jeter J. Wayne Anderson Bruce D. Shriver

Computer Science Department
University of Southwestern Louisiana
Lafayette, LA 70504

This paper reviews past experiences and discusses future work in the area of interval arithmetic at the University of Southwestern Louisiana(USL). Two versions of interval arithmetic were developed and implemented at USL(*). An interval data type declaration and the necessary mathematical functions for this data type were added to Fortran via the preprocessor Augment(4,5). In the first version, the endpoints of the intervals were represented as single percision floating point numbers. In the other version, the endpoints were represented to 56 decimal digits. Production engineering programs were run as benchmarks(8).The accumulation of computational and algorithmic error could be observed as a widening of the intervals. The benchmarks were also run in normal single and double precision arithmetic. In some instances, the result obtained from a single or double precision calculation was not bounded by the corresponding interval result indicating some problem with the algorithm. The widening of an interval does not necessarily indicate a data sensitivity nor error in an algorithm. However, these large intervals can be used as indicator of no problems. As could be expected, the 56-decimal digit precision interval gave better results in terms of smaller intervals due to the increased amount of precision. The obvious problem with this version is that the amount of overhead required for its execution is high.

1.0 Introduction

The floating point number system used on contemporary computers is an approximation to the real number system. In interval arithmetic, a non-representable real number is approximated by an interval consisting of machine representable endpoints which bound the number. Intervals will be regarded as bounds on an exact but unknown real number. This means that if the interval (a,b) is a computer approximation to the exact result x then $a \leq x \leq b$. To obtain the "best" machine representation of the interval, a must be the greatest lower bound for x and b must be the least upper bound for x. In this way the interval (a,b) will be the smallest computer representable interval that contains x.

In order to obtain the smallest computer representable interval for the result of arithmetic operations on intervals, directed roundings on the computer arithmetic 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, the result of a finite precision arithmetic operation does not always produce a machine representable number. In other words, a op b, where a and b are machine representable numbers and op is, in general, one of the machine arithmetic operations, may not be a machine representable number and must be rounded.

Since the exponent range of floating point numbers is bounded, exponent overflow and underflow may occur during an arithmetic operation. If underflow occurs, then the true result is between zero and the smallest positive or negative representable number. In the case of underflow, a directed rounding may 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 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, (a,b) by defining

F([a,b] = [c,d] such that f(x) is contained in [c,d] for every x in [a,b]

where c and d are machine representable numbers.

When f is evaluated at a point x using a machine representable approximation to x, a computer approximation to f results. This computer approximation F([a,b]) is defined as an interval that contains f(x). If f is monotonic increasing on [a,b], F([a,b]) = [rd(f(a)),ru(f(b))] where rd is such that $rd(f(a)) \leq f(a)$ and ru is such that $ru(f(b)) \geq f(b)$. Ideally, we would like rd(f(a)) to be the largest machine representable number such that $rd(f(a)) \leq f(a)$ (i.e., a greatest lower bound) and ru(f(b)) to be the smallest machine representable number such that $ru(f(b)) \geq f(b)$ (i.e. a least upper bound). Similarly, if f is monotonic decreasing on [a,b], then F([a,b]) = [rd(f(b)), ru(f(a))].

If f is not monotonic on [a,b], then the interval [a,b] can be divided into disjoint subintervals; $[a,b]_i$, i = 1,2,3,...,n; where each a_i and b_i are machine representable numbers and f is monotonic on each subinterval. Further, U $[a,b]_i$ contains

CH1412-6/78/0000-0075$00.75 (c) 1978 IEEE

all machine representable numbers in the interval [a,b] and f is monotonic on each subinterval. It can be shown in this case that $F([a,b]) = \cup F([a_i, b_i])$.

Algorithms for performing the machine arithmetic operations with directed roundings can be found in Yohe (9). 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, interval addition is defined as follows:

$$[a,b] + [c,d] = [rd(a \oplus c), ru(b \oplus d)]$$

where $\oplus$ is the machine addition operation and rd is a downward directed rounding and ru is an upward directed rounding.

It may not be possible to obtain the best bounds for the result of the computer approximation to the function f. An example would be a machine calculation of the sine which is known to be accurate to only 7 digits out of 9.

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 have been 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, MRC, of the University of Wisconsin (2,6,10). A description of the implementation of the MRC interval arithmetic package on the Multics system is given in Appendix A. This appendix is quite lengthy but contains information related to the implementation of mathematical software rarely found in the literature.

