Why do we need a floating-point arithmetic standard?

W. Kahan

University of California at Berkeley
March 5, 1981

Contents

Why do we need a floating-point arithmetic standard?

W. Kahan

University of California at Berkeley

~~February 17~~, 1981

Mar. 5,

> "...the programmer must be able to state which properties he requires... Usually programmers don't do so because, for lack of tradition as to what properties can be taken for granted, this would require more explicitness than is otherwise desirable. The proliferation of machines with lousy floating-point hardware – together with the misapprehension that the automatic computer is primarily the tool of the numerical analyst – has done much harm to the profession."
>
> Edsger W. Dijkstra [1]

> "The maxim 'Nothing avails but perfection' may be spelt shorter, 'Paralysis'."
>
> Winston S. Churchill [2]

After more than three years' deliberation, a subcommittee of the IEEE Computer Society has brought forth a proposal [3, 4, 5] to standardize binary floating-point arithmetic in new computer systems. The proposal is unconventional, controversial and a challenge to the implementor, not at all typical of current machines though designed to be "upward compatible" from almost all of them. Be that as it may, several microprocessor manufacturers have already adopted the proposal fully [6, 7, 8] or in part [9, 10] despite the controversy [5, 11] and without waiting for higher-level languages to catch up with certain innovations in the proposal. It has been welcomed by representatives of the two international groups of numerical analysts [12, 13] concerned about the portability of numerical software among computers. These developments could stimulate various imaginings: that computer arithmetic had been in a state of anarchy; that the production and distribution of portable numerical software had been paralyzed; that numerical analysts had been waiting for a light to guide them out of chaos. Not so!

Actually, an abundance of excellent and inexpensive numerical software is obtainable from several libraries [14-21] of programs designed to run correctly, albeit suboptimally, on almost all major mainframe computers and several minis. In these libraries many a program has been subjected to, and has survived, extensive tests and error-analyses that take into account the arithmetic idiosyncrasies of each computer to which the program has been calibrated, thereby attesting that no idiosyncrasy defies all understanding. But the cumulative effect of those idiosyncrasies and the programming contortions they induce imposes a numbing intellectual burden upon the software industry. To appraise how much that burden costs us we have to add it up, which is what this paper tries to do.

This paper is a travelogue about the computing industry's arithmetic vagaries. Instead of looking at customs and superstitions among primitive tribes, we shall look at arbitrary and unpredictable constraints imposed upon programmers and their clients. The constraints are those associated with arithmetic semantics rather than syntax, imposed by arithmetic hardware rather than by higher-level languages. This is not to say that the vagaries of higher-level language design, of compiler implementation, and of operating system conventions are ignorable, even if sometimes they can be circumvented by assembly language programming. Language issues are vital, but our itinerary goes beyond them.

Numerical software production is costly. We cannot afford it unless programming costs are distributed over a large market; this means most programs must be *portable* over diverse machines. To think about and write portable programs we need an abstract model of their computational environment. Faithful models do exist, but they reveal that environment to be too diverse, forcing portable programmers to bloat even the simplest concrete tasks into abstract monsters. We need something simple or, if not so simple, not so capriciously complex.

Rational One-Liners.

Why are continued fractions used far less often than their speed and sometimes accuracy seem to deserve? One reason can be gleaned from the example

$$R(z) := 7 - 3/(z-2 - 1/(z-7 + 10/(z-2 - 2/(z-3))))$$

which behaves well ($3.7 < R(z) < 11.8$) for all z and can be computed fairly accurately and fast from the foregoing "one-line" definition provided certain conventions like

$$(\text{nonzero})/0 \to \infty, \quad (\text{finite})+\infty \to \infty, \quad (\text{finite})/\infty \to 0$$

have been built into the computer's arithmetic, as has been done to some machines. But on most machines attempts to calculate

$$R(1) = 10, \quad R(2) = 7, \quad R(3) = 4.6, \quad R(4) = 5.5$$

stumble after division by zero, which must then be avoided if the program is to be portable over those machines too. Another algebraically equivalent one-line definition

$$R(z) := ((((7z-101)z+540)z-1204)z+958)/((((z-14)z+72)z-151)z+112)$$

avoids division by zero but falls afoul of exponent overflow when z is huge enough, no bigger than 3×10^9 on some machines; moreover, this second expression for $R(z)$ costs more arithmetic operations than the continued fraction and is less accurate. In general, no way is known to avert spurious over/underflow, division by zero or loss of accuracy, without encumbering expressions with tests and branches that result in portable but inscrutable programs.

Tests and Branches.

What makes tests and branches expensive is that programmers must decide *in advance* where and what to test; they must anticipate *every* undesirable condition in order to avoid it, even if that condition cannot arise on any but a few of the machines over which the program is to be portable. Consequently, programmers generally are obliged to know that on some widely used computers a statement like

if $z \neq 0$ **then** $s := 3-\sin(z)/z$ **else** $s := 2$

will, when executed with certain very tiny values z, stop the machine and allege that division by zero was attempted. These machines treat all sufficiently tiny nonzero numbers z as if they were zero during multiplication and division, but not during addition and subtraction; consequently these machines calculate

$z/0.004 = z \times 250. = 0$ and $0.004/z =$ (division by zero)

whereas

$(z+z)/0.008 = (z+z)\times 125. \neq 0$ and $0.002/(z+z) =$ (a finite number).

To be portable over these machines the statement above must be changed to

if $1\times z \neq 0$ **then** $s := 3 - \sin(z)/z$ **else** $s := 2$

or better

if $1+|z| \neq 1$ **then** $s := 3 - \sin(z)/z$ **else** $s := 2$.

The last test opens another can of worms.

Some compilers try to be helpful by using extra precision to calculate subexpressions during the evaluation of arithmetic expressions. This is a good idea provided the programmer knows that it is being done. Otherwise conundrums can be created by statements like

```
p := q + r ;
x := y + z ;
if x ≠ y+z then print "why not?" ;
if p ≠ q+r then print "how come?" ;
```

which print nothing on some systems, print *why not*? *how come*? when $q+r$ and $y+z$ are evaluated to more precision than can be stored in p and x, and print just *how come*? when the compiler's optimizer notices that the subexpression $(x \neq y+z)$ involves a value x that has just been calculated in an extra-wide register and need not be reloaded from memory. Consequently subexpressions like $(y+z \neq y)$ may remain *true* even when $|z|$ is so tiny that $y+z$ and y would be equal were they rounded to the same precision.

Precision and Range.

The accuracy of floating-point arithmetic operations is worse than about 6 significant decimals on some machines, better than 33 on others. Some machines serve more than one level of precision, some as many as four. One machine's single-precision format can be almost as accurate as another machine's double. If he does not know how precise "SINGLE PRECISION" really is, the would-be portable programmer faces dilemmas. An algorithm that is faster than any other to achieve modest accuracy may be incapable of achieving high accuracy. An algorithm that works superbly if executed in arithmetic substantially more accurate than the given data and desired solution may fail ignominiously if the arithmetic is only slightly wider than the data and solution. An algorithm that uses some double-precision arithmetic to support successfully a computation performed mainly in single-precision may collapse if "DOUBLE PRECISION" is actually less than twice as wide as "SINGLE PRECISION", as happens on several machines. Therefore a library of portable programs may have to cope with a specific task by including just one program that is grossly suboptimal on almost every machine, or else by including several similar programs of which each user must reject all but the one that suits his own machine. Neither choice is a happy one for the people who assemble and maintain the library.

A similar dilemma is posed by various machines' over/underflow thresholds.

The overflow threshold Λ is the largest number, the underflow threshold λ is the smallest positive normalized number that can be represented by a machine's floating-point arithmetic. The diversity of thresholds is sampled in Table 1. Worse than that diversity is the unpredictability of reactions to over/underflow; many machines trap or stop, most set underflows to zero, some overflow to Λ, some overflow to ∞, a few overflow to zero, and so on.

Table 1: Floating-Point Over/Underflow Thresholds

Machine	Underflow λ	Overflow Λ
DEC PDP-11, VAX, F and D formats	$2^{-128} \approx 2.9\times10^{-39}$	$2^{127} \approx 1.7\times10^{38}$
DEC PDP-10; Honeywell 600, 6000; UNIVAC 110x single; IBM 709X, 704X	$2^{-129} \approx 1.5\times10^{-39}$	$2^{127} \approx 1.7\times10^{38}$
Burroughs 6X00 single	$8^{-51} \approx 8.8\times10^{-47}$	$8^{76} \approx 4.3\times10^{68}$
H-P 3000	$2^{-256} \approx 8.6\times10^{-78}$	$2^{256} \approx 1.2\times10^{77}$
IBM 360, 370; Amdahl; DG Eclipse M/600; ...	$16^{-65} \approx 5.4\times10^{-79}$	$16^{63} \approx 7.2\times10^{75}$
Most handheld calculators	10^{-99}	10^{100}
CDC 6X00, 7X00, Cyber	$2^{-976} \approx 1.5\times10^{-294}$	$2^{1070} \approx 1.3\times10^{322}$
DEC VAX G format; UNIVAC 110X double	$2^{-1024} \approx 5.6\times10^{-309}$	$2^{1023} \approx 9\times10^{307}$
HP 85	10^{-499}	10^{500}
Cray 1	$\approx 2^{-8192} \approx 9.2\times10^{-2467}$	$\approx 2^{8192} \approx 1.1\times10^{2466}$
DEC VAX H format	$2^{-16384} \approx 8.4\times10^{-4933}$	$2^{16383} \approx 5.9\times10^{4931}$
Burroughs 6X00 double	$8^{-32755} \approx 1.9\times10^{-29581}$	$8^{32780} \approx 1.9\times10^{29603}$
Proposed IEEE Standard: INTEL i8087; Motorola 6839		
single	$2^{-126} \approx 1.2\times10^{-38}$	$2^{128} \approx 3.4\times10^{38}$
double	$2^{-1022} \approx 2.2\times10^{-308}$	$2^{1024} \approx 1.8\times10^{308}$
double-extended	$\leq 2^{-16382} \approx 3.4\times10^{-4932}$	$\geq 2^{16384} \approx 1.2\times10^{4932}$

No wonder then that simple tasks spawn hordes of complex programs; here is one example, the calculation of the root-sum-squares norm of a vector V,

$$Rtsmsq(n,V) := \sqrt{(\ V_1^2 + V_2^2 + ... + V_n^2)}.$$

The obvious program is a simple one:

```
sum := 0;  for i = 1 to n do sum := sum + V[i]**2 ;
Rtsmsq := √(sum) .
```

This simple program is the best on machines with ample range and precision, but on most machines this program encounters at least one of the following hazards:

i) When n is huge (10^6) but the precision is short (6 significant decimals) then *sum*, and hence *Rtsmsq*, may be badly obscured by roundoff amounting to almost $n/2$ units in its last place.

ii) Even though *Rtsmsq*'s value should be unexceptional, *sum* may over/underflow (e.g. if some $|V[i]| > \sqrt{\Lambda}$ or all $|V[i]| < \sqrt{\lambda}$).

The simplest way to subdue both perils is to evaluate the sum of squares using extra precision and range as may be achieved in a few computing environments via a declaration like

Double Precision *sum*.

The proposed IEEE floating-point arithmetic standard allows implementors, at their option, to offer users just such a capability under the name "Extended Format". But most computing environments afford no such luxury, and instead oblige programmers to circumvent the hazards by trickery. The obvious way to circumvent hazard (ii) is to scan V to find its biggest element V_{max} and then evaluate

$$Rtsmsq := |V_{max}| \times \sqrt{(\sum_1^n (V[i]/V_{max})^2)}$$

but this trick violates all but the third of the following constraints upon the calculation:

I) Avoid scanning the array V more than once because, in some "virtual memory" environments, access to $V[i]$ may cost more time than a multiplication.

II) Avoid extraneous multiplications, divisions or square roots because they may be slow. For the same reason, do not request extra precision nor range.

III) Avert overflow; it may stop the machine.

IV) Avert underflow; it may stop the machine.

The only published program that conforms to all four constraints is due to J. L. Blue [22]. Other published programs ignore constraint IV and assume underflows will be flushed to zero. One such program is C.L. Lawson's SNRM2 in LINPACK[17], called *norm* by W.S. Brown[23]. Another program, VECTOR_NORM by Cox and Hammarling [24], violates constraint II. All these programs succumb to the first hazard (i) above, so there is need for yet another program; it will be furnished in Figure 7. Only this last program can be generalized conveniently to cope with sums of products as well as sums of squares, and then only by violating constraint III, as will be shown in Figure 8. None of the programs is transparent to the casual reader. None is satisfactory for vectorized machines.

Suppose an ostensibly portable program works correctly for all physically meaningful data when run on one of the machines with a wide range listed below the middle of Table 1. But the program is not robust in the face of intermediate

over/underflow, so it produces wrong answers and/or warning messages and/or stops when run with meaningful but unusual data on a machine with a narrow range. Who is to blame? We, who supply machines and programs, tend to exculpate ourselves and blame instead whoever used *that* program to treat *that* data on *that* machine; he should have spent more money to buy a machine with far wider range than encompasses his data and output, or he should have paid more money for a better and robust but more elaborate program, or he should not worry about unusual data beyond the normally ample capacity of what we have recently sold to him. Is this issue really just a question of cost vs. capability? No. From time to time a simple program, run on a system with narrow range and precision but designed felicitously, will deliver better results and sooner than an elaborate program run on a system with wider range and precision. Thus the competency of its design, its intellectual economy and many other parameters of a system must figure significantly enough in its performance to deserve our consideration too.

Radix.

Almost every machine that provides floating-point arithmetic does so in binary (radix 2), octal (8), decimal (10) or hexadecimal (16). Biological and historical accidents make 10 the preferred radix for machines whose arithmetic will be exposed to frequent scrutiny by humans. Otherwise binary is best. Radices bigger than 2 may offer a minuscule speed advantage during normalization because the leading few significant bits can sometimes remain zeros, but this advantage is more than offset by penalties in the range/precision tradeoff [25] and by "wobbling precision" [19, p.7]. For instance, the proposed IEEE standard squeezes as much range and worst-case precision from a 32-bit binary format as would demand 34 bits in hexadecimal. For the programmer whose task is to produce

as accurate a program as possible

the technical hindrance arises less from not enjoying the use of the optimal radix than from not knowing which radix his program will encounter.

