Floating Point and Composite Arithmetics

W.N. Holmes
Department of Computing, University of Tasmania, Launceston, Tasmania 7250, Australia.

(Proceedings of the Eighth Biennial Computational Techniques and Applications Conference, Adelaide, South Australia, September 29 to October 1, 1997)

1. Introduction

The widespread adoption of the IEEE/ISO standard for binary floating point arithmetic [8] has brought a measure of order to the chaos of *technical computation.* Persistent complaints about calculator design [18], spreadsheet implementations, and about computational software generally [6] demonstrate that *general computation* suffers from the lack of similar support. Composite arithmetic, as outlined in a paper [7] recently published by the IEEE, addresses these broader problems.

However, composite arithmetic could also be a basis for improving technical programs. This paper contrasts the use of floating point arithmetic and composite arithmetic in technical computation.

2. Floating Point Representation and Arithmetic

The numbers used in floating point arithmetic are represented in semilogarithmic form by expressing their value as two component numbers—one simple number giving its significant digits and another giving the level of scaling used. The so-called scientific notation, now in popular use, is similar, and in the example 2.73×10^{15} the significant digits (or *significand*) are given as 2.73 and the level of scaling as 15. Here the base of logarithmic scaling is explicitly given as 10, but most uses of semilogarithmic notations allow the base to be implied, as in the so-called e-notation, which would express 2.73×10^{15} as `2.73e15`.

The first computers built after the second World War directly implemented only fixed point arithmetic, usually integer arithmetic, and writings of the time recommended that hand scaling be used for numbers in scientific computation, firstly as more reliable (and more familiar in that era of the ubiquitous slide rule), and secondly so that precision was not sacrificed in storing an explicit scale [14]. However, the convenience of automatic scaling soon led to semilogarithmic representation being used through software subroutine libraries, and the popularity of these libraries soon led to the introduction of circuitry to carry out arithmetic, now somewhat misleadingly called floating point arithmetic, directly on values represented semilogarithmically.

2.1. Parameters The variety of computer architectures used in the 1950s and '60s led naturally to a variety of floating point arithmetics, where the variety was within the three parameters of the design of such arithmetic [15].

Numeric base: values were represented digitally in two ways—binarily and decimally, and both were used in floating point representations. The representational base was usually also the scaling base, but not always.

Range : the range of values that could be represented was determined by the scaling base, together with the number of digits allocated to represent the scaling level—usually called the *exponent*. Negative exponents were sometimes allocated a sign bit, and sometimes kept as complements.

Precision : the number of significant digits allocated to that part of the representation giving the value before scaling—once called the *mantissa* but nowadays called the *significand*—depends on the word size of the computer, less whatever was allocated to the exponent. Again, negative significands were sometimes allocated a sign bit, and sometimes kept as complements.

Eventually, a standard for floating point arithmetic became widely accepted [8], which reduced the variety of floating point arithmetics without removing any of the basic problems of scaling and of arithmetic on scaled values.

2.2. Floating point arithmetic problems The problems of floating point arithmetic are inherent in the very method of representation [4].

Special values: neither logarithmic nor semilogarithmic representation (normalised or not), has a natural place for the common value of zero. Zero, its reciprocal, and its self quotient, need special representation if they are to be represented at all.

Range: although a much greater range of values can be represented in floating point than in fixed point, arithmetic overflow cannot be completely avoided, and the additional problem of underflow is introduced. The arithmetic must provide some way of signalling these exceptions.

Precision: values rarely occur singly, and often congregate in very large collections. Floating point representation must therefore allow the designers of files to be generous or frugal in allowing space for floating point values to be stored, consonant with any constraints arising from need for accuracy. More than one length of representation is usually provided, and this affects the accuracy of the arithmetic.

Accuracy: floating point arithmetic is inherently inexact, both in the conversion of values to and from display form, and through truncation and rounding errors. Greater precision in storage is often needed to maintain an acceptable level of accuracy in arithmetic.

To get around such problems special programming techniques, such as logarithmic arithmetic [3] and multiple precision, are often needed. But the problems are not confined to the arithmetic.