3.0 Benchmarks

Several production programs were obtained from the Army Corps of Engineers, Waterways Experiment Station, Vicksburgh, Mississippi, to be run as benchmarks. These programs consisted of four linear equation solvers, a matrix inversion routine, a fast fourier transform routine, a slope stability program and a stress program.

The accumulation of computational and algorithmic error can be seen as a growth in the width of intervals. Wide intervals are not necessarily a sign of data sensitivity or algorithmic error. When a program is run using interval data types, a natural tendency is for intervals to grow wider. However, small intervals are an indication of no problems and wide intervals serve as indicators of possible trouble spots.

During the testing of the initial interval implementation, there were many instances where the intervals became quite large. It was difficult to determine during analysis whether this widening was a problem with the algorithm, an unavoidable result from interval arithmetic, or due to the lack of precision of the representation of the endpoints. 56 decimal digit interval was implemented to help resolve this problem.

3.1 Linear Equation Solvers

Four linear equation solvers were included in the benchmarks supplied by the Army Corps of Engineers. Included was a Gaussian Elimination program. It was first tested on a simple 4 by 4 linear system. Using the standard interval package, the magnitude of the resulting intervals were from 10**-4 to 10**-2. All routines were also run in regular single and double precision. The results obtained by using standard interval insured the correctness of the results only to the third or fourth decimal place. In all instances the intervals bounded the results produced in single and double precision. The same test case was executed using the 56 decimal digit interval package. In this case the width of the intervals varied from 10**-51 to 10**-50. This extra precision obtained from using extended precision interval was obtained at the cost of an increase in cpu time used. The standard interval run required only .44 seconds of processor time while the extended interval required 12.64 seconds. More will be said about the cost of interval and extended interval later.

A second test case, this time a 7 by 7 linear system, was also tried. The standard interval version did not produce any results as the intervals grew too large. However, the extended interval version was able to compute results. The width of the intervals produced varied from 10**-45 to 10**-43.

There were three other equation solvers. The second equation solver, BANSOL, solved banded systems of equations using Gaussain 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 SESOL program solved 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 fourth equation solver was a spline program. It solved a system of linear equations using an interative technique to calculate the moments of a set of data points in order to fit a cubic spline to those data points. In all three cases the results were similar to those above and are discussed in detail in (Private communications with B. Boyt and N. Radhakrishman Army Corps of Engineers, Waterways Experimental Station).

3.2 Matrix Inversion

The matrix inversion program finds the inverse of a squared matrix.The first test case was a Hilbert matrix of order 4. The interval results from the standard interval run were quite wide, from 10**-3 to 0.26. The extended intervals were from 10**-50 to 10**-47. When an attempt was made to invert a Hilbert matrix of order 10, standard interval could not find a solution and the single precision results were erroneous. The extended precision intervals widths ranged from 10**-36 to 10**-28 and again indicated that the double precision results were good to only 8 or 9 digits of precision.

3.3 Fast Fourier Transform

The fast fourier transform (FFT) program supplied by the Army Corps of Engineers proved to be a quite stable algorithm. A difficulty in its implementation in double precision and interval should be mentioned. A FFT program produces complex arithmetic results. Fortran does not normally support double precision complex arithmetic and, therefore, it had to be simulated. The same type of simulation had to be done for interval. This slowed the execution of the algorithm considerably. In all test cases, all arithmetics produced good results. The single precision intervals had a width of on the order of 10**-6 and the extended intervals, 10**-53.

3.4 Slope Stability Program

An application program, SLOPE, was also sent us by the Army Corps of Engineers. Testing using this program consisted of varying a set of three inputs (cohesion, unit weight, and phi) for the program plus or minus ten percent. This resulted in 81 runs for each type of arithmetic.

Two problems arose when implementing the slope program in interval arithmetic. The first resulted from the way in which the interval package evaluates the test value in an arithmetic IF statement. When an arithmetic IF statement is encountered with an interval test value, the interval is converted to real, i.e., the midpoint is taken. In one of the subroutines a particular branch was to be taken only if the test value is 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 and, since the interval contained zero, a zero divide error occured. The solution to this problem was to recode using a logical IF which is evaluted in a different manner and avoids this problem.

