Floating-Point from Three Perspectives

An Integrative Paper

Daisy Erin Parker

April 23, 2001

1 Introduction

This paper surveys the following three subject papers from the field of computer science.

- William Kahan's *Mathematics Written in Sand* [4] considers several devices used widely in the early 1980s for numerical computation. Kahan examines the trend of increasingly complicated and mathematically dense circuitry in the silicon chips of these devices and how it impacts the end users.
- Charles Farnum's *Compiler Support for Floating-point Computation* [2] stresses the importance of generating predictable machine-level floating-point operations from high-level language source code and discusses some pitfalls common to past compilers.
- Johnathan Richard Shewchuk's *Robust Adaptive Floating-point Geometric Predicates* [9] introduces fast implementations of four widely-used geometric predicates. The implementation techniques gain speed by exploiting features of the predicates and handling floating-point values efficiently.

Floating-point arithmetic is a thread common to all three of these subject papers. Each paper addresses some of the issues often posed by floating-point representation and arithmetic: the exactness of floating-point numbers, the execution time required by floating-point operations, and the difficulty of developing floating-point programs. This integrative paper emphasizes such floating-point issues at the levels of hardware, compiler, and software.

The remainder of this paper is structured as follows. Section 2 reviews background material on floating-point representation and arithmetic. Section 3 discusses the precision of floating-point numbers. Section 4 considers the relationship between the speed of floating-point operations and the accuracy of their results. Section 5 examines the development of floating-point code. Section 6 presents the IEEE Standard 754 for Binary Floating-Point Arithmetic and conclusions.

2 Background

This section provides some background information on floating-point notation that will be useful throughout the remainder of the paper.

2.1 Purpose of floating-point representation

Before the use of floating-point representation became popular in the mid-1950s, most computers employed a *fixed-point* representation for numbers. As the name fixed-point implies, the *radix point*, which serves as the boundary between the integer portion and fractional portion of a digit string, is fixed for every number represented. Representing a larger range of numbers in fixed-point notation requires increasing the length of the digit string or scaling large numbers into the existing range. Floating-point notation evolved from a need to represent a larger range of numbers than fixed-point notation afforded [1].

2.2 Representation of floating-point numbers

The floating-point representation of a number X consists of four parts. A number X has the magnitude of its *base* number raised to its *exponent* and multiplied by its *significand.* The *sign* bit of X represents the sign of its magnitude.

$$X = base^{exponent} \times significand \times (-1)^{sign}$$

The value of base (typically 2) is implicit in the floating-point hardware. Usually, a *biased exponent* is stored. The biased exponent is the result of adding a quantity equal to half of the exponent's range to the true exponent (*e.g.*, with 2^8 possible exponent values and a true exponent of -74, then the biased exponent is $\frac{1}{2}(256) - 74 = 34$). Use of a biased exponent simplifies the comparison of two floating-point numbers, by converting all exponent values to positive integers. It also defines a unique zero representation, assigning a zero significand and the most negative exponent in biased-exponent form to represent the floating-point value of zero. The bit-representation of a floating-point number typically takes on the following form,

sign	*biased-exponent*	*significand*
0	1 e	e+1 e+s

where the biased exponent is represented using e bits, and the significand using s bits [3, 7].

As the previous paragraph describes it, the floating-point representation of a number is not *unique.* One can certainly conceive of multiple ways to represent the same number (*e.g.*, $10^2 \times 1 \times (-1)^1 = -100 = 10^{-2} \times 10000 \times (-1)^1$). Therefore, rather than allow redundant floating-point representations, *normalization* of the significand defines a unique exponent. Normalization moves the radix point of the significand's digit string to the position just after the most significant nonzero digit, and updates the exponent accordingly. Normalization has two desirable side effects: it preserves the maximum *precision* allowed in the represented number, and it ensures that every number represented will have equal precision. Precision measures the exactness of a number.