Consider for example two algebraically equivalent expressions

$$q_1(z) := 1/(1+z); \quad q_2(z) := 1 - z/(1+z) \ .$$

Which one can be calculated more accurately? If $|z|$ is big then $q_1(z)$ is better because $q_2(z)$ suffers from cancellation. If $|z|$ is tiny then $q_1(z)$ is worse because its error can be bigger than $q_2(z)$'s by a factor almost as large as the radix, and this is serious if the radix is 16 and the precision short. To minimize that error a conscientious programmer might write

if $0 < z < t(B)$ **then** $q(z) := 1 - z/(1+z)$ **else** $q(z) := 1/(1+z)$

where $t(B)$ is a threshold whose optimal value depends deviously upon the radix B and upon whether arithmetic is rounded or chopped. Specifically, when arithmetic is rounded after normalization the optimal values are

$$t(2) = 1/3, \ t(8) = 0.728, \ t(10) = 0.763, \ t(16) = 0.827 \ ;$$

but when arithmetic is chopped after normalization the optimal values are different. And when arithmetic is rounded or chopped before normalization, different thresholds and a rather different program are called for:

if $0 < z < t(B)$ **then** $q(z) := (0.5 - z/(1+z)) + 0.5$
else $q(z) := 1/(0.5 + (z + 0.5))$.

The reason for using 0.5+0.5 in place of 1 will become clear later.

End Effects.

Some computers can do funny things. Each of the following phenomena is possible for a wide range of operands on some machine which is or was widely used:

$y \times z \neq z \times y$; $z \neq 1 \times z \neq 0$; $z = y$ but $z - t \neq y - t$; $1/3 \neq 9/27$.

These phenomena are caused by peculiar ways of performing roundoff. Further anomalies are caused by peculiar ways of handling exponent over/underflow without stopping the machine and sometimes without any indication visible to the program or its user:

$((y \times z)/y)/z < 0.00001$	...caused by overflow to Λ ;
$y > 1 > z > 0$ but $y/z = 0$	...caused by overflow to 0 ;
$((y \times z)/y)/z > 100000.$	...caused by underflow to λ ;
$y/z < 0.99$ but $y - z = 0$	...caused by underflow to 0 ;
$a>0,\ b>0,\ c>0,\ d>0,\ z>0,$ but $\dfrac{(a \times z + b)/(c \times z + d)}{(a + b/z)/(c + d/z)} > 1.5$	...caused by underflow to 0.

Other paradoxes were discussed above under **Tests and Branches**. Some further anomalies cannot be blamed upon computer architects. For instance, discrepancies can arise whenever decimal-binary conversion is performed differently by the compiler than by the run-time Input/Output utilities:

> Input z ... *the user types* 9.999 *to signal* end-of-data...
> **if** $z = 9.999$ **then** print result **else** continue processing data;
> ... *but no* result *ever gets printed.*

These funny things computers do can cause confusion. Some of the confusion can be alleviated by education, whereby we come to accept and cope with those anomalies that are inescapable consequences of the finiteness of our machines. But education cannot mitigate the demoralizing effects of anomalies when they are unnecessary or inexplicable, when they vary capriciously from machine to machine, when they occur without leaving any warning indication, or when no practical way exists to avert them.

The end effect of caprice is a perverse indoctrination. After a while programmers learn to distrust techniques which formerly worked perfectly and provably on their old computer system but now fail mysteriously on the new and better system. By declaring those techniques to be "tricks", as if they never deserved to work, we reverse the traditional educational paradigm:

> "A trick used three times is a standard technique."
> (Attributed to G. Polya.)

Models.

I know about several attempts to impose some kind of intellectual order upon the arithmetic jungle. An early attempt by van Wijngaarden [26] failed partly because it was excessively abstract and complicated (32 axioms) and partly because a few very widely used computers did not conform to his model. Lately W.S. Brown [23, 27, 28] has contrived another model. It is an outstanding accomplishment, simultaneously simpler and more realistic than every previous attempt, easily the best available description of floating-point arithmetic for programs that must be portable over all machines within reason. By apt

assignment of possibly pessimistic values for a few parameters including radix, precision and over/underflow thresholds pertinent to an artfully designated subset of the computer's floating-point number system, Brown's model encompasses not only

> "... computers of mathematically reasonable design, but can also encompass a variety of anomalies... While a thorough study of real-world floating-point anomalies may lead one to despair, the situation can be summarized rather neatly, and with little exaggeration, by stating that any behavior permitted by the axioms of the model is actually exhibited by at least one commercially important computer." [23, pp. 11-12]

Conversely, every anomaly enumerated in previous paragraphs, except possibly unannounced overflow to zero, is subsumable within a model like Brown's. All these models, new and old, share the notion of a fuzzy floating-point variable whose fuzziness, though unknowable, cannot exceed a known tolerance.

The proposed IEEE standard is quite different. Rather than describe abstractly some long list of minimal properties that an arithmetic engine must honor, the proposal prescribes in detail how an arithmetic engine shall be designed, whence follow the minimal properties and many more. The engine's designer is allowed only a limited amount of leeway in the optional features, mostly related to capacity, that he may choose to implement; the designer may choose to support only the single-precision format (32 bits wide), or to support two formats, or possibly three, and to charge different prices accordingly. Each format has its designated subset of the real numbers represented as floating point numbers; the single-precision format has one sign bit, an eight-bit exponent field, and a 23-bit field for the significand's fraction, allowing for one more "implicit bit" to make up 24 bits of precision. Each format has its own over/underflow thresholds (cf. Table 1) and special bit-patterns reserved for $\pm\infty$ and *NaNs*; more will be said later about *NaN* = *N*ot-a-*N*umber. Also to be discussed later are the obligatory responses prescribed by the standard for every exception (Invalid Operation, Division by Zero, Over/Underflow and Inexact Result); for now we note that the designer can supplement but not supplant those responses, so programmers can predict for every exception what response will occur unless a program has explicitly requested something else. The designer may choose how much of the proposal to implement in hardware, how much in firmware (microcode in read-only memory), how much in software, thereby trading off speed against cost. The proposal says nothing about the relative speeds of, say, multiplication vs. division. But the designer cannot perform roundoff arbitrarily; his design must conform to the following rules unless a program asks explicitly for something else:

Every algebraic operation ($+$, $-$, $\times$, $/$, $\sqrt{}$) upon one or two operands must deliver its result to a destination, either implicit in an arithmetic expression or designated explicitly by a program's assignment statement; the destination's format cannot be narrower than either operand's format.

> The result delivered must be the closest in the destination format to the exact value that would have been calculated were range and precision unbounded; and if the exact value lies just midway between two adjacent numbers representable in the destination's format then the one whose least significant digit is even shall be the result.

The only exceptions to this rule are the obvious ones — Invalid Operations like 0/0 with no exact value, and Overflow to $\pm\infty$ which occurs only when the rounded value would otherwise be bigger than the overflow threshold Λ.

These rules are comparatively simple as arithmetic rules go, and permit a programmer to infer from his chosen format(s) exactly how the arithmetic engine will behave. Most of the inferences are as pleasant as anyone accustomed to computation might desire. Some of the inferences associated with underflow are slightly surprising to programmers accustomed to having underflows go to zero; that is contrary to the rules set out above. Therefore the proposal includes a *Warning Mode* designed to defend those programmers against arithmetic ambush; it will be discussed later.

The difference between the proposed IEEE standard and the aforementioned models boils down to this: Having selected a format with its concomitant width, radix (2), precision and range, a programmer knows exactly what results must be produced by arithmetic engines that conform to the standard, whereas an engine that merely conforms to one of those models is capable of excessive arithmetic diversity. Programming for the standard is like programming for one of a small family of well-known machines, whereas programming for a model is like programming for a horde of obscure and ill-understood machines all at once.

Program Libraries' Costs and Penalties.

In the absence of a prescriptive standard like the IEEE proposal, two strategies are available to the would-be architect of a great library of numerical software. One strategy is to ...

Customize: Calibrate a version of the library to the arithmetic idiosyncrasies of each computer upon which the library is intended to be supported.

This is the strategy chosen most often for the elementary transcendental functions like exp, cos, ... and for some similarly heavily used higher transcendental functions like erf. It could lead to almost as many different libraries as there are different styles of arithmetic, though the situation is not yet that bad.

A second strategy is to strive for universal ...

Portability: Impose upon programmers a discipline whereby all their programs exploit only those arithmetic properties supported by some universal model encompassing all styles of arithmetic within reason; when the discipline succeeds the programs are provably portable to all machines within reason.

This strategy has succeeded for many matrix calculations and, when environmental parameters [29, 30] pertaining to radix, precision and range are accessible to the program, for iterative equation-solving, quadrature, and much more. But the parameters are not always easy to interpret unambiguously [31, 32]; see the section after next. Neither need a program's reliability and effectiveness be easy to prove from the model's abstract axioms. Suppose a programmer seeks but cannot find such a proof; the logical next step is to scrutinize his program to find a *bug* and fix it. After exhaustive tests reveal no bug, the programmer may suspect that only because he is unskilled in the model's style of inference was he unable to find a proof. What should he do next? Should he encumber his program with unnecessary defenses against imaginary threats? Suppose he can prove by test as well as theory that his program works flawlessly on his own machine, with which he has become thoroughly familiar; has he the right to hope that his program will not fail except on some hypothetical machine that, while conforming to the model, does so only perversely? This question will be re-examined a few paragraphs below.

The architects of the great numerical subroutine libraries [14-21] deserve our admiration for their perseverance in the face of arithmetic anomalies which appear to be accumulating insurmountably, though each is by itself a minor

irritant. To contain costs, the architects have pursued the second strategy, *portability*, whenever possible, even if occasionally a proliferation of ostensibly portable programs had to be tolerated in order to accommodate irreconcilable differences among arithmetic engines. The results have been surprisingly good, all things considered. But these triumphs of intellect over inscrutability are Pyrrhic victories won at the cost of too many man-years of misdirected ingenuity; had the libraries not been subsidized by government [33] nor by occasionally inadvertent corporate munificence, none of us could afford to use them. As befits an era of diminished expectations, the libraries' performance has intentionally been compromised; some programs accept an unexpectedly limited range of data, some programs are less accurate than they could be, some more complicated to use than they should be, some less helpful than we would like when things go wrong, and some are slow. We shall see in detail why these performance penalties cannot be avoided entirely if programs must be portable over machines with unnecessarily widely disparate arithmetic engines. Such penalties, or the belief that they exist, tend to undermine the perceived utility of the libraries and stimulate an urge to replace a portable program by another, made-to-order and presumably more nearly optimal for a specific machine. Moreover, programmers have egos that will not be denied self-expression. Ironically, many a made-to-order program has turned out worse than the library program it was to supplant. Let us not blame sub-optimal decisions about sub-optimal programs upon sub-optimal programmers when the culprit is actually a programming environment so sub-optimal as to defy education and repel tidy minds. Let us look at that environment.

Models of Paranoia.

No realistic conclusion about the programming environment for portable numerical software can be drawn without some experience of the way simple tasks turn into quagmires. Here is an example of a simple task:

Write two fast, accurate and portable programs to calculate

$$\sin\vartheta(t) \text{ given } t = \tan(\vartheta/2), \text{ and also}$$
$$\Psi(t) = \sqrt{(\arcsin(1)^2 + \arccos(0.25 + 0.75\sin^3\vartheta(t))^2)}.$$

The reader may escape the quagmire by skipping over several pages to **Diminished Expectations** but the programmer assigned a task like this must wade through the following muck.

The diligent programmer soon discovers that

$$\sin\vartheta(t) = \sin(2\arctan t) = 2/(t + 1/t) = 2t/(1+t^2),$$

and the last two expressions cost much less time to evaluate than $\sin(2\arctan(t))$ provided $1/t$ or t^2 does not overflow. However $|\sin\vartheta| \le 1$ whereas both $|2/(t + 1/t)|$ and $|2(t/(1 + t^2))|$ might conceivably exceed 1 by a rounding error when evaluated for t slightly less than 1 on some (unknown) machine. Therefore the formula used for $\sin\vartheta$ when $|t|$ is near 1 must (for the sake of portability to that unknown machine) be transformed into, say,

$$\sin\vartheta(t) = t/(|t| + (|t|-1)^2/2),$$

from which $|\sin\vartheta| \le 1$ follows immediately because universally

$$|y| \le z \text{ implies } |y/z| \le 1$$

despite roundoff. Keeping $|\sin\vartheta| \le 1$ avoids misadventure during subsequent calculation of

$$\arccos(0.25 + 0.75(\sin\vartheta)^3)$$

by constraining the arccosine's argument to lie always between −0.5 and 1 inclusive despite roundoff. These precautions are justified because without them the arccosine expression above could be invalid on some machines; it flashes lights on the T.I. SR-52, 58, 58C and 59 when they calculate $\sin \vartheta = 1.000000000004$ at $\vartheta = 89.99995$ degrees or $\vartheta = 1.5707954541$ radians.

The arccosine's argument must be defended against another hazard that could force it past 1 into misadventure. Although constants like 0.5, 0.25 and 0.75 are all representable exactly in every computer in the Western world, they could be blurred slightly by inept decimal-binary conversion. Such blurring occurs sometimes when 0.75 is converted not as 75/100 but as 75×0.01 using an approximation to 0.01 drawn from a table of powers of 10 converted into binary. If the decimal value 0.75 is converted to the binary string 0.1100...001 in a machine that rounds sums, then the arccosine expression above will be rendered invalid by an argument bigger than 1 when $\sin\vartheta = 1$. This kind of unnecessarily blurred conversion of modest-sized constants is forbidden by the proposed IEEE standard but not by Brown's model [23, 27, 28, 29], so the prudent programmer should not assume exact conversion for any constants other than small integers. Consequently the prudent programmer will replace the arccosine expression above by

$$\arccos\left((1 + 3(\sin\vartheta)^3)/4\right)$$

to avoid misadventure, though it costs an extra division.

```
Real function sin2arctan (t) : real t ;
        if |t| ≥ 2 then return   2/(t + 1/t)
        else return   t/(|t| + 0.5×(|t|−1)**2)
                      end sin2arctan;

Real function psi(t) : real t ;
        return √(arcsin(1)**2 + arccos((1 + 3 × sin2arctan(t)**3)/4)**2)
                      end psi.
```

Figure A

From these protracted deliberations ensue the brief programs shown in Figure A. The constant $\arcsin(1)^2$ has been retained intact to promote portability regardless of whether angles are reckoned in degrees or radians. The notation z**2 and z**3 has been used to represent $z \times z$ and $z \times z \times z$ respectively rather than $\exp(2\times\ln(z))$ and $\exp(3\times\ln(z))$ which misbehave when $z \le 0$. These programs assume that square root, arcsin and arccos subroutines are available and accurate to within a few ulps (*U*nits in the *L*ast *P*lace), in which case the programs in Figure A can be proved to be comparably accurate and fast on every computer in the Western world with full floating-point division built into its hardware. But these programs cannot be proved portable using Brown's model nor any other that encompasses *all* arithmetic engines in current use!

