THE DESIGN AND USE OF A FLOATING-POINT (SOFTWARE) SIMULATOR FOR TESTING THE ARITHMETIC BEHAVIOR OF MATHEMATICAL SOFTWARE

Myron Ginsberg
Department of Computer Science
Southern Methodist University
Dallas, Texas 75275

Dennis J. Frailey
Department of Computer Science
Southern Methodist University
Dallas, Texas 75275

Summary

An important aspect of any evaluative procedure for developing high quality mathematical software is testing the effects of arithmetic behavior on algorithmic implementations. This paper describes a proposed design approach and various applications of a high-level language floating-point simulator which has two inputs: the program to be tested and a description of the floating-point arithmetic under which the routine is to be executed. A brief discussion of the motivation for this approach is given along with a review of existing efforts to study the influences of computer arithmetic on the accuracy and reliability of mathematical software. An overview of the simulator's structure is presented as well as suggestions for experiments to assist in determining the effects of floating-point behavior across several different computer architectures. Present and future uses of the simulator are also indicated.

1. Introduction

An important aspect of any comprehensive evaluation of mathematical software is testing the effects of computer arithmetic on high-level language implementations of numerical algorithms. Each computer has its own floating-point number representation; its exact form and use in arithmetic operations during program execution influence the numerical accuracy (and hence reliability) of the tested routine. Mathematical software written in FORTRAN or ALGOL exhibit varying degrees of accuracy across diverse computer architectures of the same or different manufacturers. Therefore, during both the developmental and maintenance phases for math routines the effects of accuracy degradation must be observed for a wide variety of existing and future computers in order to make necessary software adjustments in particular program versions.

We are in the initial stage of development for a high-level language system to assist in observing computer arithmetic behavior. The system has two inputs: the routine to be tested and a description of the floating-point arithmetic under which the tested routine is to be executed. Then the accuracy performance of the program under those conditions can be simulated. The resulting error propagation can be observed and compared with output obtained under other restrictions. Such a system permits simulation and analysis of past, present, and future computerized floating-point arithmetic on existing and proposed mathematical software. The simulator also provides multiple precision capability.

In Section 2 we indicate the need for such a testing facility and in Section 3 we give a brief discussion of computer arithmetic features whose influences should be considered. Section 4 reviews attempts to provide accuracy measures for scientific computations. The objectives and criteria for our system are outlined in Section 5 and an overview of the simulator's structure is given in Section 6. Section 7 is devoted to defining experiments for use with the simulator to check out a program's arithmetic behavior. Potential extensions and uses of our approach are mentioned in Section 8. Some open questions are suggested in Section 9 and the paper concludes with a brief summary in Section 10.

2. Need for Comprehensive Accuracy Testing

Highly reliable math software is a necessity for the complex problems encountered in modern science and engineering. Unfortunately, the rapid advance of computer technology coupled with the growing needs of the scientific community have resulted in the production of a large quantity of low quality software; many of the routines have poor or undefined accuracy requirements and ignore the idiosyncrasies of computer arithmetic. Furthermore, there has been a meager amount of effective testing of algorithmic implementation behavior under a wide variety of conditions. A host of computer manufacturers and user groups as well as individual companies and government laboratories have built large and small mathematical libraries with the above-mentioned inadequacies and have all too often repeated each other's shortcomings.

Costs have been high. For example, it has been estimated that the Boeing subroutine library costs \$14.58 per source card.[7] A set of eigenvalue programs called EISPACK[48] costs about \$20 per source card; this does not include the original developmental costs of the ALGOL programs which served as a starting point for these FORTRAN routines. Program libraries that have met with some initial success have often become outmoded or suffered from numerical accuracy problems because of conversion difficulties as new machines are introduced.

One of the lessons to be learned from this dismal situation is obvious: Techniques must be developed for the construction and effective testing of math software with known accuracy requirements and limitations as well as high reliability over an extensive spectrum of computer arithmetics.

The properties of floating-point arithmetic representations and their intricate relationships with numerical algorithms complicate the task of assessing a mathematical routine's accuracy. Some of the basic difficulties are discussed in the next section.

3. Computer Arithmetic Influences

Most high-level languages permit input of numbers in base 10 scientific notation. A conversion is usually performed to an internal representation in some other number base. Most machines use base 16 (e.g., IBM Systems/360 and 370), base 8 (e.g., Burroughs 5500) or base 2 (e.g., CDC 6000-7000 Series). These differences contribute to numerical inaccuracy problems exclusive of any error induced by the arithmetic in the program.