Floating-point representation is not *closed.* To exhibit closure with respect to an operation, any two operands exactly representable in floating-point format must produce a result exactly representable in floating-point format. While the rational numbers have closure for addition, subtraction, multiplication and division (except by zero), floating-point numbers certainly do not. There are two reasons that a floating-point number may not be exactly representable — the value of its significand may not be representable with the precision allowed, or the value of its exponent may not fit in range [1].

2.3 Floating-point arithmetic

Floating-point arithmetic approximates the arithmetic of real numbers. It is more complicated than a simple addition, subtraction, multiplication, or division of two integers. To perform floating-point addition or subtraction, one must adjust the operand with the smaller exponent to match the larger exponent before adding or subtracting the significands. To perform floating-point multiplication, one must add the exponents of the operands and multiply their significands. To perform floating-point division, one must subtract the exponent of the divisor from the exponent of the dividend and divide the significands appropriately. At the very least, following these procedures is necessary, already making floating-point arithmetic expensive compared to integer arithmetic.

Moreover, several other floating-point issues are likely to arise and require resolution. The first issue is normalization (see Section 2.2). If an arithmetic result has more than one nonzero digit before the radix point, it is not in normalized form. Such results need an extra normalization step. The second issue is *overflow.* Overflow occurs when the exact result of an operation has an absolute value larger than $N_{\max}$, the largest finite number representable in floating-point format. Typically, situations of overflow are resolved by setting the result to $\pm N_{\max}$ or interrupting/terminating execution. The third issue is *underflow.* Underflow occurs when the exact result of an operation has magnitude less than the smallest number representable in floating-point format. Usually, situations of underflow are resolved by setting the result to zero. The third issue is unrepresentable numbers. Some arithmetic results simply are not exactly representable in floating-point format, and *rounding* such a result resolves this issue. The rounding process chooses a representable floating-point neighbor of a number unrepresentable in floating-point format. Most machines have several modes of rounding, and each mode has a different rule for determining which neighbor to choose. These floating-point issues further complicate the arithmetic and add to its expense [1, 7].

2.4 The inherent problem

There is a great challenge inherent to the task of using a machine to represent and operate on numbers. A finite amount of discrete values simply cannot model a dense system of real numbers accurately. Kahan [4] recognizes that some amount of error is inevitable.

> So, uncompromising adherence to the most rigorous rules for approximate arithmetic will not protect a computer from unpleasant surprises. Apparently the approximation of the continuum by a discrete set must introduce some irreducible quantum of noise into mathematical thought, as well as into computed results, and we don't know how big that quantum is.

Therefore, the true challenge of floating-point representation, or any other system for approximating real numbers, is how to contain a small amount of unavoidable noise and restrain from inducing larger, unnecessary error.

2.5 Levels of abstraction

Blaauw and Brooks [1] describe floating-point notation as an abstraction, "Floating point is one of the first successful steps toward a higher-level language . . . The user thinks in terms of numbers, not of their representations." To the programmer, a floating-point value is just a number, as an integer value is just a number. In software, *programming languages* provide one or more data types for floating-point numbers and overloaded operators on the numbers, according to the data types of the operands. This eliminates the burden on the programmer to add numbers of one data type differently than numbers of a different data type. The *compiler* has the responsibility of determining whether addition means integer addition or floating-point addition, as well as at what precision to perform floating-point addition. Yet even the compiler does not have the onus of manipulating the bit-representations of sign, exponent, and significand. It leaves these details to *hardware*. Abstracting floating-point representation at each of these levels has implications on the precision of floating-point numbers (Section 3), the relationship between the speed of a floating-point program and the accuracy of its results (Section 4), and the ease of developing floating-point programs (Section 5).

3 Precision

Precision of a number indicates the exactness of the quantity, which is often expressed by the number of significant digits. A real number has no limit on the significant digits allowed, and thus, it will always be exact. A machine number has limited precision, and as a result, it may be only an approximation of the value it intends to represent. In particular, limiting the precision of a floating-point number affects the exactness of its significand. Given that we must accept some inaccuracy in floating-point representation due to finite precision, two questions arise:

1. How much precision should a floating-point format allow?
2. If more than one precision is present in the same computation, how should they be allowed to interact?