2.3. Programming problems Programming for computation with floating point arithmetic is fraught with decisions that could greatly affect the reliability of any results. Simply dealing with the doubts and uncertainties of the arithmetic greatly complicates the work of the programmer. And often the user doesn't know about the decisions, and blithely accepts erroneous results.

Fix or float ? For what values, and in which computations, should floating point be used, and for what and in which should fixed point be used ?

Short or long ? For what values, and in which computations, should single precision be used, and for what and in which should double be used ? Should a multiple precision subroutine library be resorted to ?

Exceptions ? Is exception handling provided, should it be used, and if so, what handling for which computations ?

Conversion ? Where does a program convert values from one precision to another, or between fixed point and floating point ? How does a compiler convert numeric constants in a source program ? Is the loss of accuracy significant ?

Display ? How is accuracy displayed, if known ? Or brought into a program ? How are exceptions to be displayed ? Special values ?

The possible effects of these decisions are often ignored, but often mentioned in research reports with a touch of exasperation, though the problems have all long been known and many solutions have long been considered for particular aspects [15].

3. Composite Representation and Arithmetic

Composite arithmetic is intended to provide a single solution to many of the problems of floating point arithmetic by providing for a variety of representations which the arithmetic can use to adapt automatically to changes in kinds of values, both those fed into computations and those resulting from them [7]. Composite arithmetic is based on a composition of four formats of value representation within three different forms.

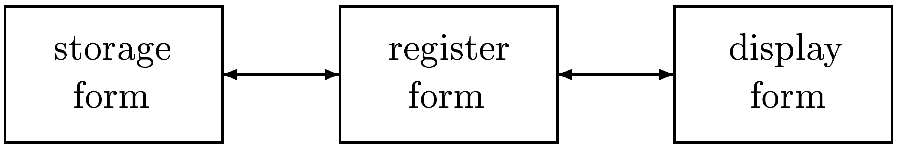
3.1. Composite forms Three composite forms are proposed to meet separately the otherwise conflicting requirements of value storage, value display, and arithmetic.

Storage form strives for efficient storage of values kept within both the main storage, and the secondary storage of computers. As the designer of files would like control over how much storage is taken up by numeric values, subject to any need for particular precision, four different sizes (4 8 16 and 32 bytes) are proposed for storage form, which is binary.

Display form strives for effective communication of information about values between a program and its users. Display form uses character fields whose lengths can be determined by the programmer or the user, and provides for the representation to show not only the value, but also what kind of value it is, and, for an inexact value, what its accuracy is, within the limits of the display field specified.

Register form strives to provide a representation which allows the best feasible arithmetic to be used. Register form is binary and very long—always 512 bytes. Values can be stored in register form, but will take up much more space than even the longest storage form.

The relationship between the three forms is shown in the following diagram. Note particularly that no provision is made for direct conversion between storage form and display form. This means that all conversions are to or from register form so that the best possible conversions are made—those with the least loss of information.



3.2. Composite formats Each composite form has four formats, two for exact values, two for inexact. The specific format to be used for any value is chosen by the arithmetic, not by the programmer. Format changes may be required within register form in the course of a computation, or between forms during form conversion.

Primary exact format represents integers. An exact zero is provided. An integer of more than 500 bytes, binarily represented, can be accommodated in register form, but the value may be too large for either exact, or for primary inexact storage format.

Secondary exact format represents rational values as two integer components [9] though display form will use three, if appropriate. This format represents indeterminacies and exact infinities.

Primary inexact format represents values which are inherently inexact, or cannot be represented exactly, or become inexact through computation. In storage form pure logarithmic representation is used, and in register form fractional is used.

Secondary inexact format represents inexact values which are beyond the representational capacity of the primary inexact format. In storage form antitetrational representation is used, which effectively eliminates overflow and underflow. In register form pure logarithmic representation is used.