Floating-point numbers which can be represented exactly in base 10 do not necessarily have an exact representation in the number base and format of a computer's internal representation. Furthermore, the amount of conversion error is dependent on the internal number base as well as on the tradeoff between the number of bits assigned to the characteristic and mantissa. Thus, base 10 floating-point data accepted by a high-level language program might exhibit a substantial variation in internal accuracy when the routine is executed on several computers which use different number bases, word lengths, or a variety of lengths for characteristic and mantissa representations. These inaccuracies occur both when inputting and outputting data. A formal analysis of such conversion problems has appeared in the literature.[35,36]

Another accuracy problem arises in connection with the use of normalization, rounding, and truncation operations. In the basic arithmetic operations, the operands are usually normalized before the operation is executed and the resultant is also normalized. Since the resultant is often computed to more digits than can be stored in the internal floating-point representation, a rounding or truncation of the answer must be performed along with normalization. If the rounding or chopping precedes the normalization process (as it does on some computers such as the CDC 3300), then there can exist some floating-point numbers x such that x multiplied by 1 does not equal x; thus, on such machines there is no multiplicative identity for all representable numbers.[8] Most of the time the user does not have the choice between rounded and truncated arithmetic. The standard default is usually truncated arithmetic which tends to increase inaccuracy. The shifting of mantissa digits such as is done in floating-point addition can result in situations where $x + y = x$ for $y < < x$, assuming x and y are nonzero floating-point numbers.[24]

To complicate matters further, the associative, closure, and distributive laws of the real line do not necessarily hold for all cases involving computer arithmetic.[51] The computer number representations are not uniformly distributed over the real line, thus preventing several resultants of the same type of arithmetic operation from being represented with the same accuracy. Also some computers (such as the IBM System/360) use "guard" digits to assist in slowing down accuracy degradation; such a facility can introduce additional arithmetic peculiarities.

The influences described above are interrelated in a very complicated manner which is not completely understood in theory and which is difficult to monitor by isolating one or more effects. There have been a few attempts to formally analyze computer arithmetic[35,37,51,57] but often times the theoretical assumptions do not hold in practice or are not easily verified. Some empirical studies (a few of a statistical nature) have been performed to observe the effect of one or two specific influences[4,10,27,34] but have not attempted to generalize their findings nor to include all the influences discussed above.

A computer arithmetic simulation package can assist in observing the interactive effects of all the aforementioned features on algorithmic implementations. Such a system should provide as input a description of the floating-point internal format (number base, characteristic size and biases, mantissa size) as well as indications of when pre- and post-normalization of operands and resultants for each arithmetic operation occur and when rounding or truncation is performed. The presence of guard digits must also be noted along with rules for shifting mantissa digits during arithmetic operations. Any other specialized actions which occur during arithmetic operations must be clearly defined.

4. Attempts to Measure Numerical Accuracy

There are several different approaches reported in the literature for monitoring of error propagation in scientific computations. The existing methods can be divided up into two categories: error-bounding techniques and multiple precision packages. We shall briefly indicate the shortcomings in both areas.

Most error-bounding methods can be applied only to relatively restrictive classes of problems and often involve extensive computational overhead. It is difficult for such schemes to accurately assess all computer arithmetic influences and their effect on computing the actual error bounds. The results are often unreliable or very conservative and usually require extensive execution time for non-trivial computations. One of the most well-know approaches in this category is interval analysis[2,44] and its variations. Unfortunately, it suffers from many of the above-mentioned weaknesses; also its computer implementations are not widely distributed and many existing versions are somewhat dependent on the machine architecture for which they are designed. Further details and references concerning error-bounding methods and their features are reported in the literature.[1,2,5,18,20,44] Most of the suggested approaches in their present forms do not look too promising for efficiently studying the numerical reliability of a wide range of math software; it should, however, be pointed out that scattered throughout the literature are some interesting ideas which warrant further investigation.[5,6,24,42,47]

The second category of available methods offers many different approaches for developing an extended precision arithmetic capability. The main premise of these efforts is that increasing the precision under which a mathematical routine is executed can lead to extremely accurate results which, in turn, can be used to judge the accuracy of a program's single and double precision performance.[23] Most multiple precision packages are very slow. It has been estimated[14] that execution time increases linearly with precision for addition and quadratically for multiplication; time increases are far worse for extended precision functions evaluations.[14] Several of the existing implementations are dependent on the assembly language or the arithmetic unit design of a particular machine, are restricted to programs written in a single language, or have required the use of precompilers; examples of such packages are described in several papers.[3,11,12,13,21,22,23,25,26,27,29,30,39,43,45,46,50,55,56] Thus much of the work which has been done is slow in execution as well as compiler-language - and/or machine-dependent.