3.1 Defining precision in hardware

Typically, a machine offers at least two types of precision. It uses a *normal precision* for the initial operands and the final results of arithmetic operations, and a *long precision* for the intermediate results of arithmetic subroutines (composition of several arithmetic operations). As discussed previously, the floating-point representation of a number is potentially an approximation of its actual value. In an arithmetic subroutine that requires several intermediate results, the approximations can compound until the accuracy of the final result has been seriously compromised. Allowing long precision for intermediate results can help prevent such a situation. Consider the following (contrived) example in decimal arithmetic.

Example 1 Let normal precision be 4 digits and let long precision be 6 digits. Compute `x:=a*b*c`, where `a=0.0040`, `b=0.0120` and `c=100.0`.

	Intermediate results have normal precision	Intermediate results have long precision
`temp:=a*b`	`temp:=0.0000`	`temp:=0.0000 48`
`x:=temp*c`	`x:=0.0000`	`x:=0.0048`

The computation requires long precision at the intermediate step to ensure the same accuracy as the result of a normal arithmetic operation.

Machine architectures usually reserve long precisions for *register* locations (but they may be found in other memory locations, as well). The registers make up a small set of memory devoted to temporarily containing operands and results. Registers are quickly addressable and close to the processor.

Kahan [4, pages 9-10] cites the case of digits being dropped prematurely from the right-hand side of a register-resident value as the most common aberration in floating-point arithmetic. Insufficient precision causes this aberration, and in the presence of such disparity, arithmetic properties may not seem to hold. Kahan provides examples on various calculators where multiplication is not commutative or monotonic and the identity property fails. It is important to realize that this aberration can lead to unexpected results on one machine; however, realizing that the phenomenon will vary across machines is far more important. Certainly, on two machines that retain a different number of significant digits the computed result of the same arithmetic operation may differ.

It is difficult to know how much precision is enough. The number of significant digits necessary for one computation will not be adequate for another. In fact, Kahan [4] mentions several circumstances where keeping any reasonable number of significant digits cannot prevent inaccuracies that are destined to occur. Of course, greater precision costs more computation time, so hardware designers must consider the trade-off carefully. Hardware designers should also take care in determining how many more bits are in long precision than in normal precision.

3.2 Compiler evaluation of mixed precision

Many languages support multiple precisions, associating a data type with each. The data type of a value indicates how many bits its representation allows. For illustration, we will consider *single precision* and *double precision*, where double precision has twice as many digits as single. Most languages include at least single and double precisions, despite the fact that their target machines may be limited to a different set of precisions. Since a source language abstracts the target hardware, having both language and hardware support the same precisions is not necessary. A compiler maps source code to machine instructions for the target hardware. However, most language standards do not specify the exact semantics of floating-point operations, since floating-point hardware systems can vary widely. As a result, there is a lot of freedom for the compiler writer to determine the machine-level instructions generated from source-level floating-point operations.

Farnum [2] suggests that compiler writers often abuse such freedom, offering the case of mapping more than one source-language precision to the same machine precision. Promoting all single precision variables to double precision may seem harmless, but it could be the case that the program depends on a double precision variable having twice the precision of a single precision variable (*e.g.*, the product of two single precision variables is exactly representable in double precision). Unless the target machine supports only one precision, even reaching this situation demonstrates an undesirable compiler.

Farnum [2] also presents the problem of multiple precisions in a single statement. Consider the following example, which is not language-specific.

Example 2 Let `s` denote a single precision variable and let `d` denote a double precision variable. Then, the statement `d:=d+s*s` includes both single and double precisions.

The difficulty with Example 2 is deciding at what precision to evaluate the subexpression `s*s`. As before, the compiler writer has the freedom to make this determination. Farnum cautions that the choice be considered carefully, and discusses the following alternatives.

Strict evaluation uses a precision only as large as the largest operand (evaluating `s*s` with single precision). In Example 2, the error induced by rounding `s*s` to single precision will taint the extra precision of `d`.

Widest available uses the widest precision available (evaluating `s*s` with double precision). It is difficult to anticipate when the extra precision will be worth the increased computation time, as it happens to be for Example 2. In some cases, the expense will be wasted.