How might Figure A's programs fail? A few computers calculate every quotient y/z as a product $y\times(1/z)$ after estimating the divisor's reciprocal $1/z$. As long as that reciprocal is correctly rounded or chopped, the proposition

$$|y| \le z \text{ implies } |y \times (1/z)| \le 1$$

will remain valid despite roundoff, and Figure A's programs will continue to function flawlessly as befits their careful design. But the proposition cannot be proved from any version of Brown's model that encompasses the Cray-1, a

machine on which division y/z entails an estimate for $1/z$ that could be a bit too big. Because the inference $|y \times (1/z)| \leq 1$ cannot be supported by the model, the programs in Figure A must be declared non-portable even though they are provably infallible on every mainframe in the Western world except *maybe* the Cray-1.

In fairness to the Cray-1 we should digress momentarily. Its arithmetic is not so aberrant as to deserve invidious mention. Had a different task than $\Psi(t)$ been selected to illuminate a different quagmire, some other computer would have emerged egregious. The speed of the Cray-1 makes it too important commercially to be excluded from any arithmetic model with universal pretensions, so Brown's model must omit division from his list of "*basic arithmetic operations* ... addition, subtraction, multiplication, ..." [23, p. 6]. Because too little is known about the Cray-1's reciprocal and division algorithms [34, p. 3-30], nobody can tell whether the proposition above is still valid or not; values y and z may exist for which $|y| \leq z$ but the Cray-1 calculates $|y \times (1/z)| > 1$. Therefore this possibility must be allowed by the axioms in Brown's model, as indeed it is. Unfortunately, the possibility is nowhere mentioned explicitly among Brown's twenty-odd theorems and lemmas, so a programmer unfamiliar with a Cray-1 might be forgiven if at first she takes the proposition for granted.

Is it fair to blame a programmer for a bug that cannot come alive except *possibly* when her programs run on a Cray-1? Not unless the old song that says

"... It's always the woman that pays..."

sets the standard for fairness. None the less, dogmatic adherence to the axioms of universal portability demands that either the programs or the computer be mended, and the programs are more eligible than the computer. Let us consider how to change the programs.

The customary way to defend arccos(z) from an argument with $|z| > 1$ is to insert conditional statements like

if $|z| \leq 1$ **then** ... arccos (z) ...
else ...

into the program. Good programmers know many reasons to eschew such expedients. First, conditional branches hamper high-speed machines that would otherwise achieve part of their speed by looking ahead into the instruction stream. Secondly, conditional statements make for cluttered and ugly programs. These two considerations induce the best programmers to linger over conditional statements hoping to excise some of them, thereby exacerbating the third consideration:

> Among the most time-consuming tasks that confront numerical programmers are the decisions about thresholds and tests -- where to put them, what to test, where to go afterwards, when to quit.

For instance, why was 2 chosen for the threshold value compared with $|t|$ in Figure A's program sin2arctan? For machines whose divide is much slower than multiply, a threshold much larger than 2 would yield a faster program. On the other hand, a carelessly chosen threshold could conspire with roundoff to subvert the monotonicity of sin2arctan(t) as $|t|$ crosses the threshold, so the threshold should be a modest power of 2 at which sin2arctan suffers at most one rounding error. Whether this threshold matters or not, its choice will dissipate a programmer's time.

The crowning irony is that a test like the customary

if $|z| \le 1$ **then** ... arccos (z) ...
else ...

cannot be proved via Brown's model to defend arccos(z) against $|z| > 1$ even though it works infallibly on the Cray-1, the only machine on which the test might be needed. The model renders the test futile by allowing for the possibility that a computer may say $|z| \le 1$ when actually $|z| > 1$ by a bit. In the model all comparisons, like all numbers, are a bit fuzzy. Machines do exist that will allege $|z| \ge 1$ when actually $|z| < 1$ by a bit; among them are the T.I. MBA and CDC 6400, 6600, 7600 and Cyber machines. Later we shall see what causes such fibs. Notwithstanding these precedents and the model's license, no computer yet built will say $|z| \le 1$ when actually $|z| > 1$; whatever might cause this fib would probably cause the computer to calculate $x-x < 0$ for every $x > 0$ contrary to both Brown's model and normal expectations. Perhaps some day the model may be refined to reflect another property of real arithmetic engines, namely that a difference $x-y$ between two numbers with the same sign and exponent must be computed exactly, but until that day comes the model denies to Figure A's programs any such repair as the obvious test above.

The foregoing discussion is not hypothetical nor is it a condemnation of Brown's model. Alas, his model portrays faithfully the current *Weltanschauung* for portable programs; conditional expressions can be hazardous in floating-point arithmetic. Expressions like $z \le 0.25$ suffer from double-jeopardy because 0.25 may not mean what it says and then the computer may lie about z. The expression $z \ne 0$ is ambiguous, as we have seen above under **Tests and Branches**, and should be replaced by $1 \times z \ne 0$. The expressions $x < y$ and $x-y < 0$ are exactly equivalent on some computers but not on others; on the former kind of machine, underflow when x and y are both tiny can cause the machine to deny falsely that $x < y$, and when x and y are both huge with opposite signs the expression $x < y$ may overflow and stop the machine. The expression $1 + eps = 1$ should be *true* when *eps* is negligible but may be *false* for all negative *eps* on a machine which truly chops, or *false* for all nonzero *eps* on a machine which rounds the way John von Neumann suggested, namely by forcing the last retained bit of an inexact sum or difference to be 1 and chopping the rest.

If a test like

if $|z| \le 1$ **then** ... arccos (z) ...
else ...

is unreliable, what other modification to Figure A's programs would make them run reliably and provably so on every computer? Ask a consultant skilled in error-analysis. He will observe that when z is near 1 then arccos (z) is near zero where relatively small perturbations in z cause relatively huge but absolutely small perturbations in arccos (z). Since $\Psi(t)$ involves (big) + arccos (...)^2, the perturbations don't change $\Psi(t)$ much. Therefore, the way to avoid misadventure when $|z| > 1$ is to depress $|z|$ by several ulps, something accomplished by using 4.000...005 as a divisor instead of 4 in the program psi. This works, but is a poor idea for two reasons. First, it makes psi slightly less accurate. The optimal value 4.000...005 must depend upon the characteristics of the machine in a way determined by the error-analyst to sacrifice only a little of psi's accuracy as insurance against misadventure. The machine's characteristics (radix, precision, roundoff properties, ...) are available in principle [29, 30] to portable programs, but their use introduces a new complication into Figure A and buries the distinction between portable programs and customized programs under a blizzard of environmental parameters, about which more is said below.

Secondly, there is a way to repair program psi that is simpler and can be proved within Brown's model; insert a test like this:

if $|z| < 1$ **then** ... arccos (z) ...
else ... 0 ... .

Environmental Parameters.

"Environmental Parameters" [29, 30] is the term now applied to a list of numbers that describe a computer's arithmetic to a programmer and to his program without betraying the machine's name, rank and serial number. Language implementors are being urged by some numerical analysts to set aside lists of names for these parameters so that every program may enquire of its compiler about the computer's arithmetic characteristics like

Radix,
Precision (number of significant digits carried),
Range (overflow threshold(s), underflow threshold(s)),
Roundoff properties (guard digits, which operations are chopped, etc.)

Some lists are more parsimonious, and therefore less repellent to compiler writers, than others. Some lists are obtainable, without any concessions from compiler writers, by executing ostensibly portable environmental inquiry subprograms like MACHAR [19, appendix B] which purport to discover the parameters' values at execution time by means of very devious but not entirely foolproof codes. The big question is not *whether* environmental parameters should be available, but *how* and *which*. However they become available, they cannot defeat the would-be portable programmers' worst enemy, unnecessary complexity.

A convenient way to describe environmental parameters is with the aid of the generic function [3]

Nextafter(x, y)

which stands for the first number in the computer, representable in the same format as x, after x in the direction towards y. For instance

Nextafter(1.0, 2) = 1.000...001

with as many significant digits as the computer carries for numbers near 1. For simplicity we consider only one level of precision for all floating-point expressions; otherwise we should have to distinguish single-Nextafter from double-Nextafter and so on. A few useful examples are:

Bigeps := Nextafter(1.0, 2) − 1 = 0.000...001.
Littleps := 1 − Nextafter(1.0, 0) = 0.000...0001.
B := *Bigeps*/*Littleps* = the machine's floating-point radix.
Λ := Nextafter(+∞, 1) = the overflow threshold.
λ := Nextafter(0.0, 1) = the machine's tiniest positive number.
λ/*Littleps* = a threshold commonly used to test whether underflow has had a significant effect.

At first sight these environmental parameters appear to be defined uniquely for each format, as they are in the proposed IEEE standard. However, some machines have different over/underflow thresholds for multiplication and/or division than for addition and/or subtraction. Many machines miscalculate 1.0 − Nextafter(1.0, 0) and get *Bigeps* instead of *Littleps*; a safer expression is

Littleps := (0.5 − Nextafter(1.0, 0)) + 0.5 .

Different authors define environmental parameters differently. For example, a common definition is

Eps := the smallest positive number such that the calculated value of 1.0 + *Eps* exceeds 1.0.

This *Eps* depends upon how sums are rounded or chopped;

Eps = *Bigeps*/2 if sums are rounded up,
= *Bigeps* if sums are chopped down,
= λ if von Neumann rounding is used.

Brown and Feldman [29] define ε in terms of *model* numbers;

ε := the smallest positive value of $x - 1$ for model numbers x.

Because the model numbers constitute an artfully selected and well-behaved subset of the machine's representable numbers, ε satisfies

$$\varepsilon \geq Eps \quad \text{and} \quad \varepsilon \geq Bigeps$$

but not much else can be said about its relation to the machine's actual numbers and operations.

The diversity of definitions is not what undermines environmental parameters; no matter how they may be defined their relationship with the arithmetic operations will remain enigmatic unless the parameters include information about the numbers of guard digits carried [35], whether operations are chopped or rounded or something else, and when the chopping or rounding is performed (before or after normalization), to mention only a few possibilities. Brown's model was designed in the hope that four environmental parameters would be enough for all practical purposes, but we have already seen reasons to expect otherwise and more reasons follow.

Precision and range are not absolutes that can be captured entirely by a few numbers. To illustrate why this is so, consider several different programs each using a different method to calculate the same function $\varphi(z)$ to about the same physically meaningful accuracy, say six significant decimals, over a physically meaningful range, say $|z| < 10^{10}$. Assuming speed and memory usage are about the same for all the programs, none of them is distinguishable from any other by the obvious attributes mentioned so far, but this is not to say that they are indistinguishable for numerical purposes. Imagine a larger program which includes among its numerical sub-tasks either the calculation of $\Phi = \min_z \varphi(z)$, or the solution of the equation $\varphi(z) = y$ to define the inverse function $z = \varphi^{-1}(y)$. That program will calculate divided differences

$$(\varphi(z + \Delta z) - \varphi(z)) / \Delta z$$

to determine the direction in which to pursue a search; the success of the search will depend upon the smoothness of the calculated value of $\varphi(z)$ regardless of its accuracy. Indeed, if $\varphi(z)$ is supposed to be a strictly monotonic function of z in the absence of error, then a program that computes a strictly monotonic approximation to $\varphi(z)$ correct to six significant decimals will frequently yield better results overall than a ragged approximation correct to seven. Similarly, when the search process wanders temporarily far afield producing accidentally a sample argument $z > 10^{11}$ with no physical meaning, a program that delivers a mathematically plausible value for $\varphi(z)$ will be better appreciated than one that simply stops and shouts "Overflow." In short, the quality of a program and its results depend upon the quality of arithmetic in ways that go beyond the most obvious measures of precision and range.

Diminished Expectations.

Programs conforming to a universal model that subsumes extremely diverse arithmetic engines, a model like Brown's [23] or van Wijngaarden's [26], are programs written for a hypothetical machine far worse than any ever built, a machine afflicted by the flaws of all machines subsumed under the model. Writing such programs is a painful experience which we have just sampled. Using such programs is painful too for reasons now to be explored.

The nicer properties of a carefully implemented arithmetic engine cannot be exploited by programs conforming to an abstract model of grubby arithmetic. Therefore conforming programs cannot escape entirely from performance penalties like those mentioned under **Program Libraries' Costs and Penalties**. Worse, the model's hypothetical machine may be provably incapable of computations that could be accomplished, by arcane tricks in some instances [36], on almost every real machine. Paradoxically, no problem exists that could be solved on a real machine but cannot be solved using, say, Brown's model, even though computations exist that can be performed correctly on every widely used computer but are provably impossible to perform correctly on the model's hypothetical machine. Before I explain this paradox, let me explain why it matters to me.

For the moment accept the paradox at face value; some computations are achievable nowadays on every actual machine but are provably impossible on the model's hypothetical machine, and therefore provably impossible for portable programs. Were that impossibility accepted by the computing world as a Law of Nature it would inevitably become Law legislated by atrophy. No educated customer would demand, nor would any knowledgeable programmer attempt to provide, performance generally believed to have been proved impossible. Hardware designers would be not rewarded but penalized for designing arithmetic engines any better than barely in compliance with the model, since better arithmetic would be a feature of no use to programmers who wish to produce portable (and therefore widely marketable) programs. Thus would arithmetic deteriorate until it matched the model; and numerical programming for profit would become the exclusive preserve of a priestly cadre, the devotees of the model. This is no way to convey computer power to the people.

Regardless of whether my forebodings are realistic, the paradox is real and has immediate practical consequences some of which will be exhibited after the paradox has been explained.