These systems have not usually been designed to promote portability for an extensive collection of computers.

There have been two attempts[38,56] known to us to produce portable FORTRAN extended precision systems and one attempt[23] is currently under way to create an ALGOL facility. The package constructed by Wyatt[56] and colleagues is the most extensively developed of these. It uses a precompiler to scan certain super precision data types. The system includes a FORTRAN extended precision library of standard functions and permits multiple precision calculations in bases 2 to 16. It does not permit multiple precision operations on non-FORTRAN programs nor does it permit complete user flexibility in the areas of floating-point representation and arithmetic activities. For example, the package would not easily allow the user to define his own (nonstandard) rules for rounding or chopping (and where they are to be performed) nor would it permit dynamic changes of precision throughout the program (although it does permit mixed mode operations involving extended precision quantities). These are not serious drawbacks for the intended purpose of this package - to provide a portable extended precision capability for scientific computations. These restrictions are, however, somewhat irksome for persons wanting to have more flexible control of floating-point representations and computer arithmetic behavior in order to better observe algorithmic implementation aberrations.

5. System Design Objectives

Our primary goal is to construct a floating-point arithmetic simulation facility which has the potential capability of inputting test programs written in one or more high-level languages and permits users to define their own floating-point representations and the details of all arithmetic operations or to use defaults to certain options and formats. Such a system would permit accuracy testing of new and old algorithmic implementations over an extensive range of existing or proposed computer arithmetic designs. We look upon this package as a research tool for the study of arithmetic influences on computerized numerical algorithms; it will also minimize restrictions on user specification of floating-point representations and associated operations.

A secondary objective is to use the system to offer extended precision arithmetic in one or more high-level languages. In this connection, we hope to benefit from some of the multiple precision schemes referred to in Section 4 and, if possible, to reduce execution times. Such a reduction may be realized by invoking specialized, efficient routines for low and/or often used precisions (e.g. double and triple precision) rather than applying the generalized routines which are employed for higher precisions or for simulation of nonstandard arithmetics.[23]

We want to minimize necessary program modifications for the routines to be tested. We therefore avoid use of new data types (which would require the creation of a precompiler) and explicit function calls to simulator routines in the source program. Instead, the user would insert a few control statements into his high-level source program along with descriptions of how to execute the basic arithmetic operations (initially, in the form of subroutines to be used by the system, later, in a special language developed for this purpose).

The user should be able to work with almost any floating-point form and arithmetic that he can explicitly define, regardless of how nonstandard or unconventional his design may be. In this connection, one of our long-range goals is to develop a generalized, "canonical" floating-point model (internal to package) in terms of certain "primitives" which define the floating-point arithmetic to be used in a simulation and into which the user-defined specifications would be translated; much research is yet to be done on developing such a viable model. It would also be advisable to use the aforementioned "primitives: to create a library of standard support routines.

Of course, as our system is created it should undergo testing with regard to efficiency, accuracy, and portability considerations. Simulations of arithmetic associated with existing machines can be compared with actual results run on those computers. The simulator can be used on several machines to determine the extent of portability and to suggest design changes to increase the possible class of host machines. Performance studies and analysis should be devised to assess efficiency and accuracy of simulated results. In addition, a collection of experiments should be developed to thoroughly test the arithmetic behavior of algorithmic implementations inputted to our system.

6. System Design Overview

The system is to be written as a compiler in ANSI standard FORTRAN and is composed of the following components:

Compiler "front end"
Compiler "back end"
Conversion Package
Simulation Routines

The Compiler "front end" is a lexical and syntactic scanner for the language being compiled. Separate "front ends" can be supplied for different languages in a manner which has been used successfully in several systems.[15,16,52] The purpose of the "front end" is to convert the high-level input language into a "virtual machine" language or UNCOL.[49]

UNCOL (UNiversal Computer-Oriented Language) is an intermediate level language between assembly and procedure-oriented languages; it preserves the relative machine independence of high-level languages such as FORTRAN or ALGOL while reducing language dependency by more explicit specification of the control structure and other characteristics which are unique to a given language. For example, a sequence of FORTRAN statements and a comparable sequence of ALGOL statements may have slightly different effects. When expressed in UNCOL, such differences are made more explicit. Although the UNCOL statements are assembly - like, they contain no explicit machine dependencies. The "virtual machine" whose language is UNCOL is one which represents all machines. Its programs may be translated for any given machine via a suitable "assembler".