The second problem was more difficult to pin down. During testing using standard interval, some of the runs contained intervals which were "blowing up", that is, the width of the intervals were becoming unacceptably large. After a considerable analysis effort, a correlation was uncovered between the large intervals and the -10% value for unit weight. By starting with the initial value for unit weight and decreasing its value in increments of .25%, the initial value at which the intervals blew up could be pinpointed. This occured at about -2.25% of the initial value. As long as unit weight did not go below this value, acceptable results were obtained. After further effort, the problem was traced to a single statement, "T3 = FS1 - FSL". As unit wwight decreased below -2.25% of its original value, values of FSL and FSL became closer and closer together. This subtraction resulted in stripping off the significant digits. T3 was subsequently used as a divisor compounding the effect.

During the procedure of tracking down the error source, a side benefit was reaped which is indicative of the type of recoding of algorithms sometimes necessary to get satisfactory results from limited precision interval arithmetic. Several computations could be combined and an interval consistently of less than optimal width could be factored out producing a more accurate algorithm. The set of runs was repeated using the extended interval package. Most of the data sensitivity noted above disappeared. No interval widths exceeded 10**-4.

3.5 Testing Summary

The 56 decimal digit interval package did prove useful in many cases. Often the standard interval either produced no solution or solutions with extremely wide intervals. Some massaging of the code supplied by the Army Corp of Engineers was required to execute it satisfactorily using interval arithmetic. The primary cost of the use of extended precision interval arithmetic was in terms of central processing time consumed and increased paging activity. On a system like Multics, both of these figures can be perturbed by the load on the system. The figures in Table 3.1 for the FFT routine indicates a general trend. This data was gathered from runs made during a contiguous time interval during a period of low system utilization.

	PAGE FAULTS	CPU TIME(Seconds)
single precision	23	0.3623
double precision	36	0.6678
standard interval	39	14.4994
56 decimal interval	3195	466.8781

Table 3.1
FFT subroutine Overhead

4.0 Conclusions and Future Work

Interval arithmetic can, at times, be extremely useful. For instance, it can be used to indicate the limits of precision of an algorithm for a given set of data. From the testing it was shown that much better bounds on the precision were obtained using the extended interval package. This was, of course, not unexpected. 56 decimal digits carry more precision than 27 binary digits (equivalent to approximately 8 decimal digits) and there is no conversion error on input and output for the 56 decimal interval package. The price paid was in terms of runtime efficiency. Standard precision interval resulted in approximately, at most, an order of magnitude increase in execution time over that of single or double precision arithmetic. 56 decimal interval arithmetic resulted in a further increase of more than one to more than two orders of magnitude. It should be noted here that the 56 digit version was based upon the 59 decimal digit hardware arithmetic unit of the Honeywell H68/80 processor. Extended precision arithmetic using software simulated basic operations could be expected to take much longer.

One obvious application of extended interval arithmetic would be to validate existing programs. Any data sensitivity discovered could be included in a description of the algorithm and directions on its use. Although extended precision interval arithmetic is expensive, its cost must be balanced against possible consequences of using invalid results. An organization like the Corps of Eningeers might weigh a defective dam or the cost of moving 100,000 tons of dirt against the cost of a few hours of computer time.

A more effective technique would be to first test the algorithm using standard precision interval arithmetic. Its relatively small decrease in run time efficiency indicates that its use is more than justified as an economical means of identifying possible trouble areas in an algorithm for the data under consideration. The more expensive extended interval package could be applied to just those cases where possible trouble areas have been identified.

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 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 obtianed, 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 standard 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. A variable precision interval arithmetic package is currently under development at US.

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 for just such requirements. As a side effect, an arithmetic unit that can execute variable precision interval arithmetic can also execute traditional variable precision floating point 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. The algorithm, then, could be executed using only the required precision.

A.0 The Implementation of the MRC Interval Arithmetic Package for the Multics System

In the Multics implementation, 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.

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. Several of the Fortran subroutines call routines that are written in PL/1. The PL/1 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/1.

The routines which perform the four basic arithmetic operations of addition, subtraction, multiplication, and division on interval numbers are machine dependent. 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 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 routimes perform the "best possible arithmetic" computer operations with directed roundings as described by Yohe (9). All the routines are written in PL/1 for the Multics system.

A.1 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. 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 it does not necessarily produce 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.

A.1.1 Accuracy Testing

To our knowledge, there is no documentation concerning the implementation of the basic external functions on Multics used by PL/1 and Fortran. Therefore, the accuracy of these functions had to be determined. Three approaches were considered for use in determining the accuracy of the required external functions:

1) rigorous error analysis of actual implementations
2) rewriting of the algorithms
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 (most of which was written in ALM, the 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 values in published tables (1)

"Driver" programs were written which generated test data and called the routines which were to be tested. 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 tabular digits (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 reason to believe that accuracy is much greater than that.