Widest needed assigns precisions via an expression tree (also evaluating `s*s` with double precision). Tentative precisions are assigned in a bottom-up traversal using strict evaluation. Then, a top-down traversal assigns a subexpression the wider of its tentative precision and the precision expected by its parent. Although this strategy is not as simple as the other two alternatives and requires more effort from the compiler writer, Farnum prefers widest needed precision because it retains the usefulness of overloaded operators without wasting extra computation.

Of course, a programmer can force the compiler to make the appropriate decisions by employing explicit type conversions, but this leads to increased programmer effort and cluttered code (see Section 5.2).

3.3 Arbitrary precision in software

It is possible that the precisions supported by language and machine are not sufficient to meet an application's needs. Software libraries can provide the illusion of extended precision, by mapping software-created data types to those of language/machine precisions. Shewchuk [9] computes floating-point geometric predicates using *arbitrary precision*, which allows the representation of numbers whose precision surpasses the usual limits.

In particular, Shewchuk uses *multiple-term format* for storing numbers of arbitrary precision. Multiple-term format represents a number as a sum of ordinary floating-point numbers, each with its own sign, exponent, and significand. An arbitrary precision value x is expressed as an expansion $x = x_n + \ldots + x_2 + x_1$, where each x_i is a floating-point value. For Shewchuk's application, he imposes some additional structure on the expansion by requiring that it be *nonoverlapping* and ordered by magnitude. Two floating-point values qualify as nonoverlapping if the least significant nonzero bit of one is more significant than the most significant nonzero bit of the other (assuming base two). For such an expansion x, x_n is an easy but rough approximation of x, and the sign of x is simply the sign of x_n (Section 4.3 discusses why this is desirable). Suppose that each x_i's significand is limited to 6 digits of precision. A number $x = 1010.1101101_2$, which requires 11 digits for exact representation, can be stored as $x = x_2 + x_1 = 1010.11_2 + 0.0001101_2 = 1010.1101_2$. Shewchuk also provides algorithms for the addition and multiplication of expansions.

The multiple-term format, and arbitrary arithmetic in general, is a nice option, but it has a downside. It gives the writer of a software library much flexibility in representing and operating on numbers of the precision that she desires; however, it also burdens the library-writer, by forcing her to consider the details that are often left to hardware and compilers. The readability, portability, and development speed of the library's code suffers as a result.

4 Speed vs. Accuracy

The great expense of floating-point representation and arithmetic cannot be ignored. At a minimum, operations in floating-point arithmetic require more work than simply adding, subtracting, multiplying or dividing two integers. Of course, it is possible to *pipeline* such work, an implementation technique that overlaps the execution of multiple instructions. However, handling exceptional cases, like overflow/underflow and rounding (see Section 2.3), can consume even more execution time. An increase in the number of bits to be manipulated, say for greater precision, also increases execution time.

In a word, it is the *accuracy* of a floating-point number that is so expensive. Floating-point operations can certainly gain speed if they take shortcuts in exception-handling and/or desire fewer digits of precision. It can also be efficient to optimize algorithms, which may reorder fragile floating-point operations. Of course, this compromises the correctness of the result. Are there applications in which a fast but possibly inexact solution is more acceptable than a slow but correct solution? Speed for accuracy is an important trade-off, and its applicability should be examined at each level that abstracts floating-point arithmetic.

4.1 A simple goal for floating-point arithmetic in hardware

Kahan [4] proposes a simple goal for any operation of floating-point arithmetic: *Keep the error strictly smaller than one unit in the last place (ulp).* The goal requires every floating-point result to be correct up to the least significant digit. However, describing the goal as simple is misleading, and Kahan quickly says so. It is an easy way of specifying the accuracy of results, but difficult to enforce. Often, keeping the error below one ulp requires greater precision for intermediate calculations (recall Section 3.1). Carrying more and more digits slows computation, and quickly becomes impractical.