UNCOL systems have had less success than originally hoped for, due to such factors as the difficulty of generating efficient code while retaining machine independence; however, compilers based on modifications[15,16,52] of this concept have successfully managed to generate correct code for a variety of languages and machines. Since languages of interest to this project are procedure oriented, scientific ones as opposed to general purpose data processing languages, it is reasonable to expect that a set of UNCOL statements could be developed which would handle this class with acceptable efficiency. It may be necessary to limit which non-numeric features are used in certain general-purpose languages, but this is mainly in the interest of practicality rather than feasibility.

The Compiler "back end" will convert the intermediate UNCOL form into true machine language. This involves three phases:

Phase I:	Storage Allocator and Information Collector
Phase II:	(optional) Code Optimizer
Phase III:	Code Generator

Only Phase III will be highly machine dependent.

This conversion to machine language has the following differences from a normal compilation:

i) The amount of space allocated for data (variables, constants, arrays) will be a function of the kind of arithmetic being performed. Thus, if REAL were to mean 50 digit interval arithmetic, each datum of type REAL would be allocated sufficient storage to handle the two interval endpoints with this much accuracy.

ii) Conversion of constants to internal form will be handled according to the type of arithmetic under investigation; thus the compiler will call the Conversion Package for constant conversion.

iii) Run-time code will call the Simulation Routines for all floating-point arithmetic and the Conversion Package for I/O conversion.

The Conversion Package will have the task of converting data between external forms and the internal form necessary to simulate the kind of arithmetic under consideration. It will be called from the Compiler "back end" as well as from the run-time simulation and I/O routines.

The set of Simulation Routines is, of course, the most important component of our system. Simulation will be required for all primitive operations (e.g. add, multiply, divide, negate, absolute value, exponentiation, etc.). Higher level routines such as library support functions (e.g. sin, cos, etc.) will be written in the UNCOL language of the compiler, in which the only floating-point arithmetic is that represented by the primitive operations. Essentially, the virtual machine's floating-point "commands" are to be simulated according to the type of arithmetic under examination. All non-primitive arithmetic routines are to be written in terms of these "commands", hopefully in such a way that the details of the format, type of arithmetic, etc. can remain unspecified.

We envision two phases for the development of the set of Simulation Routines. In the first phase, primarily intended to get the system "on the air", the primitive operations will be written separately for each type of arithmetic of interest. In the second phase, a more general set will be developed using a process in which an arithmetic format is described as input and the primitive routines are produced as output with the intermediate form involving the floating-point canonical representation mentioned in Section 5. This phase requires much additional investigation.

We shall attempt to take into account portability considerations in our system design, i.e. we want our system to be easily modifiable so as to run on several computers. This is our reason for writing almost everything in our system in ANSI standard FORTRAN; however, since the output of our compiler will be machine (or perhaps, assembly) language, it is not possible to change machines without some program modification. The goal is to keep machine dependencies isolated to Phase III of the "back end" so as to limit the amount of necessary reprogramming. To make the conversion and simulation components portable, they will be written in FORTRAN or, if necessary, in the same UNCOL language generated by the compiler "front end", except that conversion routines and run-time support will probably be prone to dependence on the object machine and the type of arithmetic being considered.

Our approach differs from those employed by most existing multiple precision systems referred to in Section 4 in that our proposed package uses a compiler rather than a pre-compiler or a hand-coded "front end"; most existing systems require either that a superset of FORTRAN or ALGOL or else that hand-coded modifications (subroutine calls or special statements) be written. With our system, commonly used programs can be tested to see how well they will work under different arithmetic conditions without any internal modifications or insertion of special statements, except at the beginning (or via "job control language"). As an option, we will permit the use of some "nonstandard" statements; for example, optional directives would be allowed which indicate that extra precision should be used in certain portions of the program. In either case, our proposed simulator will use as input high-level language programs which do not require the extensive revisions demanded by present systems.

Thus, our system will accept existing programs (or slight modifications of the original), "compile" them, and execute them using a simulated version of an arithmetic under investigation. To study a new type, it will sometimes only be necessary to supply a new set of simulation subroutines (or later, a description of a new type of arithmetic). For an arithmetic which is substantially different from those previously studied, viz. requiring a different internal representation of floating-point numbers, a "recompilation" would also be required. In no case, however, would program modification be necessary. Our approach will therefore minimize restrictions on user-defined innovative or unconventional floating-point representations and their associated arithmetic. Thus, numerical behavior of mathematical software could be observed over a broad spectrum of existing or proposed computational environments.

Other advantages of our compiler approach include the following points:

. more control over code generation and optimization, thus potentially higher efficiency in object code

. wide flexibility in translation (we would not be restricted to the use of subroutine and function calls to handle features not found in the original language)

. "natural" expression of constants and expressions, since we control the translation.