First, no problem exists that could be solved on a real machine but cannot be solved on a hypothetical machine that conforms to Brown's model. This is so because all of at least the first several thousand small integers are model numbers within which computation must be exact; therefore a program can be written that uses vast numbers of small integers to simulate any real computer on the hypothetical one. Brent [37, 38] has written just such a program to simulate floating-point arithmetic of any desired range and precision on any sufficiently capacious computer with a FORTRAN compiler. His program includes decimal-binary conversion for input and output, elementary transcendental functions like arccos and others not so elementary, and much more; any problem that can be solved by writing FORTRAN programs for a real computer can be solved (and more accurately) by writing FORTRAN-like programs for Brent's simulation. Of course, Brent's simulated floating-point is some orders of magnitude slower than the native floating-point of the computer on which the simulation runs, so it should not be used indiscriminately.

On the other hand consider a hypothetical machine that conforms to Brown's model [23] but is otherwise no better than it has to be. For definiteness

assume that the machine's hardware caters to just one floating-point format -- call it *working-precision*. Let z be a working-precision variable satisfying $0 < z \leq 1$. Paradoxically, there is no way for the machine to calculate accurately (to within a few ulps of their working-precision values) any of

$$\ln(z), \quad \arccos(z) \quad \text{or} \quad 1 - z$$

despite the availability of Brent's simulation and despite the fact that all of these functions can be calculated accurately on any real machine. (The last two are calculated below under **Subtraction**.)

The paradox arises because Brown's model allows these expressions to be calculated no more accurately than if they had first been replaced respectively by

$$\ln(z_1), \quad \arccos(z_2) \quad \text{or} \quad 1 - z_3$$

where z_1, z_2 and z_3 are unknown but differ from z only in its last digit(s). For instance, on a machine which carries six significant decimals but is fuzzy about the last of them, any value z between 0.999990 and 1.000000 can be replaced by any other values z_1, z_2 and z_3 in that interval whenever any arithmetic operation ($+$, $-$, $\times$, $/$) is performed with z. Therefore the hypothetical computer can produce any value between

-1.000005×10^{-5} and 0 for $\ln(z)$,
0 and 4.47214×10^{-3} for $\arccos(z)$ radians,
0 and 1×10^{-5} for $1 - z$,

without traducing the model. Brent's simulation of high-precision floating-point cannot help because first z would have to be converted, using the machine's arithmetic, from working-precision to Brent's format, but the model implies that conversion must deliver to Brent's program some unknown value z_4 no closer to z than z_1, z_2 or z_3 have to be.

After acquiescing to the indeterminate effect of model operations upon z's last digit(s), a programmer might as well choose an algorithm to calculate, say, $\ln(z)$ that really does contaminate z's last digit by roundoff on every machine, and therefore calculates $\ln(z)$ quite wrongly when z is almost 1. Why should a programmer labor to implement $\ln(z)$ accurately to within a few ulps of its value, as Cody and Waite [19, ch. 5] have shown how to do on all real machines, if that accuracy cannot be justified by the model's fuzzy view of floating-point numbers? He might as well do as Brown and Feldman [29, p. 515] have done, namely choose an algorithm that performs badly on all machines, yet no worse than the best that the model predicts for poorly designed ones. This is a pity because, the model notwithstanding, excellent results on well designed machines (whose every number is a "model number") could have been obtained from a slightly different and equally portable algorithm that does at least as well as theirs on every machine. Instead of choosing an integer k such that $f := z/2^k$ satisfies $1 \leq f < 2$, they could equally easily have chosen k so that $1/\sqrt{2} < f < \sqrt{2}$ and then calculated

$$\ln(z) := k\times\ln(2) + L((f - 0.5) - 0.5)$$

where $L(w) := \ln(1+w) = w - w^2/2 + \cdots$. The fact that $(f - 0.5) - 0.5$ is computed exactly by all machines is not provable from the model, but true nonetheless.

Perturbations in z's least significant digit appear to be unimportant when z is a computed value whose last digit is already uncertain because of earlier rounding errors; then the uncertainties in $\ln(z)$, $\arccos(z)$ and $1 - z$ caused by subsequent rounding errors cannot be much worse than what must be inherited

from before. But appearances can deceive; consider now the function

$$f(z) = \arctan(\ln(z))/\arccos(z)^2 \quad \text{if } 0 \le z < 1$$
$$= -1/2 \quad \text{if } z = 1 .$$

This function is well-behaved and comparatively insensitive to changes in z's last significant digit. Therefore the obvious program to calculate $f(z)$ is this:

```
Real function f(z): real z ;
    if z < 0 or z > 1 then protest ... invalid argument
        else if z = 1 then return -0.5
            else return arctan(ln(z))/arccos(z)**2 end f.
```

Provided arctan, ln and arccos are calculated to within a few ulps of working-accuracy, the obvious program for $f(z)$ is almost equally accurate. But if $\ln(z)$ and $\arccos(z)$ cannot be calculated that accurately, then an unobvious program must be contrived to calculate $f(z)$ accurately, perhaps by using a power series like

$$f(z) = -1/2 + (z-1)/6 - (z-1)^2/20 - 124(z-1)^3/945 + \cdots$$

when z is close enough to 1 and the obvious expression otherwise. The programmer must decide how many terms of the series to calculate, and at what threshold value of z to switch from the obvious expression to the series; the number of terms needed to achieve n correct significant decimals throughout $0 \le z \le 1$ somewhat exceeds n. When working-precision is very wide many a programmer will despair of calculating $f(z)$ to within a few ulps of working-accuracy on the hypothetical machine, declaring the task impractical; by acquiescing to less accurate results the programmer and his client acknowledge Diminished Expectations.

Of course $f(z)$ must be very special to invite accurate calculation by an obvious program despite its approach to 0/0 as $z \to 1$; some observers would describe the obvious program as a trick. Other functions just as well-behaved as $f(z)$ and even closely related, say

$$g(z) = \ln(-2f(z))/(1-z) \quad \text{if } 0 < z \le 1 ,$$
$$= 1/3 \quad \text{if } z = 1 ,$$

cannot be calculated accurately to nearly full working-accuracy by any obvious program when z is close to 1. Instead a series like

$$g(1-t) = 1/3 - 7t/45 - 1229t^2/5670 - 8699t^3/113400 + \cdots$$

has to be used when $t := 1-z$ is tiny. The trick still pays off; without it twice as many terms of the series would be needed to achieve nearly full working-accuracy in $g(z)$ throughout $0 \le z \le 1$. But describing my techniques as tricks misses the point:

> If you believe a technique cannot work,
> you will presume some task to be impractical.

Here is another example: accurate calculations of functions like

$h(x, y) = (y^x - 1)/(y - 1)$ for all $y > 0$ and all real x

are performed routinely by certain financial calculators [39]. $h(x, y)$ can be calculated accurately by a short and ostensibly portable program [40, p. 216] which merely assumes exponentials, logarithms, products, quotients and differences are accurate to nearly full working-precision. The program is short but not obvious; it circumvents the 0/0 problem when $y \to 1$ in a way that fails when $\ln(z)$ and $1-z$ are no more accurate than is possible on the hypothetical

machine. The modality of Brown's model supplies no incentive to look for *that* program, less to think it might work. Had the calculators' designers been in the habit of thinking only along the lines of the model, every accurate algorithm they devised would have overflowed the space available for microcode, and somebody would have had to choose between producing inaccurate calculators or none.

The most elementary tasks are vulnerable to Diminished Expectations if they must be programmed too portably. Take the solution of a quadratic equation

$$ax^2 - 2bx + c = 0$$

given working-precision values of the coefficients a, b, c. Can't the roots be calculated correct to within a few ulps of working-precision from a well-known formula? Yes and No. The calculation is possible for an ostensibly portable program which, by making the fullest use of Environmental Parameters (q.v. ab.ve), will work correctly on every North American mainframe. But if the program must be rigorously portable over all the machines covered by Brown's model, and hence over the hypothetical machine, then the best that can be done is this [36]:

> Except for over/underflow, the calculated roots differ each by at most a few ulps from corresponding exact roots of some unknown quadratic whose coefficients differ each by at most a few ulps from the respective given coefficients.

This kind of accuracy specification, generally associated with *backward error-analysis*, is not the easiest kind to understand. It implies for the quadratic equation that as many as half the figures carried may be lost when the roots are nearly coincident.

To calculate the roots to nearly full instead of half working-precision, it suffices that a program evaluate the discriminant b^2-ac as accurately as if it were first evaluated to double-working-precision and then rounded back to working-precision. This can be achieved by ostensibly portable and truly efficient programs published by Dekker [41] which represent each double-working-precision value as a sum of two working-precision numbers, but the programs malfunction on a few families of machines with insufficiently meticulous arithmetic. Whether a package of programs like Dekker's could be devised to work on every real machine is not yet known. No such package could possibly work on the hypothetical machine; on the other hand, every mainframe machine built so far has been found susceptible to precision extension even if only via a non-portable program calibrated to that machine. What makes the quadratic equation solvable by a program which is portable *de facto* but not *de jure* is a loophole; if the discriminant is positive it need only be evaluated at first to a little less than full double-working-precision [42, §7], and then only if b^2 and ac mostly cancel. By combining Dekker's techniques with another [36, p. 1233] that provokes cancellation of errors, a single program can be devised that solves quadratic equations to nearly full working-precision on all the machines I know, though verifying that the program works must be done very differently on different machines. This program is ugly and too slow to be practical. A practical program similar to the one in Figure 10 (to be discussed later) does exist; it is much faster than indiscriminate double-working-precision because it uses only a few of Dekker's techniques, and it is portable over most machines, but it fails on a few commercially important machines. Perhaps those few should be excluded from the purview of portable programming, though such a stigma

would surprise their designers and purchasers who had believed, in good company, that allowing arithmetic to be a bit fuzzy would obscure just the last digit or two, not half of the digits carried.

Among the casualties of Diminished Expectations are our mental paradigms. For instance, the term *ill-conditioned* is applied to a problem whose answer's accuracy deteriorates drastically when certain kinds of data for the problem are very slightly in error. The archetypal example is matrix inversion; the inverse A^{-1} of a nearly singular matrix A is hypersensitive to tiny perturbations in A. The roots of a quadratic equation are hypersensitive to tiny perturbations in the coefficients when the roots are nearly coincident. These problems are ill-conditioned regardless of the program used to solve them, regardless of whether the program calculates the answers correctly or not in the face of roundoff. Backward error-analysis [43, 36] is a paradigm intended to explain (but not excuse) the effects of roundoff in some programs and is regarded as successful whenever

> the program's results differ by at most a few (specified) ulps from the exact results belonging to a similar problem with data differing by at most a few (specified) ulps from the data actually given.

This explains successfully why a matrix inversion program should be expected to produce a poor inverse A^{-1} when applied to a nearly singular, and hence ill-conditioned, matrix A. Only if the program's calculated A^{-1} is much worse than could be caused by a few ulps' perturbation in A -- as might happen if the program neglects to perform pivotal interchanges -- would the program be condemned as *numerically unstable* and supplanted by a good program drawn from LINPACK [17]. Numerical instability is not necessarily fatal; if working-precision sufficiently exceeds what might have been thought warranted by the accuracy of the data and desired results, then an ostensibly unstable program may deliver eminently satisfactory results fast despite losing most of the figures it carries. On the other hand, solutions correct to within an ulp or two can be calculated economically for many an ill-conditioned problem, including the examples $\ln(z)$, $\arccos(z)$, $1-z$ and the quadratic equation discussed above; these accurate solutions are easier to understand and apply than inaccurate solutions excused by ill-condition. Thus, the epithets *ill-conditioned* and *unstable* have come not so much to characterize problems and programs as to reflect our attitudes towards them. From a paradigm created to explain erstwhile perplexing numerical phenomena, backward error-analysis has evolved into an exculpatory mechanism for any programmer who believes, sometimes correctly, that his portable program does about as well as is practical even if it does badly.

Diminished Expectations circumscribe range as well as precision. The hypothetical machine of Brown's model becomes unpredictable (it may stop) when overflow occurs, so programs should not allow that to happen unnecessarily; if a problem's data and solution both lie within range then ideally the program should not allow overflow to obstruct its progress. But the only way to avert overflow is to test the magnitudes of operands against preselected thresholds before every operation that cannot be rendered safe *a priori* by the introduction of apt scale factors (choices of units). This strategy is consistent with John von Neumann's antipathy to floating-point arithmetic [44, §5.3] but he was much better at mental analysis than the rest of us for whom the strategy is usually impractical. Nowadays most programmers take perfunctory precautions if any, leaving pre-scaling and other defenses against over/underflow to the care of their program's users [17, p. 15]. In the light of this policy let us consider so

simple a computation as the scalar product of two vectors,

$$\langle U, V\rangle := U_1V_1 + U_2V_2 + \cdots + U_nV_n \ .$$

It is clearly unreasonable to test before every multiply and add lest it over/underflow, so the user must be obliged to choose scale factors σ, τ in advance and calculate $\langle \sigma U, \tau V\rangle = \sigma\tau\langle U, V\rangle$ for subsequent unscaling. But scaling *a priori* might be impractical too. Try it on the expression

$$q := \frac{\langle A, B\rangle \langle C, D\rangle}{\langle A, D\rangle \langle C, B\rangle}$$

allowing for the possibility that the vectors may resemble these:

$$A := (\Lambda, a, a, \cdots, a, a, 0)$$
$$B := (0, b, b, \cdots, b, b, \Lambda)$$
$$C := (\Lambda, c, c, \cdots, c, c, 0)$$
$$D := (0, d, d, \cdots, d, d, \Lambda)$$

where Λ is the overflow threshold and a, b, c, d are not much bigger than the underflow threshold. After getting $q = 0/0$ because all underflowed products ab, cd, ad, cb were flushed to zero by the machine, try to find scale factors α, β, γ, δ which will permit

$$q := \frac{\langle \alpha A, \beta B\rangle \langle \gamma C, \delta D\rangle}{\langle \alpha A, \delta D\rangle \langle \gamma C, \beta B\rangle}$$

to be calculated correctly ($q = 1$) despite underflow and without overflow. No such scale factors exist.

In general, scale factors that avert overflow without stumbling over something else tend to be unnecessary (1) for most data but hard to find when needed, if they exist at all. Programmers incline neither to waste time looking for them nor to encumber programs with them when they are obvious but tedious. Therefore the customary portable program for calculating q is the obvious program, and it sacrifices about three quarters of the machine's exponent range to achieve practical portability; the program may malfunction if the vectors' nonzero elements are bigger than the fourth root of the overflow threshold or tinier than the fourth root of the underflow threshold. This Diminished Expectation is practically unavoidable unless the computer resumes calculation after overflow and allows the program subsequently to determine whether overflow occurred, as do many computers today and as is required by the proposed IEEE standard, in which case programs like those in Figures 8 and 9 (discussed below) become practical.