A.1.2 Error Conditions

The Univac 1108 double precision floating point number 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. In Multics, both single precision and double precision floating point numbers have an 8-bit exponent field. Therefore, the check for eventual fault conditions had to be made prior to the calls to the double precision functions.

Overflow could be produced by the following functions: exp, sinh, cosh and tanh. In the Multics implementation of interval arithmetic, overflow in these functions was prohibited by restricting the domain of the arguments to the interval (-88.029, 88.028). Should an argument fall outside this domain, special actions (described later) had to be taken. Restricting arguments to this domain prevented overflow from occuring during the evaluation of the functions. However, the magnitude of the endpoints of the results were always much smaller than the largest representable number. This implies that the domain of the arguments should be extended.

A.2 Input/Output Routines

The I/O routines implemented on Multics were designed to some extent after the I/O routines implemented for the UNIVAC 1108 version of the interval package (4). Additional routines were included in the Multics version to handle scalar interval variables and a matrix of interval variables

A.3 56 Decimal Digit Interval Implementation

A 56 decimal digit version of the original Multics interval package has also been implemented on the Multics system. This version uses the decimal arithmetic hardware available on the Honeywell H68/80 processor. The decimal arithmetic unit performs both fixed and floating point 59-decimal arithmetic. Fixed decimal arithmetic was used to implement the decimal interval package. Floating point decimal arithmetic was not used due to 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 endpoints 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 because in the implementation of the BPA 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. In this way the logic of the original interval package was used and the number of errors encountered in the implementation could be reduced. The entire 56 decimal digit interval package was written in PL/1 as Fortran does not support decimal arithmetic. Only the number of words required to carry the PL/1 representation of the interval was declared in Fortran. The Fortran routines would carry the interval to be passed to the PL/1 routines.

The first step in the implementation of the 56 decimal digit interval package was the implementation of the best possible arithmetic, BPA, routines (see section 1.0 of the attached paper). The existing procedures for doing BPA for the original interval package were modified to perform 56 decimal version. In the single precision interval package the implementation of the I/O routines proved to be one of the most difficult tasks. This was due to the required conversions between floating decimal and floating binary. The correct roundings had to be done for the conversions in either direction and the algorithms for the conversions became rather involved. The implementation of the 56 decimal digit interval I/O presented no such problems as the internal representation of the interval was already in decimal. The only rounding done is on output when the user requests less than 56 decimal digits of precision.

In the initial interval effort, 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, the Fortran Multiple Precision Package, MPP, developed by Brent (3) was used. The values produced by the basic external functions could be calculated to an arbitrary precision using MPP. It was necessary to construct an interface between Brent's routines written in Fortran and the interval package written in PL/1. The implementation of the SIN and COS routines presented an especially difficult implementation problem. The arguments had to be reduced to a value 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, 5pi/2, 3pi and 7pi/2. These constants had to computed using the MPP.

References

(1) Abramovitz, M. and Stegun, I.A., (ed.), Handbook of Mathematical Functions, National Bureau of Standard Applied Mathematics Series, June, 1974.

(2) Binstock, W., Hawkes, J. and Hsu, N., "An interval input/output package for the UNIVAC 1108," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1212, September, 1973.

(3) Brent, R.P., "A fortran multiple-precision arithmetic package," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May, 1976.

(4) Crary, F.D., "The AUGMENT precompiler, I. User information," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1469, December, 1974.

(5) Crary, F.D., "The AUGMENT precompiler, II. Technical documentation," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1470, October, 1975.

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

(7) Moore, R. E., Interval Analysis, Prentice-Hall Inc., Englewood Cliffs, N.J., 1966.

(8) Reuter, E.K. and Podlaska-Lando, S., "Source Listing for the MULTICS Interval Arithmetic Package," Computer Science Department Report No. 76-7-3, University of Southwestern Louisiana, Lafayette, Louisiana, August, 1976.

(9) Yohe, J.M., "Best possible floating point arithmetic," The University of Wisconsin, Mathematics Research Center, Technical Summary Report No. 1054, March, 1970.

(10) Yohe, J.M., "Software for interval arithmetic: reasonably portable package," Transactions on Mathematical Software, to be published.