7. Experiments for Arithmetic Behavior

The types of tests we discuss here are independent of our system's internal design; they only assume the existence of a floating-point arithmetic facility (simulated or real). We could simply run a set of high-level language programs with our simulator and observe the numerical results; however, to provide more comprehensive and systematic testing as well as to check basic arithmetic properties, we are advocating additional checks. The experiments can be categorized into the following areas: program data variation; computer arithmetic properties; variable precision tradeoffs; automated checks.

A naive approach to testing the effects of data variation is to use a large collection of random numbers as data. Unfortunately, this is not a foolproof scheme for spotting algorithmic or program weaknesses[9,31]; often times, arguments which result in substantial inaccuracy are very dependent on a computer's arithmetic unit, or its floating-point policies. Although some poor results may be discovered by using extremely large sets of random data, this approach seems to be somewhat inefficient since no assurance can be given that most or all computer hardware and software numerical anomalies will make their presence known. Instead, what is needed is a workable set of data which thoroughly exercises the entire range of applicability of a mathematical routine. Selected bit configurations which may cause difficulty should be used along with random data that is uniformly or exponentially distributed over the entire range of the simulated floating-point representation.[9,31]

Several theorems are given by Sterbenz[51] to define properties of computer arithmetic. These can serve as a starting point for checking out simulated floating-point behavior. Some of the conclusions of these theorems may not hold for a specific, implemented computer arithmetic because hardware and/or software influences may have violated one or more of their hypotheses. The theorems and proofs suggest ways to determine specific representations which may cause trouble. Additional tests[24] used to check out existing computers would also be helpful, especially if we are studying a simulated arithmetic which resembles that of a specific computer.

Experiments can be devised to observe the accuracy tradeoffs introduced by multiple precision computations for the simulated arithmetics. For example, calculations can be performed in single, double, triple, and quadruple precision to determine the accuracy variation as the precision is changed for a given simulated arithmetic applied to a specific algorithmic implementation. The effects of subtractive cancellation can be observed across several different precisions for the user-defined arithmetic. Tests involving precision variation can also be performed in connection with algorithmic modifications. Multiple precision resultants can further be utilized to assist in assessing the accuracy obtained from other numerical experiments.

There are a few promising techniques for automatic verification of accuracy and stability properties of computerized numerical methods. An automatic a posteriori roundoff error bounding scheme has been proposed by Richman.[47] An automatic a priori roundoff analysis a la Wilkinson is currently being developed.[40,41,42] Such approaches could greatly assist us in evaluating results from our simulator.

As our project progresses, additional experiments will be devised to determine computer arithmetic behavior on algorithmic implementations.

8. Potential Applications

Our primary motivation is to provide a flexible and practical tool to assist in the investigation of floating-point computer arithmetic influences on scientific computations. Such a facility is useful in academic and industrial environments. In both areas, users can develop an intuitive feel as to how various programming techniques and modifications affect the accuracy of a numerical method executed under a given arithmetic. In an academic setting, students and researchers could test out existing or newly-proposed arithmetic procedures and observe the resulting numerical effects on various mathematical routines. In industrial situations, the simulation capability could be used to emulate the arithmetic behavior of a new computer system or of a machine which is not readily accessible.

Another important application area for our package is the development and documentation of portable mathematical software. During the past few years a few groups have attempted to produce high quality math software:
NAG - Numerical Algorithms Group (in England);
NATS - National Activity to Test Software (in U.S.);
IMSL - International Mathematical and Statistical Libraries (private U.S. company). Comprehensive numerical testing methods are needed by these groups as well as any others who want to develop or evaluate general purpose mathematical software. Our proposed simulator could assist in numerical behavior profiles for assessing accuracy variation over a myriad of arithmetic environments. Some possible testing strategies have been outlined in Section 7.

The removable "front end" structure for our system (see Section 6) facilitates the evaluation of programs written in several languages; this is particularly helpful in judging a large collection of available routines since the Europeans prefer to write in ALGOL and the Americans in FORTRAN. Such flexibility also allows us to compare the numerical behavior of algorithmic implementations in different languages.

Another important use for our package is its extended precision capability. There are several practical applications:[23,53,54,56] to test the accuracy of math routines as well as to improve the quality of these programs; to precisely accumulate scalar products and to accurately compute the residuals in iterative refinement; to measure, trace, and control round-off error effects; to improve computation of multiple roots associated with nonlinear equations; to improve the solution of ill-conditioned polynomial equations; to retard the effect of subtractive cancellation; to observe effects on program behavior caused by perturbation of data; to alleviate any other poor numerical situations rather than resorting to specialized tricks.