Because of space limitations, the original composite arithmetic proposal [7] left out many of the details, in particular those of the means for keeping track of accuracy. Some of those details can be got from Internet in the following documents.

ftp://ftp.comp.utas.edu.au/pub/nholmes/{dssf.ps,ipdf.ps,iprf.ps}

A related document in the same place is *vlcl.ps.* The scheme for accuracy tracking within the inexact representational forms is like that first proposed by Gray and Harrison [5], keeping a central value combined with an index of significance.

3.3. Composite Arithmetic The arithmetic functions of composite arithmetic are defined solely between registers holding values in register form. This form provides extremely high precision within one length of representation, exact arithmetic for exact values, and inexact arithmetic with a significance index for inexact values.

The exact arithmetic is of particularly high precision. The primary exact format provides more than twelve hundred decimal digits. (This format was elided from [7] but is described in the Internet document *iprf.ps* cited above.) The secondary exact format provides more than six hundred decimal digits each for numerator and denominator.

Although the inexact arithmetic is defined on the basis of formats including an index of significance, this provides merely a minimum level of accuracy tracking. Implementations could even provide an interval arithmetic [13] between and within its registers, though the defined format would necessarily be reverted to when values were to be stored in register form.

Primary inexact register format uses a straightforward representation with the bytes split roughly equally between the integer part and the fraction part. The many and signal advantages of this representation for inexact arithmetic are well known as *long accumulator* arithmetic [10].

Secondary inexact register format keeps the logarithm of its value in the same format that the primary uses directly for its value. This does not provide the same freedom from overflow and underflow that the antitetrational representation of secondary inexact storage format provides [11], but it will nevertheless take quite an extraordinary computation to flow out.

3.4. Feasibility Floating point arithmetic has been used in digital computers in one form or other for some fifty years [17], and the reigning standard has been successfully implemented within a single chip.

Obviously, composite arithmetic is more complex than floating point, but many aspects of composite arithmetic have been implemented in hardware and software, though the implementation of composite arithmetic in software would be relatively slow in running [16]. The sheer componentry of modern computer processors [1] strongly suggests that implementation on a single chip would be feasible, provided a suitable standard could be agreed.

Composite arithmetic was originally proposed to support general purpose arithmetic, such as that used by spreadsheet packages and electronic calculators, by defining arithmetic that would give better results, in particular for naive users. But composite arithmetic also has benefits for technical computation beyond simply the better arithmetic—it will make better programs easier to write.

3.5. Program Reliability Programs will of course be more reliable if their final results are better, in particular if some estimation of the accuracy of those results can be given automatically as provided for by composite arithmetic. Most of the techniques within composite arithmetic are already used piecemeal for some technical computation, in particular computational geometry has been supported by both SLI (semiantitetrational) arithmetic [11] and rational arithmetic [12]. But the greatest boon to reliable programming would be any systematic way to allow technical programs to be simpler [19]. This, composite arithmetic also provides.

Fix or float ? The programmer has no choice. The user, or the arithmetic, can choose according to values when the program is run.

Short or long ? The programmer has more choice than at present, but that choice is mainly a matter of file design, not of arithmetic, because the choice only applies to storage form, as the arithmetic being carried out in the sole precision of register form. Multiple precision subroutine libraries would almost never be needed.

Exceptions ? There are no exceptions for storage form, and only extremely unlikely exceptions for register form, and those only exceptions which could be satisfactorily handled by representing the result as an inexact infinity. (Indeterminacy has a representation as well.) Display form would cause exceptions, either on output if the field were shorter than say five characters (which the compiler or interpreter might be expected to prevent), or on input when the error could be corrected by interaction with the user.

Conversion ? Any conversions during computation would be done in register form. Conversions of keyed input would be done straight into register form of the appropriate format carrying exactly the accuracy expressed by the user. Results would be displayed to the user converted from register form with as much accuracy as allowed by the length of the display field provided, and with any inaccuracy explicity displayed. Inaccuracy would either result from inaccurate starting values, from algorithmic inaccuracies, or from inaccuracies introduced when register form values are converted to storage form—this last is the only source of inaccuracy the programmer would normally be concerned with, and this would be a matter of balancing the exigencies of file design against the need for sustaining accuracy.