Moreover, Kahan's goal accomplishes less than one would hope. This provision guarantees neither the sign-symmetry of $\sin(x) = -\sin(-x)$ nor the monotonicity of $\sqrt{x}$ [4]. For some persisting inaccuracies, achieving this goal is practically impossible. Is the mere possibility of enhancing the accuracy of a floating-point result worth the penalty in execution time? The answer is yes and no. Extending the precision of intermediate calculations benefits computation, but recognizing the point of diminishing returns is crucial. A limit exists beyond which the gains from carrying more digits are sparse, and they cannot offset the extra expense.

4.2 Optimizing compilers

An optimizing compiler not only translates source code into machine instructions, but also structures the machine instructions in a way that encourages efficient computation. Ideally, the compiler should ensure that the processor constantly does useful and non-redundant work, which involves the rearrangement and elimination of some machine instructions. Farnum [2] warns against some common optimizations in the presence of floating-point arithmetic and asserts that under no circumstance should the optimizations change the output produced by a program: "... a compiler whose output produces correct results slowly is preferable to one that quickly produces misleading numbers."

Two optimizations legal under several language standards become dangerous when floating-point arithmetic is introduced. First, reordering expressions often makes efficient use of registers. Because floating-point arithmetic can violate laws of identity, associativity, and commutativity, this may lead to results unintended by the programmer. Second, moving code out of loops or evaluating constant arithmetic at compile-time may decrease execution time. However, floating-point code can have side effects (*i.e.*, flag setting or trapping due to exceptions), and relocating the code may produce side effects inconsistent with the behavior of the code at the old location. Instead, Farnum suggests optimizations that do not affect a program's output: removing unnecessary coercions, vectorizing instructions, and branch prediction.

Floating-point representation already approximates values, so how much worse is another approximation if it provides for an increase in speed? Some applications welcome the potentially dangerous optimizations discussed above. The programmer and user of the program are in the best position to make such a determination, and most optimizing compilers offer varying levels of optimization (and risk). However, the compiler should never perform optimizations that have the potential to alter the program's output without the user's knowledge, and Farnum's suggestions should certainly be followed in the default case.

4.3 Adaptive arithmetic via software

Certainly, the best compromise is to endure a speed penalty only when an application requires the extra accuracy. Hardware and compilers cannot anticipate an application's needs, but software can. With an intimate knowledge of the geometric calculations to be made faster, Shewchuk [9] makes use of *adaptive arithmetic*. Adaptive arithmetic avoids exact computation, but still returns correct answers.

Shewchuk gains speed by exploiting features of the geometric predicates for testing the *orientation* and *incircle* properties. The orientation test determines if a point lies to the left of, to the right of, or on a line or a plane. The incircle test determines if a point lies inside, outside, or on a circle or a sphere. These tests evaluate the sign of a matrix determinant. Shewchuk takes advantage of this property of the geometric predicates: they need only the sign, and not the exact magnitude, of the determinant. A correct result can be achieved without performing an exact computation, and in fact, knowing the determinant's exact value is no more useful than simply knowing its sign. However, it is worth noting that these very geometric tests often fail because of roundoff error. So despite that the tests require only a sign evaluation, it must be a robust evaluation.

Shewchuk's adaptive versions of the geometric predicates compute a sequence of successively more accurate approximations to the determinant. Computation terminates when forward-error analysis indicates that the sign of the approximate result can be trusted. To gain even more efficiency, Shewchuk refines the work done for previous approximations to yield a more accurate approximation at each step.

5 Ease of Development

The ease or difficulty of developing a program includes issues of *reliability*, *readability*, and *portability*. The behavior of a program for the set of all possible input defines its reliability. The readability of a program concerns the presence or absence of defensive, and often cluttered, code for handling special cases or complicated details. The number of different platforms on which a program can execute correctly and reliably affects its portability.

5.1 Many hardware architectures, many arithmetics

> Mathematical craftsmanship can be shared as computer software designed to be used conveniently by people among whom most will understand its mathematics little better than most motorists understand their cars' drive trains. But numerous obstacles impede the dissemination among computers of programs as easy to use as are the keys of calculators . . . One of those obstacles is gratuitous: computer arithmetics are too diverse and . . . occasionally too capricious to allow programs so delicate as those in the calculators to be copied mindlessly onto other machines with no risk of malfunction.