The lesson is clear. Regardless of what may be provably *impossible*, currently available arithmetic engines are so diverse that provably portable programming is *impractical* unless we agree either

i) to expect significantly poorer accuracy, range and/or speed from portable programs than we expect from customized programs, or

ii) that portable programs are expected to work correctly on most machines but not all, although we cannot easily ascertain which machine deserves more than certain others to be odd man out.

Programmers vs. Users vs. Computer Salesmen

Rather than waste human time and talent worrying about computers' precision, range and/or speed, the wisest policy is to purchase more precision, range, speed and memory than are needed and then squander the surplus, provided we do not squander so much as to forego the competitive advantage conferred by a powerful computer. But how much do we need? The quadratic equation and other more important examples suggest that twice as much precision is needed for intermediate calculations as is meaningful in data and results; the quotient q of scalar products above suggests that range requirements should be quadrupled. Might other examples inflate our perceptions of our needs even more? Until we understand better what can and can't be done with the equipment we have already, and who is responsible for decisions about what is feasible, and until we understand better which limitations are Laws of Nature and which are due to ignorance and indifference, we cannot say with confidence how much more is needed nor by whom. Until then, decisions about computer purchases, hardware and software, will rely more upon salesmanship than upon informed judgment.

Buying a bigger computer is the answer to a question that is too often asked too late, namely shortly after a new machine has been purchased. Before the purchase, all concerned entertain optimistic hopes and promises. Then the programmer has to realize them. I am not certain that programmers are the only ones who have to reconcile ill-defined tasks to ill-behaved machines, but certainly the programmer must be among the first to suspect that the task assigned to him might be impractical and to wonder whether he can persuade his management that this is so. The programmer cannot just buy a bigger computer, neither real nor hypothetical.

We ought to be more considerate of programmers, especially if we plan to flood the market with computers intended for users among whom almost none yearn to program. Those users cannot see the computer as it really is but rather see a portrait painted over it by programmers. If programmers have to redecorate a grubby machine, real or hypothetical, they may weigh it down with too many layers of paint. For instance, consider an engineer who has to cope with data and results to 3 or 4 significant decimals; experience teaches him to calculate with at least 7 or 8 significant decimals, so he looks for applications programs and computer systems that will guarantee that much accuracy. The applications programmers, asked to guarantee the correctness of 7 or 8 significant decimals, demand 14 or 15 from their computer system and its supporting library. The library programmers play it safe too; they ask for 30. Computer architects obligingly offer 33. The engineer who accepts this rationale may unknowingly be squandering 3/4 of his system's data memory and 9/10 of its speed as well as the opportunity to obtain a satisfactory system on one chip. I think I have exaggerated the situation here, but only to emphasize how heavily important decisions will hang upon what the user and various programmers believe to be practical.

The IEEE subcommittee that drafted the proposed standard had the programmer uppermost in mind when it abandoned the path beaten by most previous committees concerned with floating-point, and chose a different approach albeit not unprecedented [45]. A draft standard under which all commercially important arithmetic engines could be subsumed was not attempted. Numerical programs were not surveyed statistically to accumulate operation counts, nor were questionnaires mailed out. Instead, the subcommittee tried to understand how arithmetic engines have evolved up to now, why programs have been written the way they are, what programmers and their clients need and wish to

accomplish, and how best that might be done taking into account costs that must be borne by users as well as implementors. A few subcommittee members scrutinized innumerable numerical programs — real ones too, not just artificial examples like some in this paper. Most of the important discrepancies among diverse arithmetic engines were traced to accidental differences, some of them exceedingly minute.

Subtraction

Current computers and calculators perform floating-point subtraction in diverse ways that occasionally produce peculiar results. To simplify the presentation of some of the anomalies let us restrict attention to decimal floating point arithmetic with operands and results carrying 4 significant decimals. One example is $(3.414\times10^{0}) - (7.809\times10^{-3}) = 3.406191\times10^{0} \rightarrow 3.406\times10^{0}$ to 4 significant decimals, calculated as shown in Figure 1 to get a slightly different result 3.407×10^{0}.

$$\begin{bmatrix} 3.414\times10^{0} \\ -7.809\times10^{-3} \end{bmatrix} = \begin{bmatrix} 3.414 \times10^{0} \\ -0.007809\times10^{0} \end{bmatrix} \rightarrow \underline{\begin{bmatrix} 3.414\times10^{0} \\ -0.007\times10^{0} \end{bmatrix}}$$
$$3.407\times10^{0}$$

Figure 1

Many machines, as illustrated in Figure 1, shift the tinier operand to the right enough to equalize exponents but, in doing so, retain no more decimal digits of the tinier operand than will line up under the 4 significant decimals of the bigger operand. The tinier operand's excess digits are discarded (chopped off) before the subtraction is carried out to produce a final result that appears, in Figure 1, to be not too bad. A slightly better result might be expected if, instead of chopping off the excess digits, the machine were designed to round them off as in Figure 2.

$$\begin{bmatrix} 3.414\times10^{0} \\ -7.809\times10^{-3} \end{bmatrix} = \begin{bmatrix} 3.414 \times10^{0} \\ -0.007809\times10^{0} \end{bmatrix} \rightarrow \underline{\begin{bmatrix} 3.414\times10^{0} \\ -0.008\times10^{0} \end{bmatrix}}$$
$$3.406\times10^{0}$$

Figure 2

Indeed, the result there is as good as can be expressed in 4 significant decimals, so the process illustrated in Figure 3, which rounds off not the subtrahend but just the final difference, might seem to be not worth the bother of carrying an extra digit or two during the subtraction plus the extra work to round off the final result.

$$\begin{bmatrix} 3.414\times10^{0} \\ -7.809\times10^{-3} \end{bmatrix} = \begin{bmatrix} 3.414 \quad \times10^{0} \\ -0.007809\times10^{0} \end{bmatrix} = \left[\begin{array}{c} 3.414000\times10^{0} \\ -0.007809\times10^{0} \\ \hline 3.406191\times10^{0} \end{array}\right] \rightarrow 3.406\times10^{0}$$

Figure 3

But appearances can deceive. Before passing judgment look at a few complete programs rather than merely a few calculations.

The subprograms in Figure 4 show how arcsin(*z*) and arccos(*z*) might have to be calculated in a computing environment which, like the earliest FORTRAN dialects, comes with no inverse trigonometric function besides arctan.

```
Real function arcsin(z): real z ;
      if |z| > 1 then Exit with "Invalid Operand" message;
      if |z| = 1 then return sign(z)×(π/2)
            else return arctan(z/√((1−z)×(1+z))) end arcsin.
Real function arccos(z): real z ;
      if |z| > 1 then Exit with "Invalid Operand" message;
      if |z| + 0.125 = 0.125 then return π/2;
      y := arctan(√((1−z)×(1+z))/z);
      if y < 0 then return π + y else return y end arccos.
```

Figure 4

What do these subprograms produce for arcsin(*z*) and arccos(*z*) when $z = 9.999\times10^{-1}$? The results depend crucially upon the value calculated for $1-z$ which depends in turn upon how subtraction is performed. Let us apply each of the methods illustrated by Figures 1 to 3 in turn.

Subtraction performed like Figure 1 calculates 1.000×10^{-3} for $1-z$ as shown in Figure 1A. Subsequent calculation on such a machine yields $\arcsin(z) \rightarrow 1.526\times10^{0}$ and $\arccos(z) \rightarrow 4.468\times10^{-2}$.

$$\begin{bmatrix} 1.000\times10^{0} \\ -9.999\times10^{-1} \end{bmatrix} = \begin{bmatrix} 1.000 \ \times10^{0} \\ -0.9999\times10^{0} \end{bmatrix} \rightarrow \frac{\begin{bmatrix} 1.000\times10^{0} \\ -0.999\times10^{0} \end{bmatrix}}{0.001\times10^{0}} = 1.000\times10^{-3}$$

Figure 1A

Results are harder to predict for a machine that subtracts the way illustrated in Figure 2. If such a machine recognizes that $|9.999\times10^{-1}| < 1$, as seems reasonable, then it must calculate 0 for $1-z$ as shown in Figure 2A. Consequently $\arcsin(9.999\times10^{-1})$ encounters division by zero and delivers no predictable result, and $\arccos(9.999\times10^{-1}) \rightarrow 0$ on this machine.

$$\begin{bmatrix} 1.000\times10^{0} \\ -9.999\times10^{-1} \end{bmatrix} = \begin{bmatrix} 1.000\times10^{0} \\ -0.9999\times10^{0} \end{bmatrix} \rightarrow \frac{\begin{bmatrix} 1.000\times10^{0} \\ -1.000\times10^{0} \end{bmatrix}}{0.000\times10^{0}} = 0$$

Figure 2A

$$\begin{bmatrix} 1.000\times10^{0} \\ -9.999\times10^{-1} \end{bmatrix} = \begin{bmatrix} 1.000\times10^{0} \\ -0.9999\times10^{0} \end{bmatrix} = \begin{bmatrix} 1.0000\times10^{0} \\ -0.9999\times10^{0} \\ \hline 0.0001\times10^{0} \end{bmatrix} = 1.000\times10^{-4}$$

Figure 3A

Subtraction performed the third way is a little slower but produces an error-free difference $1-z = 1.000\times10^{-4}$ as shown in Figure 3A, and then leads to calculated values

$\arcsin(9.999\times10^{-1}) \rightarrow 1.557\times10^{0}$ and $\arccos(9.999\times10^{-1}) \rightarrow 1.414\times10^{-2}$

which compare favorably with the correct values:

$\arcsin(0.9999) = 1.556654...$ and $\arccos(0.9999) = 0.01414225...$.

Despite their anomalies, the first two kinds of subtraction are common among computers and calculators, defended by their designers with various arguments. One argument points out that $1.000\times10^{0} - 9.999\times10^{-1}$ involves "massive cancellation" and consequent "loss" of significant digits. A related argument observes that 9.999×10^{-1} could easily be in error by a unit or so in its last (fourth) significant decimal; should no more be known about z than that

$$9.998\times10^{-1} < z < 1.000\times10^{0}$$

then nothing more is worth saying about $\arcsin(z)$ and $\arccos(z)$ than that

$$1.55079 < \arcsin(z) < 1.57080 \quad \text{and} \quad 0.0200004 > \arccos(z) > 0.$$

These arguments will not be pursued here because they give so little satisfaction even to their proponents, especially considering that neither of the first two subtraction methods yields final results satisfying the last two inequalities. More satisfaction can be realized by substituting the esoteric subexpression

$$((0.5-z) + 0.5)\times((0.5 + z) + 0.5)$$

for $(1-z)\times(1+z)$ in Figure 4's subprograms, after which all three styles of computation will yield results very nearly correct to all 4 significant decimals despite "massive cancellation", but the modified subprograms are trickier to explain.

Little tricks here and there add up to complicated programs. The more complicated the program, the more vulnerable it is to blunders and the more it must cost to develop. Conversely, a programming environment free from *unnecessary* anomalies, entailing fewer tricks, must entail lower programming costs. By specifying that subtraction be performed as accurately as possible, and consequently exactly whenever massive cancellation occurs, the proposed

IEEE standard increases slightly the cost of implementing arithmetic in the hope that programmers can then waste less time on arithmetic trickery.

Of course, no standard can protect programmers from inaccuracies caused by their choice of unstable numerical methods. For instance, a common mistake is to use the equation

$$\arccos(z) = \frac{\pi}{2} - \arcsin(z)$$

as an algorithm to define arccos(z); this equation would produce $\arccos(9.999\times10^{-1}) \rightarrow 1.571 - 1.557 = 0.014 = 1.400\times10^{-2}$ instead of the correct value 1.414×10^{-2}. Another common mistake is to replace $((1-z)\times(1+z))$ in Figure 4 by the simpler expression $(1-z^2)$; this can lose almost half the significant decimals carried, as it does when $z = 9.968\times10^{-1}$ because

$$1 - z^2 = 1 - (.99361024) \rightarrow 1 - .9936 \rightarrow 0.0064 = 6.400\times10^{-3}$$

instead of

$$(1 - z)\times(1 + z) = (0.0032)\times(1.9968)$$
$$\rightarrow (3.200\times10^{-3})\times1.997 = 6.3904\times10^{-3} \rightarrow 6.390\times10^{-3}.$$

How should a programmer know which of these expressions make good programs and which bad? Under the proposed standard the programmer knows that massive cancellation during subtraction introduces no new rounding errors but may reveal errors inherited by the operands. Therefore differences like $(1 - z)$ between exact constants and given data are safe to use, whereas differences like $1 - z^2$ or $\frac{\pi}{2} - \arcsin(z)$ involving rounded values may be inaccurate. Without the standard no such simple rule can be trusted, and then the programmer may have to resort to tricks like $((0.5 - z) + 0.5)\times((0.5 + z) + 0.5)$ which work well enough but have to be explained differently and with difficulty for every different style of arithmetic.

Symbols and Exceptions.

The proposed IEEE standard goes far beyond specifying small details like rounding errors. It also specifies how to cope with emergencies like 1/0, 0/0, $\sqrt{-3}$, exponent over/underflow, etc. Lacking such specifications, the subprograms in Figure 4 have been encumbered by tests like

if $|z| > 1$ **then** ... to avoid $\sqrt{}$(negative number),
if $|z| = 1$ **then** ... to avoid $\pm1/0$, and
if $|z| + 0.125 = 0.125$ **then** ... to avoid 1/0 or overflow of $1/z$ when $|z|$ is very tiny.