Display ? The proposal for composite arithmetic provides for explicit display of accuracy, usable for either input or output (see document *ipdf.ps*). Special values, indeterminacy as well as exact and inexact zeroes and infinities, are also provided for in all forms, not just in display form.

With composite arithmetic, the search for good algorithms would be able to focus on the more difficult algebraic problems, because the arithmetic problems would be greatly diminished, if not eliminated. Even some of these algebraic problems might eventually be automatically solved, for example, by the value labelling methods sometimes used by compilers [2].

4. Conclusion

Composite arithmetic provides benefits to technical computing in allowing better final results to be computed with an estimate of accuracy, and in allowing simpler programs to be written. Therefore standardisation of something like composite arithmetic should be supported and promoted actively. The benefits of simpler programs are in the long run even greater than those of better final results.

References

[1] Burger, D. and Goodman, J.R., Billion Transistor Architectures, *IEEE Computer,* **30**, 9, 1997, 46–48.

[2] Briggs, P., Cooper, K.D. and Simpson, L.T., Value numbering, *Software—Practice and Experience,* **27**, 6, 1997, 701–724.

[3] Das, D., Mukhopadhyaya, K. and Sinha, B.P., Implementation of Four Common Functions on an LNS Co-Processor, *IEEE Trans. Computers,* **44**, 1, 1995, 155–161.

[4] Goldberg, D., What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys,* **23**, 1, 1991, 5–48.

[5] Gray, L.H. and Harrison, J.C., Normalized Floating-Point Arithmetic with an Index of Significance, in *Proc. Eastern Joint Comput. Conf.,* AFIPS, **16**, 244–248.

[6] Hatton, L., The T Experiments: Errors in Scientific Software, *IEEE Computational Science & Eng.,* **4**, 2, 1997, 27–38.

[7] Holmes, W.N., Composite arithmetic: Proposal for a new standard, *IEEE Computer,* **30**, 3, 1997, 65–73.

[8] *IEEE Standard for Binary Floating Point Composite arithmetic, ANSI/IEEE Std.754-1985,* IEEE, 1985, reprinted in *ACM SIGPLAN Notices,* **22**, 2, 1987, 9–25 (also called IOS/IEC 559).

[9] Kornerup, P. and Matula, D.W., Finite Precision Rational Arithmetic: An Arithmetic Unit, *IEEE Trans. Computers,* **32**, 4, 1983, 378–388.

[10] Kulisch, U.W. and Miranker, W.L., The Arithmetic of the Digital Computer: A New Approach, *SIAM Review,* **28**, 1, 1986, 1–40.

[11] Lozier, D.W., An underflow-induced graphics failure solved by SLI arithmetic, in *Proc. 11th Symp. Computer Arithmetic,* editors E. Schwartzlander, M.J. Irwin and G. Jullien, IEEE CS Press, 1993, 10–17.

[12] Michelucci, D. and Moreau, J-M., Lazy Arithmetic, *IEEE Trans. Computers,* **46**, 9, 1997, 961–975.

[13] Moore, R.E., *Interval Analysis,* Prentice-Hall, 1966.

[14] National Physical Laboratory, *Modern Computing Methods*, HMSO, second edition, London, 1961.

[15] Sterbenz, P.H., *Floating-Point Computation,* Prentice-Hall, 1974.

[16] Schulte, M.J. and Swartzlander, E.E.Jr., A processor for accurate, self-validating computing, in *Scientific Computing and Validated Numerics,* editors G. Alefeld, A. Frommer and B. Lang, Akademie Verlag, 1996, 25–31.

[17] Rojas, R., Konrad Zuse's Legacy: The Architecture of the Z1 and Z3, *IEEE Annals Hist. Comp.,* **19**, 2, 1997, 5–16.

[18] Thimbleby, H.W., A New Calculator and Why It Is Necessary, *The Computer Journal,* **38**, 6, 1995, 418–433.

[19] Wirth, N., A Plea for Lean Software, *IEEE Computer,* **28**, 2, 1995, 64–68.