Kahan [4] observes that software sharing is difficult in the absence of a floating-point standard. As arithmetics deviate from one set of hardware to another, a program's reliability is in question. Executing a program, which assumes a particular set of precisions for its data types, on an incompatible machine may yield unexpected results. Moreover, mathematical doctrines vary across computer arithmetics, and Kahan suggests that some are misleading. Even if it is possible to write code compatible with every machine and defensive against every aberration, it is not desirable to do so. The code would be unreadable clutter, which is far, far away from the implementation it was meant to be. The effort of checking for rare anomalies adds to the execution time of the program. Not to mention, the time and aggravation spent developing the code increases.

However, execution time and readability of a program are not the only sacrifices. As Kahan points out, computer software is destined to be used by people who do not understand its workings, and no one obliges them to do so. The understanding required to detect unreliable results from a shared piece of software executing on a unfamiliar computer could be enough knowledge to have written the code oneself. This eliminates the necessity for sharing, but it creates a necessity for many more computer science and mathematics experts.

5.2 The predictability of compiling floating-point code

Recall from Section 3.2 that language standards do not specify precisely the semantics of floating-point operations at the source-level; therefore, compiler writers have freedom in choosing the machine-level floating-point operations to generate. Determining the appropriate precision for subexpression evaluation is a good example of a task left to the compiler. But the compiler may not make the best choice, and it certainly may not make the choice intended by the programmer. Furthermore, different compilers will make different determinations, causing the execution of the program to vary. Mindful of this, a programmer can use explicit type conversions to ensure that the compiler generates code using the intended precisions.

To illustrate explicit type conversions, consider again Example 2 from Section 3.2: the statement `d:=d+s*s`. Suppose the programmer wishes to evaluate the subexpression `s*s` with double precision, to take advantage of the extra precision of `d`. The statement `d:=d+double(s)*double(s)` converts each value of `s` to double precision before multiplying, then the result of the subexpression must also have double precision. Now, there is no room for the compiler to make a possibly undesirable choice.

While explicit type conversion solves the problems of predictability and portability, it detracts from the program's readability by introducing clutter. Farnum [2] proposes a source-to-source translator that inserts explicit precision conversions. Translating the original program just before compilation preserves its readability and saves the programmer from manually inserting the conversions.

5.3 Software depending on arithmetic standards

Software often assumes the existence of some arithmetic standards at the compiler and/or hardware levels. For instance, Shewchuk [9] designed the geometric algorithms to work on computers whose floating-point arithmetic uses a base of two and performs *exact rounding.* The algorithms include Shewchuk's own modification to a pre-existing technique for arbitrary precision arithmetic that runs on a variety of floating-point architectures. Shewchuk improves the speed of the technique significantly by optimizing it for base two with exact rounding. In other words, he gains speed by limiting the application to a specific floating-point architecture.

Exact rounding ensures that if the result of an arithmetic operation is exactly representable in the prescribed floating-point format, then the exact result is stored. Otherwise, the exact result is rounded to the nearest floating-point value that can be represented by the significand. Ties in choosing the nearest representable value may be broken arbitrarily. Shewchuk requires exact rounding in his technique for arbitrary precision arithmetic to facilitate a quick and correct measure of the roundoff error in results.

The geometric predicate software referenced here knowingly decreases its portability for an increase in speed. Shewchuk limits himself, and subsequent users of the software, to executing on computers that comply with his requirements. Exactly how much these requirements hinder the usability of the software depends on how widespread base two and exact rounding are in floating-point architectures. The concluding section takes a look at this issue.

6 Conclusions: IEEE Standard 754 for Binary Floating-Point Arithmetic

Previous sections present some common issues associated with floating-point representation and arithmetic, as well as, how the issues manifest at three levels of abstraction. The viability of the manner in which each hardware architecture, compiler, and piece of software handle floating-point arithmetic is arguable. The real problem is the mere existence of so many different methods for handling floating-point arithmetic; in other words, the absence of a floating-point standard.