But the proposed standard specifies rules for creating and manipulating symbols like ±0, $\pm\infty$ and *NaN* – the symbol "*NaN*" stands for "Not a Number". These rules are designed so that a programmer may frequently omit tests and branches that were previously obligatory because computers treated exceptions in unpredictable or capricious ways. For instance, the proposed standard's rules for signed zero and infinity, $1 - 1 = +0 = \sqrt{+0}$, $+1/(\pm0) = \pm1/(+0) = \pm\infty$ respectively, and the rule that approximates overflowed quantities by $\pm\infty$ with the appropriate sign, allow the subprograms in Figure 4 to be simplified substantially provided the arctan program recognizes that $\arctan(\pm\infty) = \pm\pi/2$ respectively. The subprograms in Figure 5, when they are run in the proposed standard's *Normalizing Mode* (about which more later), deliver the same numerical values as do the more cumbersome subprograms in Figure 4.

```
Real function arcsin(z): real z:
      return arctan(z / √((1−z)×(1+z))) end arcsin.
Real function arccos(z): real z:
      y := arctan(√((1−z)×(1+z))/ z);
      if y < 0 then return π + y    else return y end arccos.
```

Figure 5

For invalid arguments ($|z| > 1$) the subprograms in Figure 4 deliver an error message and then wrest control of the computer away from the program that called arcsin(z) or arccos(z), thereby presumably aborting its execution. The subprograms in Figure 5 do not have to abort; instead when $|z| > 1$ they may deliver the value *NaN*, created by √(negative number) and propagated through arctan(*NaN*) → *NaN*, and then resume execution of the calling program. At the same time as *NaN* is created a flag called *Invalid Operation* is raised. Subsequently the calling program may infer either from this flag or from the *NaN* that an emergency arose and may cope with it automatically rather than abort. This capability, to cope automatically rather than just stop and complain, might be important in a program being executed far away from human supervision, possibly in an unmanned spacecraft surveying Mars. Of course, if the programmer fails to anticipate and provide for that emergency then, according to the proposed standard's rules by which *NaN*s propagate themselves through arithmetic operations, the program's final output may be *NaN*. A humane computing system might recognize when a *NaN* is being emitted to a human and convey additional retrospective diagnostic information like

> this *NaN* is descended from √(negative number)
> attempted in subprogram "arcsin".

The proposed standard allows (but does not oblige) such information to be encoded in a *NaN*.

Alternatively, when debugging programs that call the subprograms in Figure 5, the programmer may have specified in advance (by *enabling* the *Invalid Operation* trap) that invalid operations terminate execution with a message like, say,

```
Invalid operand outside domain of operation "square root"
invoked in line      1 of subprogram      "arcsin"
invoked in line     23 of subprogram      "triangle"
invoked in line      5 of subprogram      "survey"
...                  .. ...               ...
```

Note that no special provision to allow or suppress such messages has to be inserted into Figure 5's subprograms. The proposed standard specifies just the arithmetic aspects of the programming environment, and does so independently of whether the operating system is helpful enough to supply those messages when it aborts execution. A more explicit message, say

> subprogram "triangle" knows no plane triangle has sides
> 3.075, 19.62, 2.041

might be more help to whoever must debug "survey", but that message is the responsibility of whoever programmed "triangle". What the proposed standard does provide is that means must exist whereby "triangle" can be programmed to

discover whether it is being executed with the *Invalid Operation* trap enabled, in which case its message may emerge, or disabled, in which case only *NaN* need emerge when things go wrong.

Flags and Modes.

The convenience afforded by symbols like $\pm\infty$ and *NaN*, and by flags like *Invalid Operation* and *Divide by Zero*, does not come for free. Besides the significant cost of embedding the rules for symbols and flags in the arithmetic's implementation, there is a considerable cost in developing software that will interpret correctly these symbols as inputs and will raise flags only when necessary.

Let us see what Figure 5's subprograms do to flags. Because those subprograms are so brief and natural, we can verify easily that they do produce either a satisfactory numerical result or a deserved *NaN* for *every* input z, be it finite, infinite or *NaN*. But when $|z| = 1$ the *Divide by Zero* flag will be raised by $\arcsin(z)$ even though it returns $\pm\pi/2$ correctly. The same flag will be raised when $\arccos(0)$ delivers $\pi/2$. Rather than distract other users of his program by raising flags unnecessarily, the conscientious programmer should restore irrelevantly raised flags to their prior states. Figure 6 below will show how the statements needed to restore flags constitute a brief prologue and epilogue in each program without intruding upon the program's algorithm.

In addition to flags, the proposed standard has had to introduce things called *Modes* to reflect the truism that exceptions are exceptional just because no universally satisfactory way to cope with them can exist. Fortunately, each class of exceptions admits only a few reasonable responses, and the proposed standard encompasses most of those in its Modes. For instance, the *Affine* and the *Projective* modes provide the two environments in which arithmetic with infinity has been found useful. The Affine mode respects the sign of $\pm\infty$ so that $-\infty < (\textit{any finite number}) < +\infty$, and $(+\infty) + (+\infty) = (+\infty)$ but $(+\infty) - (+\infty)$ is invalid (*NaN*); this mode is apt when $+\infty$ was created by overflow or by division by zero resulting from underflow. The Projective mode treats the sign of ∞ as misinformation created, perhaps, by evaluating $1/(x-y)$ in one place and $-1/(y-x)$ in another both with $y=x$, and regards as invalid (*NaN*) all expressions "$\infty + \infty$" and "$\infty - \infty$", and raises the *Invalid* flag for ordered comparisons like "$\infty < x$" and "$\infty \geq x$" both of which are called *false* for every finite x. Because the Projective mode is the more cautious of the two, it is the default mode in which programs lacking any contrary directives are presumed to be executed. It is the apt mode for calculating rational functions in complex arithmetic. Since most programs, like those in Figure 6, work the same way in both modes, most programmers will safely ignore both modes.

Other modes in the standard control the direction and precision of roundoff, enable and disable optional traps for handling exceptions, and mediate the effects of gradual underflow [46]. The two modes associated with gradual underflow are the *Warning* and *Normalizing* modes of which the latter is invoked in Figure 6, so these two modes will be described here.

Gradual Underflow

Underflow is what happens to numbers too tiny to be represented in the normal way. Consider for instance a decimal machine carrying four significant decimals, with two-digit exponents spanning the range between ± 99 inclusive. Such a machine allows magnitudes between 1.000×10^{-99} and 9.999×10^{99} to be represented normally but needs special symbols for 0 and possibly ∞. Something special is needed for underflowed magnitudes between 0 and 1.000×10^{-99}

too; nowadays most machines either flag underflows as ERRORs and STOP, or else flush them to 0 as if they were negligible despite that sometimes, we know, they are not. The proposed standard underflows *gradually* by using *denormalized numbers*, in this instance ranging from 0.001×10^{-99} up to 0.999×10^{-99}, to approximate underflowed magnitudes. These denormalized numbers, distinguishable from other numbers only by their minimal exponent -99 and leading digit 0, simplify certain programs by helping to preserve relationships [46] that cannot survive when underflows are flushed to 0. For example "$x > y$" should imply "$x - y > 0$", and does so despite roundoff and underflow when underflow is gradual; but examples like

$$x = 3.414\times10^{-99} > y = 3.402\times10^{-99}$$

suffer when $x - y = 0.012\times10^{-99}$ underflows and is flushed to 0 or brings computation to a stop.

Gradual underflow incurs an error no bigger than a rounding error 0.0005×10^{-99} in the smallest normalized number 1.000×10^{-99} on our decimal machine; the corresponding magnitudes on a binary machine conforming to the proposed standard are respectively $2^{-150} = 7.\times10^{-46}$ and $2^{-126} = 1.\times10^{-38}$ in single precision. A programmer will ignore these errors if he is satisfied that they do no more damage than roundoff, as is almost always true; and then he will ignore as well the standard's underflow flag which is raised whenever underflow occurs. But sometimes the flag must not be ignored, as is the case for the following example:

$$q = (a\times b)/(c\times d) = (a/c)\times(b/d) = (a/d)\times(b/c).$$

Which of the three expressions should be used to compute q?

Whichever one be chosen, values a, b, c, d can be encountered that overwhelm the chosen expression with overflow or underflow despite that some other expression would have produced q correctly. For instance, evaluating all three expressions on our four significant decimal machine with $a = 8.100\times10^{-51}$, $b = 1.800\times10^{-32}$, $c = 6.000\times10^{-50}$, $d = 1.670\times10^{-50}$ produces

$$\frac{a\times b}{c\times d} = \frac{1.458\times10^{-102}}{1.002\times10^{-99}} \quad \cdots \text{ underflowing gradually and raising flag}$$

$$\rightarrow \frac{0.001\times10^{-99}}{1.002\times10^{-99}} \rightarrow 9.980\times10^{-4}\ ,$$

$$\frac{a}{c}\times\frac{b}{d} \rightarrow (1.350\times10^{-1})\times(1.078\times10^{-2}) \rightarrow 1.455\times10^{-3} \quad \text{correctly.}$$

$$\frac{a}{d}\times\frac{b}{c} \rightarrow (4.850\times10^{-1})\times(3.000\times10^{-3}) = 1.455\times10^{-3} \quad \text{correctly.}$$

Gentle or not, underflow is too dangerous here to ignore. A conscientiously written program, after calculating q from one expression, must test the *Overflow* and *Underflow* flags and then substitute when necessary whichever (if any) other expression for q can be evaluated without raising a flag. Such a program is called *robust* insofar as it copes with avoidable exceptions automatically.

Underflow's Normalizing vs. Warning Modes.

The foregoing discussion outlined what the proposed standard provides for underflow in its *Normalizing* mode; it merely raises the *Underflow* flag whenever a denormalized number or 0 is created to approximate a number tinier than the

tiniest normalized number. Rather than acquiesce to the consequences of that error, a program may respond to a raised flag by branching to an alternative procedure. But what about a program written in ignorance of the *Underflow* flag and its related two modes? Lacking any reference to either mode, the program would be expected to be executed in the standard's default mode, which is the *Warning* mode. This mode treats as invalid any attempt to divide by a denormalized number or to magnify it greatly by a multiplication or division. Consequently the Warning mode would produce a *NaN* and raise the *Invalid Operation* flag rather than produce the dangerous value 9.980×10^{-4} for $q = (a\times b)/(c\times d)$ above. If the *Invalid Operation* trap were not enabled the program's user would subsequently have to investigate why his program

> propagated a *NaN* into its output,
> raised the *Invalid Operation* flag, and
> raised the *Underflow* flag.

After educating himself about underflow he would contemplate his options:

> Ignore or change the data that induced the exception, or
> Condemn the program and return it to its author, or
> Revise the program to calculate q some other way, or
> Institute the Normalizing mode and re-run the calculation.

The last option makes sense when, as happens frequently, analysis reveals that q is destined to be added to some value sufficiently bigger than q that its figures lost to underflow cannot matter much.

Exceptions Deferred.

The proposed standard's treatment of underflow illustrates its pragmatic approach to an elusive and controversial objective:

> to specify for each kind of exception an automatic response that is economical, coherent, more likely useful than punitive, provably no worse than what has been done in the past, and no serious impediment to programs that may have to respond in another way.

Both underflow-handling modes permit programs as well as programmers to defer judgment about the seriousness of an underflow. An analysis of numerical programs shows why this deferral is so valuable; most underflows are rendered harmless when denormalized because they are destined to be added into accumulations of like or larger magnitudes, (matrix calculations) or to be further diminished by subsequent multiplications (polynomial evaluation) or divisions (continued fractions). Harmful underflows tend to draw attention to themselves when subsequent results that should be finite and not zero turn out otherwise. Other harmful underflows, like the dangerously wrong value 9.980×10^{-4} for q above, are captured in the Warning mode which, when it would be too pessimistic, can be over-ridden by the Normalizing mode as is done in Figure 6. A few kinds of harmful underflow will remain elusive as ever, known only by the flag they raise.

The proposed standard's treatment of exceptions has been elected because it appears, on balance, to be better than any known alternative although certainly not foolproof. The only foolproof way to cope with exceptions is to abort computation whenever one occurs, but that does not deliver correct results at all.

```
... Generic subprograms to calculate arcsin(z) and arccos(z)
... ... using throughout the same precision as has been declared for z.
... ... (Insert precision declarations for z, y, π, arctan, arcsin, arccos.)
... Domain: valid for −1≤z≤1.
... ... Invalid arguments, including ∞ in both modes (Affine/Projective),
... ... produce NaN if the Invalid Operation trap is disabled, otherwise
... ... precipitate an Invalid Operation Exception " √ (Negative Number)".
... Accuracy: within a few units in the last significant digit delivered.
... Subprogram used: arctan(z), assumed correct to within a few
... ... units in its last significant digit for all arguments z including very
... ... tiny and denormalized z for which arctan(z) = z, and
... ... arctan(±∞) = ±π/2 respectively, and arctan(NaN) → NaN,
... ... all without exceptions.
...
      arcsin(z): Save & Reset Divide by zero Flag to off;
            y := arctan(z / √((1−z) × (1+z)) );
            Restore saved Flags;
            return y end arcsin.
      arccos(z): Save & Reset Divide by zero, Overflow Flags to off;
            Save Warning/Normalizing Mode & Set to Normalizing;
            y := arctan(√((1−z) × (1+z))/ z );
            if y < 0 then y := y + π ;
            Restore saved Flags, Modes;
            return y end arccos.
```

Figure 6

Arcsin and Arccos.

Figure 6 illustrates how the conscientious programmer might produce impeccable and portable subprograms despite a potentially bewildering diversity of options and exceptions recognized by the proposed standard. He does not have to change Figure 5's algorithms; he does not have to devise nor explain arcane tests like the one in Figure 4, i.e.

if $|z| + 0.125 = 0.125$ **then** ···

Instead he includes in his program or documentation just those statements pertinent to the selection of options and responses to exceptions that concern him. He need not know about any option he does not exercise nor about any exception his program will not precipitate. Those aspects of the programming environment mentioned by him explicitly are thus proclaimed to be the aspects for which he has assumed responsibility; at least he has thought about them.

Opportunity vs. Obligation.

The more that can be done, the more will be expected and the more will be attempted. In this respect the proposed standard is not an unmixed blessing for software producers. Besides having to unlearn tricks that should never have had to be learned, programmers will have to think, lest they become distracted by an embarrassment of riches, about which of the enhanced capabilities are worth exercising. Three related cases in point are the solution of a quadratic equation and the scalar product

$$<U, V> := U_1V_1 + U_2V_2 + \cdots + U_nV_n$$

discussed above under **Diminished Expectations**, and the root-sum-squares

$$Rtsmsq(n, V) := \sqrt{<U, V>}$$

discussed under **Precision and Range** earlier. How should these functions best be calculated under the proposed standard?

Unquestionably the best way to calculate $sum := <U, V>$ when the proposed standard's optional *Extended* format is available is with the obvious program:

Real Extended *sum*; *sum* := 0 ;
for $j = 1$ **to** n **do** *sum* := *sum* + $U[i]\times V[i]$.