9. Open Questions and Future Plans

There are several design alternatives which must be given careful consideration as we proceed with our project. A decision has to be made concerning the manner of implementation for library support functions. Many of the multiple precision packages referenced in Section 4 have very few, if any, such facilities; an exception is the National Bureau of Standards' system[56] which contains a substantial set of extended precision library routines. It is not generally feasible when simulating the arithmetic behavior of an existing computer on another machine to use the subroutine library of the simulated machine because such libraries are usually written in assembly or machine language and/or are not portable; furthermore, even hand-coding of such routines into a high-level language version may be very tedious or time-consuming and may not be easily accomplished if the code and/or algorithms are too severely machine dependent. On the other hand, using a library other than the one used on the simulated machine may produce significantly different results than would have been produced by the actual machine (possibly because different algorithms are used to implement the same function in the two libraries). We plan to check out some such possible variations by comparison of our simulation results with output from the actual machines being simulated. Despite this situation, it seems reasonable to use one standard, comprehensive set of library routines for all simulations. We prefer to write such a library in the "virtual machine" language of our system (mentioned in Section 6) in order to promote flexibility and portability; with such a design, floating-point arithmetic is represented exclusively by primitive functions. We are planning to investigate the possibility of modifying in this manner the National Bureau of Standards' library.

Another design problem is the input specification of floating-point representations and behavior. A very restrictive approach may be easy to implement but, at the same time, severely limit the user's possible range of formats and arithmetic descriptions. At first we will adopt a restrictive approach but later on we hope to develop a very general "canonical" floating-point format and description model into which the user specification is converted; this model would be expressed in terms of certain "primitives". Extensive experimentation must be conducted to determine viable candidates for such a model and to indicate what limitations they invoke on user specifications.

We plan to eventually implement FORTRAN and ALGOL removable "front ends" for our system. With a facility to input programs in both languages, we would be able to use our simulator with most of the mathematical software produced in the U.S. and Europe.

The extended precision capability involves a tradeoff between efficiency and portability. We plan to conduct comparison tests among existing multiple precision packages to gain better insight into this problem area and to search for the best algorithms our system should employ when used strictly for extended precision computations.

Additional future activities include extensive testing of the developing system on a variety of math software routines. We also intend to expand and refine the types of floating-point behavior experiments suggested in Section 7.

10. Summary and Conclusions

The production of a large quantity of low quality and somewhat unreliable mathematical software has precipitated the need to develop comprehensive software evaluation techniques. This need is especially important for the production of widely distributed portable routines and for the assessment of the floating-point arithmetic behavior influences on the numerical accuracy of various algorithmic implementations.

We are advocating the development of a high-level language floating-point simulator to assist in producing accuracy profiles of an algorithm's behavior when executed under diverse arithmetic specifications defined by the user. Such a tool can and in determining the numerical effects of existing and proposed floating-point formats and arithmetic policies and also provide a multiple precision capability. Our proposed system will attempt to permit a wide range of user-defined formats and arithmetic specifications and to offer a facility for evaluating programs written in either FORTRAN or ALGOL (and eventually, other languages). The potential uses of such a package include support of reliability testing and documentation for mathematical software, floating-point arithmetic experimentation or simulation, and variable precision approaches to cope with such difficulties as ill-conditioning, subtractive cancellation, etc.

Somewhat independently of our internal system design, we are also proposing the development of tests for observing computer arithmetic behavior. Some such tests have already been discussed in Section 7; additional experiments should be devised.

Hopefully, our package can become a valuable, practical tool for assessing the numerical reliability of mathematical routines across a diverse spectrum of computational environments.

References

1. Ashenhurst, R. L., "Techniques for Automatic Error Monitoring and Control," Error in Digital Computation, Vol. 1, edited by L. B. Rall, J. Wiley, New York, 1965, pp. 43-59.

2. Bierbaum, F., Intervall - Mathematik. Eine Literaturubersicht, Report No. 74/2, Institut fur Praktische Mathematik, University of Karlsruhe, Karlsruhe, Germany, 1974.

3. Blum, B. I., "An Extended Arithmetic Package," Comm. ACM, Vol. 8, No. 5, May 1965, pp. 318-320.

4. Brent, R. P., "On the Precision Attainable with Various Floating Point Number Systems," IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 601-607.

5. Bright, H. S., "A Proposed Numerical Accuracy Control System," Interactive Systems for Experimental Applied Mathematics, edited by M. Klerer and J. Reinfelds, Academic Press, New York, 1968, pp. 314-334.

6. Chartres, B. A., "Automatic Controlled Precision Calculations," J. Assoc. Comput. Mach., Vol. 13, No. 3, July 1966, pp. 386-403.