6.1 IEEE 754

The disorder among floating-point arithmetics encouraged a collaboration of industry and academia to develop a standard for binary floating-point representation and arithmetic in the late 1970s and early 1980s. William Kahan of the University of California at Berkeley became part of this group as early as their second meeting in November 1977, and went on to present the *KCS Proposal* of which many ideas were eventually included in the *IEEE Standard 754 for Binary Floating-Point Arithmetic*. The standard was published in 1985, but even before its official adoption, many manufacturers implemented IEEE 754 [8, 6, 4].

Overton [6] suggests that the *IEEE Standard 754 for Binary Floating-Point Arithmetic* provides three very important requirements. A fourth requirement, widely regarded as a significant contribution of IEEE 754, is also included here.

First, all machines that adopt IEEE 754 should consistently represent floating-point numbers. The IEEE 754 representation format is as described in Section 2.2, with a required *single format* that uses a 32-bit word (e=8, s=23). The optional *double format* uses a 64-bit word (e=11, s=52) and the majority of computers provide it. *Single precision* and *double precision* correspond to the single and double formats, respectively.

Second, all machines that adopt IEEE 754 should 'correctly' round the results of floating-point operations. Let round(x) be the correctly rounded value of x, let x_+ be the nearest neighbor of x representable in IEEE 754 floating-point format such that $x \leqslant x_+$, and let x_- be the nearest representable neighbor of x such that $x_- \leqslant x$. If x is a number representable in IEEE 754 floating-point format, then round$(x) = x$. Otherwise, round(x) depends on the rounding mode specified, where the choices are the following.

- Round down: round$(x) = x_-$.
- Round up: round$(x) = x_+$.

- Round towards zero: $\mathrm{round}(x) = x_-$ if $x \geqslant 0$. $\mathrm{round}(x) = x_+$ if $x \leqslant 0$.
- Round nearest: $\mathrm{round}(x) = x_-$ or $\mathrm{round}(x) = x_+$, whichever is nearest. In the case of a tie, choose the neighbor whose least significant bit is equal to zero.

Notice that IEEE 754 accomplishes the exact rounding discussed in Section 5.3.

Third, all machines that adopt IEEE 754 should treat exceptional situations consistently. IEEE 754 categorizes exceptions according to the following five types.

- **Invalid operation** is any attempt to compute the quantities $0 \times \infty$ or $0/0$.
- **Division by zero** is the attempt to compute $x/0$, where x can have any legal value.
- **Overflow** (recall Section 2.3).
- **Underflow** (recall Section 2.3).
- **Inexact** occurs any time IEEE 754 cannot exactly represent the result of an arithmetic operation.

When an exception occurs, IEEE 754 signals it by setting the associated flag and responds according to the a default action.

Fourth, all machines that adopt IEEE 754 should support the the special values $\pm\infty$ and NaN. IEEE 754 provides different representations for both $-\infty$ and $+\infty$. Either $-\infty$ or $+\infty$ may be the result of a division by zero or overflow. The NaN, "Not a Number", value corresponds to several bit patterns that symbolically represent various errors. In IEEE 754, when an invalid operation occurs the result is set to a NaN whose representation may contain some diagnostic information regarding its creation. Supporting $\pm\infty$ and NaN values makes IEEE 754 floating-point arithmetic closed (recall Section 2.2). Therefore, upon encountering a division by zero (or an invalid operation) the standard response is to produce an infinite result and continue execution of the program, rather than interrupt or terminate execution (the usual response before the introduction of IEEE 754).

6.2 Hardware support for the standard

Most hardware manufacturers quickly supported the IEEE 754 standard, which eliminates many of the difficulties suffered because of varying machine implementations of floating-point representation and arithmetic. Among the issues benefiting from the overwhelming hardware support of IEEE 754 are some that this paper has emphasized: insufficient precision for intermediate results, keeping approximation error less than one ulp, and the portability of floating-point code.