(Should the higher-level language processor not know that the product $U[i]\times V[i]$ must be evaluated to the extra range and precision afforded by the Extended format, some assembly-language program may have to be used instead.) This obvious program illustrates, by contrast with the programs in Figures 7, 8 and 9 to be discussed below, the best reason for implementing the Extended format:

> By far the easiest errors and exceptions to cope with are the ones we know can't happen.

Especially on parallel pipelined vectorized machines, where n may be immense and/or exceptions impossible to trap, the Extended format is worthwhile even if confined to a relatively small number of (vector and scalar) registers. But some computer and language architectures will not tolerate yet another kind of floating-point data type, so we must reconsider the calculation of scalar products and root-sum-squares in an environment supporting only one floating-point format.

Previously published programs [17, 22, 23, 24] for $Rtsmsq(n, V)$ were burdened by constraints of which only two need be reconsidered now:

I) Avoid scanning the array V more than once because, in some "virtual memory" environments, access to $V[i]$ may cost more time than a multiplication.

II) Avoid extraneous multiplications, divisions or square roots because they may be slow. For the same reason, do not request extra precision nor range.

And those programs overcame only one of the two hazards, roundoff and over/underflow, succumbing to the former:

> When n is huge (10^6) but the precision is short (6 significant decimals) then $Rtsmsq(n, V)$ may be badly obscured by roundoff amounting to almost $n/2$ ulps.

Moreover, the programs are unsatisfactory for a vectorized machine with a parallel pipelined arithmetic engine because they entail several non-trivial branches within their inner loops. Attempts to generalize these *Rtsmsq* programs to cope with scalar products compound this last complication beyond practicality.

The programs in Figures 7 to 10 are practical solutions for the foregoing problems. The programs use two functions recommended widely [3, 19, 29] to cope with scaling problems and available under diverse names; one is

$$scalB(z, n) := z\times B^n$$

where B is the machine's radix (2 for the proposed standard), and the function over/underflows only if the product would otherwise be finite, nonzero and out of

range. The second function is the integer valued

$$n := \mathrm{logB}(z) \quad \text{that satisfies } z = \mathrm{scalB}(f, n) \text{ where } 1 \le |f| < B$$

provided a finite such integer n exists; otherwise

$$\mathrm{logB}(0) = -\infty \quad \text{and} \quad \mathrm{logB}(\pm\infty) = +\infty \ .$$

These functions or their near equivalents are achievable universally via little more than integer arithmetic upon the exponent field of the floating-point argument z; without them neither $\exp(z)$ nor $\ln(z)$ could be computed economically. Should $\mathrm{scalB}(z, n)$ be slower than a multiplication, replace it by $z \times S$ inside a loop having first computed the scale factor $S := \mathrm{scalB}(1, n)$ outside that loop; such an S will not over/underflow in the programs of Figures 7-9.

The programs in Figures 7 to 10 achieve their accuracy by exploiting a technique akin to some used by Dekker [41] to extend the available precision, but faster, though not fast enough to be worth using in all instances; this feature is easy to remove by following the comments at the programs' ends.

The programs are practical, but program *Dotprod* in Figure 8 is unreasonable; each of its inner loops is too much slowed down by a test for overflow that could branch out of the loop. This program, derived from *Rtsmsq* in Figure 7 via a few modifications, illustrates the danger in unthinking acceptance of a prior constraint, namely constraint I. Scalar products are far more common than root-sum-squares, and far less likely to suffer over/underflow. Therefore *Dotprod* should be programmed as in Figure 9 to maximize its speed in the usual case and accept a modest penalty, retardation by a factor of 2 to 5 overall, in the rare event of over/underflow. Then the program to calculate the root-sum-squares might as well be this:

Real function Rtsmsq(n, V) : **integer** n; **real array** $V[n]$;
    integer k; **real** *sum*;
    Dotprod(n, V, V, *sum*, k) ; **Return** scalB($\sqrt{}$(*sum*), $k/2$) **end**.

In the special case $n=2$, important for complex absolute value

$$\mathrm{cabs}(x + iy) = \sqrt{x^2 + y^2} \ ,$$

better and shorter programs have been presented elsewhere [47].

Figure 10 exhibits a program which solves the quadratic

$$ax^2 - 2bx + c = 0$$

correct to working precision without using Extended nor Double precision. It scales to preclude premature over/underflow, and tests for pathologies like $a=0$ to detect infinite or indeterminate roots when appropriate. The program achieves full robustness by saving and restoring flags and modes in appropriate places, thereby ensuring the appropriate flag will be raised only when a root overflows, underflows, is infinite because of division by $a=0$, or is invalid ($0/0$, ∞/∞, or dubious because of denormalized coefficients in the Warning mode). The program is complicated most of all by an accurate calculation of the discriminant $d := b^2 - ac$ using simulated nearly-double-precision arithmetic whenever this is necessary. Consequently the accuracy does not deteriorate when roots are nearly coincident. The same goal is achievable by far simpler programs when either an Extended format [5, pp. 19-20] or double-precision [36, p. 1222] is available. Alternatively, when the loss of half the figures carried would be tolerable, the subroutine *redoD* in Figure 10 can be thrown away.

Programmers always have to juggle trade-offs: Speed, Memory, Generality, Simplicity of use, Reliability, Delay in development. Accuracy, Range and Robustness lengthen this list for numerical software. The perplexities of these

trade-offs cannot be eliminated by the proposed IEEE arithmetic standard; it may make them worse. Whereas fuzzy arithmetic vitiates certain kinds of close analysis, thereby relieving the programmer from an obligation to analyze too closely, the proposal's precise specifications raise hopes that careful analysis will be rewarded at least occasionally by significant performance enhancements. Whether these enhancements be won easily with the aid of an Extended format, or won deviously without that format, enhanced performance will stimulate enhanced expectations. Thus are obligations born of opportunity.

Conclusion.

Alas, the proposed IEEE standard for binary floating point arithmetic will not guarantee correct results from all numerical programs. But the standard weights the odds more in our favor. It does so by meticulous attention to details that hardly ever matter to most people but matter, when they do matter, very much. That attention to detail has been found by several implementors of this standard to cost tolerably little extra compared with the intrinsic cost of meeting any standard. And the expected benefits go beyond the presumed benefits conferred by *any* standard, namely that in identical circumstances we all enjoy or suffer identical consequences. The benefit of careful attention to detail is that consequences follow from what we have done rather than from what has been done to us.

Acknowledgements

Thanks are due to my colleagues and friends, especially J.T. Coonen, S.I. Feldman and B.N. Parlett, for helpful comments on several preliminary drafts of this document. Preparation was supported in part by research grants from the U.S. Office of Naval Research, contract N00014-76-C-0013, and from the U.S. Department of Energy, contract DE-AT03-76SF00034. What first lent credibility to the scheme that has evolved into the IEEE proposal was J.F. Palmer's work developing the INTEL i8087.

References

[1] O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press, New York, 1972, pp. 15-16.

[2] W.S. Churchill, a note to General Ismay, Dec. 6, 1942; cited on p. 793 of *The Hinge of Fate*, Bantam Books, New York, 1962.

[3] "A Proposed Standard for Binary Floating-Point Arithmetic," Draft 8.0 of IEEE Task P754, with an introduction by D. Stevenson and appraisals by W. J. Cody and D. Hough, *Computer*, 14 no. 3, March 1981.

[4] J.T. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *Computer*, 13 no. 1, Jan. 1980, pp. 68-79; updated in March, 1981.

[5] *ACM SIGNUM Newsletter*, Special Issue, Oct. 1979 ...contains an earlier draft 5.11 of [3] plus commentary and a counter-proposal.

[6] "The 8086 Family User's Manual Numerics Supplement - July 1980," Intel Corp. manual no. 121586 ... for the i8087 Numeric Data Processor on a chip.

[7] R. Capece, J.G. Posa, "16-bit Makers Solidify Positions," in *Electronics*, May 8, 1980, pp.89-90 ...pre-announces Motorola's 6839 and 68341 floating-point ROMs.

[8] J. Palmer, R. Nave, C. Wymore, R. Koehler, C. McMinn, "Making Mainframe Mathematics Accessible to Microcomputers," in *Electronics* May 8, 1980, pp. 114-121 ...about the i8087.

[9] S. Bal, E. Burdick, R. Barth, D. Bodine, "System Capabilities get a Boost from a High-Powered Dedicated Slave," in *Electronic Design*, Mar. 1, 1980 ...about the one-chip 16081 floating-point slave processor that has all but square root, binary decimal-conversion and gradual underflow on chip.

[10] S. Cheng, K. Rallapalli, "Am9512: Single Chip Floating-Point Processor" in session 14 "Hardware Alternatives for Floating-Point Processing" of *1980 Electro Professional Program*, Proceedings of the 1980 Electronic Convention held in Boston, May13-15, 1980 ...uses the proposed standard's single and double precision formats but not its exception handling.

[11] Session 18, "Floating-Point Standards for Micros and Minis" of *1980 Electro Professional Program*, referred to in [10].

[12] "Statement on Proposals for IEEE Standard for Micro-Processor Floating-Point" prepared by IFIP Working Group WG2.5 on Numerical Software, Dec. 16, 1978 in Baden Austria; communicated to Richard Delp, then chairman of the IEEE/CS subcommittee, by Prof. C.W. Gear.

[13] Letter dated Dec. 14, 1979, to Richard Delp, then chairman of the IEEE/CS subcommittee, from Prof. L.D. Fosdick, chairman of the board of SIGNUM, the ACM Special Interest Group on NUmerical Mathematics.

[14] "The IMSL Library Contents Document, Edition 8; FORTRAN Subroutines in the areas of Mathematics and Statistics," brochure put out by International Mathematical & Statistical Libraries, Inc., Houston, Texas 77036.

[15] ALGOL 68 and FORTRAN Libraries of numerical subroutines are available from the Numerical Algorithms Group Ltd., Oxford OX26NN, England.

[16] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, C.B. Moler, "Matrix Eigensystem Routines – EISPACK Guide" 2nd ed., no. 6 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1976 ...FORTRAN programs.

[17] J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, *LINPACK Users' Guide*, SIAM (Soc. for Industrial & Applied Math), Philadelphia, 1979 ...FORTRAN programs.

[18] C.L. Lawson, R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, N.J. ...FORTRAN programs.

[19] W.J. Cody Jr., W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs, N.J. ...sin, cos, log, exp, ...

[20] W.J. Cody Jr., "The FUNPACK package of special function subroutines" in *ACM Transactions on Mathematical Software*, 1 (1975) pp. 13-25.

[21] Phyllis A. Fox, ed., *The PORT Mathematical Subroutine Library User's Manual*, Bell Labs, 1976; available from Computing Information Service, Bell Labs, Murray Hill, N.J., 07974.

[22] J.L. Blue, "A Portable Program to Find the Euclidean Norm of a Vector," *ACM Transactions on Mathematical Software*, 4, pp. 15-23, 1978.

[23] W.S. Brown, "A Simple But Realistic Model of Floating-Point Computation," Computer Science Technical Report no. 83, May 1980, revised Nov. 1980; Bell Labs, Murray Hill, N.J., 07974.

[24] M.G. Cox and S.J. Hammarling, "Evaluation of the language Adá for use in numerical computations," NPL Report DNACS 30/80, July 1980; National Physical Laboratory, Teddington, England, TW110LW.

[25] R.P. Brent, "On the Precision Attainable with Various Floating-Point Number Systems," *IEEE Transactions on Computers*, C-22, pp. 601-07, 1973.

[26] A. van Wijngaarden, "Numerical Analysis as an Independent Science," *BIT*, 6, pp. 66-81, 1966.

[27] W.S. Brown, "Some Fundamentals of Performance Evaluation for Numerical Software," pp. 17-29 in *Performance Evaluation of Numerical Software*, L.D. Fosdick, ed., North Holland Publishing Co., Amsterdam, 1979.

[28] W.S. Brown, "A Realistic Model of Floating-Point Computation," pp. 343-360 in *Mathematic Software III*, J.R. Rice, ed., Academic Press, New York, 1977.

[29] W.S. Brown and S.I. Feldman, "Environment Parameters and Basic Functions for Floating-Point Computation," *ACM Transactions on Mathematical Software*, 6, pp. 510-523, 1980.

[30] W.J. Cody, "Software Basics for Computational Mathematics," §III and references cited therein, ACM-SIGNUM Newsletter, 15, no. 2, June 1980, pp. 18-29.

[31] B.T. Smith, "A Comparison of Two Recent Approaches to Machine Parameterization for Mathematical Software," pp. 15-17, *Proceedings of a Conference on the Programming Environment for Development of Numerical Software* co-sponsored by JPL and ACM-SIGNUM, Oct. 18-20, 1978; JPL publication 78-92, Jet Propulsion Lab, Pasadena CA, 91103.

[32] N.L. Schryer, "UNIX as an Environment for Producing Numerical Software," pp. 18-22 of the proceedings cited in [31].

[33] W.R. Cowell, L.D. Fosdick, "Mathematical Software Production," pp. 195-224 in the book cited in [28].

[34] Cray-1 Computer System Hardware Reference Manual 2240004, 1979, Cray Research Inc., Mendota Hts., MN, 55120.

[35] P.H. Sterbenz, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[36] W. Kahan, "A Survey of Error Analysis," pp. 1214-1239 of *Information Processing 71*, North Holland Publishing Co., Amsterdam, 1972.

[37] R.P. Brent, "A Fortran multiple-precision arithmetic package...MP, Algorithm 524," *ACM Transactions on Mathematical Software*, 4 (1978), pp. 57-81.

[38] R.P. Brent, J.A. Hooper, J.M. Yohe, "An AUGMENT Interface for Brent's Multiple Precision Arithmetic Package," *ACM Transactions on Mathematical Software*, 6 (1980), pp. 146-149.

[39] R.E. Martin, "Printing Financial Calculator Sets New Standards for Accuracy and Capability," *Hewlett-Packard Journal*, 29 no. 2 (Oct. 1977), pp. 22-28.

[40] W. Kahan, B.N. Parlett, "Können Sie sich auf Ihren Rechner verlassen?" *Jahrbuch Überblicke Mathematik* 1978, pp. 199-216, Bibliographisches Institut AG, Zurich; translated into German by W. Frangen from the report "Can You Count on your Calculator," Memorandum no. UCB/ERL M77/21 of April 6, 1977, Electronics Research Lab, University of California, Berkeley, California, 94720.

[41] T.J. Dekker, "A Floating-Point Technique for Extending the Available Precision," *Numerische Mathematik*, 18 (1971), pp. 224-242.