7. Cody, W. J., "The Construction of Numerical Subroutine Libraries," SIAM Rev., Vol. 16, No. 1, January 1974, pp. 36-46.

8. Cody, W. J., "The Influence of Machine Design on Numerical Algorithms," Proc., Spring Joint Computer Conference, Vol. 30, AFIPS Press, Montvale, New Jersey, 1967, pp. 305-309.

9. Cody, W. J., "Performance Testing of Function Subroutines," Proc., Spring Joint Computer Conference, Vol. 34, AFIPS Press, Montvale, New Jersey, 1969, pp. 759-763.

10. Cody, W. J., "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic," IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 598-601.

11. Cox, A. G. and H. A. Luther, "A Note on Multiple Precision Arithmetic," Comm. ACM, Vol. 4, No. 8, August 1961, p. 353.

12. Crary, F. D., Language Extensions and Precompilers, Technical Summary Report #1317, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, February 1973.

13. Dekker, T. J., "A Floating-Point Technique for Extending the Available Precision," Numer. Math., Vol. 18, No. 3, 1971, pp. 224-242.

14. Dunham, C. B., "Nonstandard Arithmetic," Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 105-111.

15. Ershov, A. P., "A Multilanguage Programming Service Oriented to Language Description and Universal Optimization Algorithms," Proceedings, IFIP Working Conference on Algol 68 Implementation, Munich, Germany, July 1970, pp. 143-162.

16. Frailey, D. J., A Study of Code Optimization Using a General Purpose Optimizer, Ph.D. Thesis, Department of Computer Science, Purdue University, West Lafayette, Indiana, 1971.

17. Gentleman, W. M. and S. B. Marovich, "More on Algorithms that Reveal Properties of Floating Point Arithmetic Units," Comm. ACM, Vol. 17, No. 5, May 1974, pp. 276-277.

18. Ginsberg, M., A Guide to the Literature of Error Bounding Techniques and Their Applications, report, Department of Computer Science, Southern Methodist University, Dallas, Texas, 1975 (in preparation).

19. Ginsberg, M., "A Guide to the Literature of Modern Numerical Mathematics," Bibliography 36, Computing Reviews, Vol. 16, No. 2, February 1975, pp. 83-97.

20. Ginsberg, M., Introduction to the Study of Algorithms for Computing Upper and Lower Bounds to the Exact Solution of Problems in Numerical Analysis, Tech. Report CP 73024, Department of Computer Science and Operations Research, Southern Methodist University, Dallas, Texas, September 1973.

21. Hill, I. D., "Procedures for the Basic Arithmetical Operations in Multiple-Length Working," Comput. J., Vol. 11, No. 2, August 1968, pp. 232-235.

22. Howell, K. M., "Multiple Precision Arithmetic Techniques," Comput. J., Vol. 9, No. 4, February 1967, pp. 383-387.

23. Hull, T. E. and J. J. Hofbauer, Language Facilities for Multiple Precision Floating Point Computation, with Examples, and the Description of a Preprocessor, Technical Report 63, Department of Computer Science, University of Toronto, Ontario, Canada, February 1974.

24. Kahan, W., Implementation of Algorithms, Parts I and II, Technical Report 20 (or AD-769 124), Department of Computer Science, University of California, Berkeley, 1973.

25. Knuth, D. E., "Multiple-Precision Arithmetic," Section 4.3, The Art of Computer Programming, Vol. 2, Addison-Wesley, Reading, Mass., 1969, pp. 229-280.

26. Krishnamurthy, E. V., "On a Divide-and-Correct Method for Variable Precision Division," Comm. ACM, Vol. 8, No. 3, March 1965, pp. 179-181.

27. Kuki, H. and Cody, W. J., "A Statistical Study of the Accuracy of Floating Point Number Systems," Comm. ACM, Vol. 16, No. 4, April 1973, pp. 223-230.

28. Kuki, H. and J. Ascoly, "Fortran Extended-Precision Library," IBM Systems J., Vol. 10, No. 1, 1971, pp. 39-61.

29. Lawson, C. L., Basic Q-Precision Arithmetic Subroutines Including Input and Output, Technical Memorandum 170, JPL Section 314, Jet Propulsion Laboratory, Pasadena, California, October 1967.

30. Lawson, C. L., Summary of Q-Precision Subroutines as Revised in October 1968, Technical Memorandum 211, JPL Section 314, Jet Propulsion Laboratory, Pasadena, California, January 1969.