To fight the accumulation of approximation error, IEEE 754 strongly recommends support for an *extended format* and a corresponding *extended precision.* Recall the benefit of a long precision for intermediate results in Section 3.1. The standard suggests allocating at least 15 bits for the exponent and at least 63 bits for the significand.

IEEE 754's requirement of correct rounding does achieve Kahan's goal in Section 4.1 ("Keep the error strictly smaller than one unit in the last place") for one floating-point operation. Let the *absolute rounding error* associated with a real number x equal $|\mathrm{round}(x) - x|$. Using any of the IEEE 754 rounding modes, the absolute rounding error of x is less than one ulp. In fact, when using *round to nearest* mode, the absolute rounding error of x is at most half an ulp [6]. However, IEEE 754 does not guarantee that a sequence of floating-point operations will yield a correctly rounded result. The accretion of rounding error remains a problem with floating-point arithmetic.

From a hardware perspective, IEEE 754 eliminates the portability issues addressed in Section 5.1. IEEE 754's standardization of floating-point representation and arithmetic allows programmers and end users to form knowledgeable expectations of their floating-point applications. This understanding need not be machine-specific, as a program targeted for one machine that supports IEEE 754 should run on others that support it.

6.3 Software support for the standard

The software support of IEEE 754 is less impressive than the hardware support. In 1996, Kahan [5] commented on the lack of support for IEEE 754 in compilers and programming languages.

> Now atrophy threatens features of IEEE 754 caught in a vicious circle: Those features lack support in programming languages and compilers, so those features are mishandled and/or practically unusable, so those features are little known and less in demand, and so those features lack support in programming languages and compilers.

To take advantage of the large effort required by hardware designers to create and then comply with a standard for floating-point arithmetic, Overton [6] suggests that programming languages do the following: define data types compatible with IEEE 754 formats, allow control of rounding modes, and provide the standard responses to exceptions (while allowing access to exception flags and allowing trapping of exceptions). The response to this call to arms has been mixed. Only a very small number of languages have complied completely with Overton's suggestions and a somewhat larger number have complied partially, but the popular languages of C and Fortran have been very slow to respond. Until programming languages specify precisely the semantics of source-level floating-point operations, the issues of mixed precision (Section 3.2), ill-advised compiler optimizations (Section 4.2), and the predictability of compilers (Section 5.2) will remain unresolved. Since programming languages and compilers do not fully support the IEEE 754 standard, many argue that the efforts put forth in hardware design cannot be completely realized.

However, we should not overlook the benefit of raising awareness about floating-point issues. Encouraging the communities of computer science, mathematics, physics, and beyond to be informed programmers and users of floating-point codes has been the greatest contribution of IEEE 754 and the event surrounding it. For Shewchuk's applications, a floating-point standard will never meet his needs of arbitrary precision (Section 3.3) and adaptive arithmetic (Section 4.3), so he makes use of the best floating-point hardware and software have to offer. Furthermore, Shewchuk takes advantage of widespread hardware support for IEEE 754, by assuming base two and exact rounding to gain efficiency (Section 5.3). Due to the sheer nature of floating-point representation and arithmetic, there will always be some level of approximation, and the best defense against this uncertainty is understanding it.

References

[1] G. A. Blaauw and F. P. Brooks, Jr. *Computer Architecture: Concepts and Evolution.* Addison-Wesley, 1997.

[2] C. Farnum. Compiler support for floating-point computation. *Software–Practice and Experience,* 18(7):701–709, July 1988.

[3] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley & Sons, 1979.

[4] W. Kahan. Mathematics written in sand. In *Proceedings of the Joint Statistical Meeting of the American Statistical Association*, pages 12–26, 1983.

[5] W. Kahan. IEEE Standard 754 for binary floating-point arithmetic. Lecture notes on the status of IEEE 754, May 1996.

[6] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, 2001.

[7] N. R. Scott. *Computer Number Systems & Arithmetic*. Prentice Hall, 1985.

[8] C. Severance. An interview with the old man of floating-point. *IEEE Computer*, March 1998.

[9] J. R. Shewchuk. Robust adaptive floating-point geometric practices. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 141–150, May 1996.