[42] D. Hough, ed., *Implementation of Algorithms, Parts I & II*, notes taken by W.S. Haugeland and D. Hough of lectures by W. Kahan, University of California, Computer Science Dept. Technical Report 20, 1973; available from NTIS as item DDC AD 769 124/9 GA.

[43] J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, no. 32 of the National Physical Laboratory's *Notes on Applied Science*, Her Majesty's Stationery Office, London, 1963.

[44] A.W. Burks, H.H. Goldstine, J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," (1946) reprinted in *The Origins of Digital Computers – Selected Papers*, ed. by B. Randall, Springer-Verlag, New York, 1975.

[45] F.N. Ris, "A Unified Decimal Floating-Point Architecture for the Support of High-Level languages (Extended Abstract)," pp.18-23 of *ACM SIGNUM Newsletter*, 11 no.3, Oct. 1976; also pp.21-31 of *ACM SIGARCH Computer Architecture News*, 5, no. 4, Oct. 1976.

[46] J.T. Coonen, "Underflow and the Denormalized Numbers," *Computer*, 14 no. 3, March 1981.

[47] W. Kahan, "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard," pp. 99-128 of *Interval Mathematics 1980*, proceedings of a symposium held in Freiburg i.B., Germany, in May 1980; edited by K. Nickel, Academic Press, New York, 1980.

Real Function *Rtsmsq*(n, V) : $\cdots := \sqrt{(\sum_{1}^{n} V[j]^2)}$

```
Integer n; Real Array V[n];
... Roundoff cannot accumulate beyond a few ulps of Rtsmsq.
... No flags are changed unless Rtsmsq must over/underflow.
... ... To avoid premature overflow or loss of accuracy to drastic underflow,
... ... the sum of squares is scaled internally by B**2k just when necessary;
... ... B is the radix, here 2, to avoid injecting extra roundoff.
... ... For speed's sake the array V is scanned only once,
... ... but this program is unsuitable for pipelined parallel vectorized machines.
      Integer i, j, k, m; Boolean Flags Overflow, Underflow ... flags signal events.
      Real sum, round, term, temp  ... same precision & range as V, Rtsmsq.
      Constant i, m;  ... inserted by hand or at compile time using formulas *.
      Save Warning/Normalizing Mode & Set to Normalizing;
      Save Affine/Projective Mode & Set to Affine  ...respect sign of ∞.
... *  round := 1-Nextafter(1, 0) .  = 1 ulp of 0.999...  < 1/n.
... *  temp := Nextafter(+∞, 0) ... = overflow threshold.
... *  m := logB(round/√(temp)) ...  B**m ≈ round/√(temp) to scale down.
... *  term := Nextafter(0, 1)/round ...normalized underflow threshold ×2.
... *  i := (logB(temp)-logB(term)+logB(round))/2 ...  to scale up.
...
      Save & Reset Overflow, Underflow Flags to false;
      j := 1; sum := round := 0; k := i;  go to "Scaled" ...scale up first.
...
      "Huge": Reset Overflow Flag to false; k := m ...to scale down.
            sum := scalB(sum, m+m); round := scalB(round, m+m);
      "Scaled": for j = j to n do begin ...calculate scaled sum of squares.
                  term := round + scalB(V[j], k)**2;
                  temp := term + sum  ...add (V[j]×B**k)**2 to sum.
                  if Overflow then go to "Ordinary"...and record j.
                  round := (sum-temp)+term; sum := temp end j;
            Restore saved Flags, Modes; Return scalB(√(sum), -k)  ...unscaled
...
      "Ordinary": sum := scalB(sum, -i-i); round := scalB(round, -i-i);
            Reset Overflow Flag to false;
            for j = j to n do begin ...try an unscaled sum of squares.
                  term := round + V[j]**2; temp := term + sum
                  if Overflow then go back to "Huge"...and record j.
                  round := (sum-temp)+term; sum := temp end j;
            Restore saved Flags, Modes; Return √(sum)
...
... ... Optimization for small n or large tolerance for roundoff: This program
... ... spends time recalculating round to compensate for most
... ... rounding errors caused by addition and succeeds unless n > 1/round
... ... initially, which is all but inconceivable. But if an error as big as n/2 ulps
... ... of Rtsmsq is tolerable then speed can be improved slightly by omitting
... ... recalculation of round and term, as if round = 0.
      end Rtsmsq.
```

Figure 7

```
Subroutine Dotprd(n, U, V, sum, k) : ... sum := (Σ_1^n U[j]×V[j])/B**k
Input Integer n; Input Real Array U[n], V[n];
Output Integer k; Output Real sum ... B = radix, 2 for IEEE standard.
... Scaling (k≠0) is invoked only if necessary to avoid over/underflow; k is even.
... Accuracy is no worse than if every U[j] and V[j] were perturbed by less than 2 ulps.
... For speed's sake the arrays are each scanned only once, but this program
... is unsuited to pipelined parallel vectorized machines.
      Integer i, j, m; Boolean Flags Overflow, Invalid, Underflow;
      Real round, term, temp  ...same precision & range as sum, U, V.
      Constant i, m; ...inserted by hand or at compile time using formulas *.
      Save Warning/Normalizing Mode & Set to Normalizing;
      Save Affine/Projective Mode & Set to Affine  ...respect sign of ∞.
... *  round := 1−Nextafter(1, 0) ... = 1 ulp of 0.999...  < 1/n.
... *  temp := Nextafter(+∞, 0) ... = overflow threshold.
... *  m := logB(round/√(temp)) ...  B**m ≈ round/√(temp) to scale down.
... *  i := logB(temp×round×round) ... to scale up.
...
      Save & Reset Overflow, Underflow, Invalid Flags to false;
      j := 1; sum := round := 0; k := i;  go to "scaled"  ...scale up first.
...
      "Huge": Reset Overflow Flag to false; k := m ...to scale down.
            sum := scalB(sum, m+m); round := scalB(round, m+m);
      "Scaled": for j = j to n do begin ...calculate scaled sum of products.
                  term := round + scalB(U[j], k) × scalB(V[j], k);
                  temp := term + sum  ...add (U[j]×V[j]) × B**2k to sum.
                  if not Overflow then begin round := (sum−temp)+term;
                                    sum := temp end not Overflow;
                        else if not Invalid then go to "Ordinary" ...& record j.
                                    else Reset Overflow, Invalid Flags to false;
                  end j;
            k := −k−k; Reset Underflow Flag to false;
            temp := scalB(sum, k) ...attempt to unscale.
            if not (Overflow or Underflow) then begin
                        k := 0; sum := temp end;
            Restore saved Flags, Modes; Return ... possibly k≠0.
...
      "Ordinary": sum := scalB(sum, −i−i); round := scalB(round, −i−i);
            Reset Overflow Flag to false;
            for j = j to n do begin ...try to accumulate unscaled products.
                  term := round + U[j]×V[j]; temp := term + sum;
                  if Overflow then go back to "Huge" ...and record j.
                  round := (sum−temp)+term; sum := temp end j;
            Restore saved Flags, Modes; Return √(sum) ...k=0 as usual
...
... ... Optimization for small n or large tolerance for roundoff: This program
... ... spends time recalculating round to compensate for most rounding errors
... ...  caused by addition and succeeds unless n > 1/round initially, which is all but
... ... inconceivable. But if perturbations of about n/2 ulps in every U[j] and V[j]
... ... are tolerable then speed can be gained by omitting to recalculate
... ... round and term, as if round = 0.
      end Dotprd.
```

Figure 8

```
Subroutine Dotprd(n, U, V, sum, k) : ...  sum := (Σ_1^n U[j]×V[j])/B**k
Input Integer n; Input Real Array U[n], V[n];
Output Integer k; Output Real sum ... B = radix, 2 for IEEE standard.
... Scaling (k≠0) is invoked only if necessary to avoid over/underflow; k is even.
... Accuracy is no worse than if every U[j] and V[j] were perturbed by less than 2 ulps.
... If over/underflow requires it, the arrays may be scanned twice. This program
... is satisfactory for pipelined parallel vectorized machines because tests
... in the inner loop do not have to wait for any arithmetic operation to finish.
      Integer i, j, m; Boolean Flags Overflow, Underflow ...flags signal events.
      Real round, savesum, term, temp, thresh  ...precision & range of sum, U, V.
      Constant i, m, thresh; ...inserted by hand or at compile time using formulas *.
      Macro procedure Add(z): ...to be compiled in-line, rather than called...
            term := round + z; temp := sum + term; round := (sum-temp) + term;
            sum := temp end Add ... more accurate than sum := sum + z.
      Save Warning/Normalizing Mode & Set to Normalizing;
      Save Affine/Projective Mode & Set to Affine ...respect sign of ∞.
... *  round := 1-Nextafter(1, 0) ... = 1 ulp of 0.999...  < 1/n.
... *  temp := Nextafter(+∞, 0) ... = overflow threshold.
... *  m := logB(round/√(temp)) ... B**m ≈ round/√(temp) to scale down.
... *  thresh := Nextafter(0, 1)/round ....normalized underflow threshold ×2.
... *  i := logB(temp×round×round) ... to scale up.
...
      Save & Reset Overflow, Underflow Flags to false;
            sum := round := k := 0; ...try to do without scaling.
            for j = 1 to n do Add(U[j] × V[j]);
      if Overflow then begin ...scale down.
            sum := round := 0; k := m;
            for j = 1 to n do Add(scalB(U[j], k) × scalB(V[j], k));
                  end Overflow;
          else if Underflow and |sum| < n×thresh then begin
            savesum := sum; sum := round := 0; k := i ...try to scale up.
            for j=1 to n do if U[j]≠0 and V[j]≠0
                  then Add(scalB(U[j], k) × scalB(V[j], k));
            if Overflow then begin sum := savesum; k := 0; end can't scale
                  end Underflow;
      if k≠0 then begin k := -k-k  ...try to unscale.
            Reset Overflow, Underflow Flags to false;
            savesum := scalB(sum, k);
            if not (Overflow or Underflow) then begin ...successfully unscaled.
                  k := 0; sum := savesum  end
                  end k≠0;
      Restore saved Flags, Modes; Return;
...
... ... Optimization for small n or large tolerance for roundoff: This program
... ... spends time recalculating round to compensate for most rounding errors
... ... caused by addition and succeeds unless n > 1/round initially, which is all but
... ... inconceivable. But if perturbations of about n/2 ulps in every U[j] and V[j]
... ... are tolerable then speed can be gained by omitting to calculate
... ... round and term in Add, as if round = 0.
      end Dotprd.
```

Figure 9

```
Subroutine Quadratic(a, b, c, D, X, Y) : ...solve ax^2 - 2bx + c = 0.
Input real a, b, c; Output real D, X, Y;
... If D ≥ 0, real roots X and Y satisfy |X| ≮ |Y|; otherwise the complex conjugate
... roots are X ± iY. Each of X and Y is accurate within a few ulps unless it lies out
... of range, in which case it over/underflows. Infinite, zero or denormalized a, b, c
... produce X, Y compatible with modes inherited from the calling program.
Integer m; Boolean Flags Overflow, Underflow; Real A, B, C, P, R;
Integer constant L, l; ... inserted by hand or at compile time.
L := logB(Nextafter(∞, 0)) ... (radix)^(L+1) barely overflows.
l := logB(Nextafter(0, 1)/(Nextafter(1, 2) - 1)) ... (radix)^l almost underflows.
... Test for pathological coefficients first.
if a=0 and c=0 then goto bigB else D := 1;
if a=0 or c is not finite then begin Y := (c / b)/ 2; X := c / a; return end;
if c=0 or a is not finite then begin X := (b / a)/ 0.5; Y := c / a; return end;
... Henceforth a and c are finite and nonzero.
m := (l + L - max{l, logB(a)} - max{l, logB(c)}) / 2 ...rounded to integer.
A:=scalB(a, m); C:=scalB(c, m); P:=A×C;
Save & Reset Overflow, Underflow Flags to false;
Save Warning/Normalizing Mode & Set to Normalizing;
B:=scalB(b, m); R:=B×B; D:=R-P;
if D = R then begin ... B^2 >> |A×C| ≈ 1 .
        Restore saved Flags, Modes;
    bigB: D := 1; Y := (c / b)/ 2; X := (b / a)×2; return end;
if -R≤D≤P then redoD ... see internal subroutines below.
if D<0 then begin ... complex conjugate roots.
        Restore saved Flags, Modes; Y := √(-D) / A; X := (b / a); return end;
... now roots must be real.
R := B + copysign(√D, B) ... = B + √D × sign(B);
Restore saved Flags, Modes; Y := C/ R; X := R/ A; return;

Internal Subroutine redoD: ... recalculate small D = B^2-AC.
... Omit redoD if half-precision accuracy is acceptable for nearly coincident roots.
... Otherwise, by simulating nearly-double-precision arithmetic, the following program
... calculates D correct to working-precision, despite cancellation. Consequently the
... calculated roots will be correct to nearly full working-precision. And when the roots
... would be small integers if calculated exactly, they will be calculated exactly in
... binary, but perhaps only approximately when the radix exceeds 2.
real Ā, ā, B̄, b̄, C̄, c̄, p, r, E, e, F, f, G, g;
m := (logB(A) - logB(C))/2 ... rounded to integer.
split(scalB(A, -m), Ā, ā) ... see split subroutine below.
split(scalB(C, m), C̄, c̄); split(B, B̄, b̄);
p := ((Ā×C̄ - P) + (ā×C̄ + Ā×c̄)) + ā×c̄ ... = A×C - P exactly.
r := ((B̄×B̄ - R) + 2×b̄×B̄) + b̄×b̄ ... = B×B - R exactly.
E := R-P ; e := ((E+P)-R) + p ...E-e = R-P-p
F := E-e ; f := ((F+e)-E) - r ...F-f = E-e+r
G := F-f ; g := (G+f)-F ...G-g = F-f
D := G-g; return ...D = R-P-p-r = B×B - A×C.
... ...
Internal Subroutine split(Z, H, t) : Input real Z; Output real H, t;
... H := (Z rounded to half working-precision) and t := Z-H exactly provided
... that the radix is 2 or else working-precision carries an even number of digits.
real constant J := 1 + scalB(1, integer nearest ((0.5 - logB(Nextafter(1, 2) - 1))/2)) ;
H := J×Z; H := H-(H-Z); t := Z-H; return end split; end redoD; end Quadratic
```

Figure 10.