31. Lozier, D. W., L. C. Maximon, and W. L. Sadowski, "A Bit Comparison Program for Algorithm Testing," Comput. J., Vol. 16, No. 2, May 1973, pp. 111-117.

32. Lozier, D. W., L. C. Maximon, and W. L. Sadowski, "Performance Testing of a Fortran Library of Mathematical Function Routines - A Case Study in the Application of Testing Techniques," J. Res. Nat. Bur. Standards, Sect. B, Vol. 77B, Nos. 3 and 4, July-December 1973, pp. 101-110.

33. Malcolm, M. A., "Algorithms to Reveal Properties of Floating-Point Arithmetic," Comm. ACM, Vol. 15, No. 11, November 1972, pp. 949-951.

34. Marasa, J. D. and D. W. Matula, "A Simulation Study of Correlated Error Propagation in Various Finite-Precision Arithmetics", IEEE Trans. Computers, Vol. C-22, No. 6, June 1973, pp. 587-597.

35. Matula, D. W., "A Formalization of Floating-Point Numeric Base Conversion," IEEE Trans. Computers, Vol. C-19, No. 8, August 1970, pp. 681-692.

36. Matula, D. W., "In-and-Out Conversions," Comm. ACM, Vol. 11, No. 1, January 1968, pp. 47-50.

37. Matula, D. W., "Towards an Abstract Mathematical Theory of Floating-Point Arithmetic," Proc., Spring Joint Computer Conference, Vol. 34, AFIPS Press, Montvale, New Jersey, 1969, pp. 765-772.

38. Maximon, L. C., Fortran Program for Arbitrary Precision Arithmetic, Report 10563, National Bureau of Standards, Washington, D. C., April 1, 1971.

39. McKenna, J. E., "An APL Floating Point Simulator," Public Library 604 ONE, APL/PIE at SUNY Binghampton, New York, 1973.

40. Miller, W., Automatic Verification of Numerical Stability, report, IBM T. J. Watson Research Center, Yorktown Heights, New York, February 1973.

41. Miller, W., Computer Search for Numerical Instability, report, Computer Science Department, The Pennsylvania State University, University Park, Pensylvania, August 1974.

42. Miller, W., "Software for Roundoff Analysis," ACM TOMS, Vol. 1, No. 2, June 1975, pp. 108-128.

43. Moller, O., "Quasi Double-Precision in Floating-Point Addition," BIT, Vol. 5, No. 1, 1965, pp. 37-50.

44. Moore, R. E., Interval Analysis, Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

45. Pope, D. A. and M. L. Stein, "Multiple Precision Arithmetic," Comm. ACM, Vol. 3, No. 12, December 1960, pp. 652-654.

46. Rabinowitz, P., "Multiple Precision Division," Comm. ACM, Vol. 4, No. 2, February 1961, p. 98.

47. Richman, P. L., "Automatic Error Analysis for Determining Precision," Comm. ACM, Vol. 15, No. 9, September 1972, pp. 813-817.

48. Smith, B. T., J. M. Boyle, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, Matrix Eigensystem Routines - EISPACK Guide, Lecture Notes in Computer Science, Vol. 6, Springer-Verlag, Berlin, 1974.

49. Steel, T. B., Jr., "A First Version of UNCOL," Proc. WJCC, Vol. 19, 1961, pp. 371-378.

50. Stein, M. L., "Divide-and-Correct Methods for Multiple Precision Division," Comm. ACM, Vol. 7, No. 8, August 1964, pp. 472-474.

51. Sterbenz, P. H., Floating-Point Computation, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

52. Strong, J., et al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution," Comm. ACM, Vol. 1, No. 8, August 1958, pp. 12-18.

53. Thacher, H. C., "Making Special Arithmetics Available," Mathematical Software, edited by J. R. Rice, Academic Press, New York, 1971, pp. 113-119.

54. Tienari, M., Varying Length Floating Point Arithmetic: A Necessary Tool for the Numerical Analyst, Technical Report No. 62, Computer Science Department, Stanford University, Stanford, California, 1967.

55. Tienari, M. and V. Suokonautio, "A Set of Procedures Making Real Arithmetic of Unlimited Accuracy Possible within Algol 60," BIT, Vol. 6, No. 4, 1966, pp. 332-338.

56. Wyatt, W. T., Jr., D. W. Lozier, and D. J. Orser, "A Portable Extended Precision Arithmetic Package and Library with Fortran Precompiler," presentation at Mathematical Software II, Purdue University, West Lafayette, Indiana, May 1974.

57. Yohe, J. M., Foundations of Floating Point Computer Arithmetic, MRC Tech Summary Report No. 1302, Mathematics Research Center, University of Wisconsin, Madison, January 1973.