INTERVAL ARITHMETIC OPTIONS IN THE PROPOSED IEEE FLOATING POINT ARITHMETIC STANDARD

William M. Kahan

Electrical Engineering
& Computer Sciences Department
University of California
Berkeley, California

I. ABSTRACT

Implementors of the proposed standard, described in a special issue of the SIGNUM Newsletter (1) and in an article by J. Coonen in the IEEE Journal "COMPUTER" (2), are encouraged to provide two "directed rounding modes" by which the endpoints of intervals may be rounded outward automatically without unnecessarily spreading degenerate (one-point) intervals. This feature alone should bring the cost of interval arithmetic down by a substantial factor. Moreover, the stan-

Proceedings of a Symposium on
INTERVAL MATHEMATICS
held in Freiburg i.B., W. Germany
May 1980

ISBN 0-12-518850-1

dard provides symbols for infinity and specifies default responses to over/underflow in ways intended, among other things, to satisfy the needs of interval arithmetic. Finally, the standard is being implemented, options and all, by more than one major microcomputer manufacturer, so we will all get a chance soon to try it out.

II. INTRODUCTION

This paper's title is ambiguous. Does it refer to opportunities afforded users of the Standard to practice Interval Arithmetic? Does it refer to Interval Arithmetic's influence upon the architects of that Standard? Both questions are discussed below. But first the Standard itself must be described a little.

Which Standard? There have been several proposals before the IEEE/CS working group concerned with floating point arithmetic, but only three have been persistent and, of those, only one has consistently attracted majority support within the working group (5). That one is the KCS proposal described in most detail by J. Coonen (2) and in least by W. Kahan and J. Palmer (1). Furthermore, only the KCS proposal is being implemented by semiconductor manufacturers (3); here is a partial list in rough chronological order:

AMD 9512	uses single and double precision formats while omitting almost all exception handling, several operations, and most options including directed roundings, but is being sold widely at time of writing.
Intel 8087	does everything in the KCS proposal and more on one chip and fast (< 40 μsec), but only with 8086 or 8088 processors; due in late 1980, see (8).
Motorola 6839	firmware in a ROM to do all of KCS upon a 6809 processor; due in 1980-81, see (4).
National 16081	slave processor for use with 16032 processor to do most of KCS, including directed roundings, on one chip and fast (< 20 μsec), the rest via software; due by 1982. Can be used alone, see (3).

Many other implementations of all or part of the KCS proposal on boards, in software or in firmware exist or have been announced. The only other proposal for a standard to have attracted a manufacturer's support is the PS proposal which is intended to be implemented partially, and without directed roundings for the forseeable future, on the DEC VAX family; see Payne and Bhandarkar (9). The third proposal, FW, differs from the others mainly in its intrusion of two symbols to stand for the intervals containing numbers too big or too tiny to be represented in the normal way; though seemingly attractive at first sight, this proposal has repelled implementors. All

proposals specify formats for floating point numbers so that radix, precision and range will be defined adequately for equipment manufacturers as well as programmers. Typical binary formats are these:

Name	Word width	Sign	Biased Exponent	Precision
Single	32 bits	1 bit	8 bits	1+23 bits
Single extended	$\geq$ 44	1	$\geq$ 11	$\geq$ 32
Double	64	1	11	1+52
Double extended	$\geq$ 80	1	$\geq$ 15	$\geq$ 64
Quad	128	1	15	112 or 1+112

The designation "1+23 bits" means that the leading significant bit is implicit and not stored. The designation "$\geq$ 32" means that the format's specification includes only a lower bound for range and precision; these extended formats, allowed only in the KCS and FW proposals at the implementor's option, provide an inexpensive but limited way to suppress roundoff and over/underflow in intermediate results. The KCS scheme uses its formats' smallest exponents to represent "Denormalized numbers" (about which more later) and zero. Its largest exponent is reserved for $\pm\infty$ and for the NaNs; a NaN is Not-a-Number that participates in arithmetic merely by replicating itself. Various NaNs may be used, at the implementer's option, either as pointers to retrospective diagnostic information or, when trapped, to provide arithmetic extensions beyond the specifications of the standard.

All proposals specify performance standards for operations; KCS does so for add, subtract , multiply, divide, remainder (used for argument reduction for trigonometric functions and the exponential), square root, comparison, conversions between various formats (single, double, extended, integer, decimal). All proposals say something about exceptions; here KCS is particularly demanding. Every exception, when it occurs, must raise a flag that a program may subsequently sense and/or reset, perhaps long afterwards. Also every exception, including INVALID, must deliver by default a result specified by the standard in a way that is, we hope, reasonable if not universally acceptable. Traps, supplied at the implementor's option, provide a program the means to over-ride the defaults wherever something else is better.

All proposals specify rounding by default to be round-to-nearest, with the ambiguous case rounded to nearest even. This means that, if we were rounding numbers to 4 sig.dec., every x in $3.1395 \leq x \leq 3.1405$ would round to 3.140

$3.1405 < x < 3.1415$	3.141
$3.1415 \leq x \leq 3.1425$	3.142
$3.1425 < x < 3.1435$	3.143
...	...

As defaults go, this rounding is unimprovable. However, the implementor may, at his option, supply directed roundings: Round toward 0, Round toward $+\infty$, Round toward $-\infty$ which a program may then preselect for any operations that produce rounded results. These directed roundings constitute the only options in the standard designed specifically to

facilitate Interval Arithmetic. The directed roundings are integrated with the exception handling; for instance, a positive sum, product or quotient that overflows when being rounded towards $+\infty$ will actually be rounded to $+\infty$ without raising a flag (that would be redundant). Subsequently $+\infty$ will participate in arithmetic in a reasonable way (e.g. $-3.0/(+\infty) = -0$) until something unreasonable like $\pm\infty/\infty$ creates a NaN and raises the flag INVALID. The signed ± 0 and $\pm\infty$ symbols allow for intervals which may or may not include 0 or ∞ as an endpoint; for instance $[-0,1]$ includes 0 but $[+0,1]$ does not. Of course, ∞ could lie inside an interval too, as in $[1,-1] = \{x: 1 \leq x \text{ or } x \leq -1 \text{ or } x = \infty\}$, see Caplat's abstract in these proceedings or Laveuve (7). However, many applications of Interval Arithmetic remain viable even if ∞ never occurs inside an interval, and then implementation is easier.

Here ends the discussion of features in the standard intended to promote Interval Arithmetic. Henceforth we discuss how its feasability influenced the design of the KCS proposal.

III. CONTROVERSY AND MISCONCEPTIONS

On reflection I am apalled at the intensity of controvery and even acrimony that has been generated during discussions of the proposed standard. I can only conclude that many people are burdened by misconceptions about the KCS proposal and

about other topics, among them

- the conflict among Safety, Utility and Cost.
- the role played by Interval Arithmetic in numerical software.
- the role of Error Analysis in numerical software.
- the principal contributors to the cost of numerical software.
- the assignment of blame for malfunctions of numerical software.

Rather than treat Floating Point Arithmetic on a strictly mathematical and technical basis, I propose to venture into psychology and economics and other superstitions in an attempt to expose misconceptions, some of them perhaps mine, and lay them to rest.

A. The Complexity Issue

Yes, the KCS proposal looks complicated to implement. But so does an automatic transmission look more complicated than a manual transmission, yet makes driving a car easier, and at relatively small extra cost. Since KCS is known to cost not much more to implement than alternatives, what remains to be proved is its impact upon the costs and performance of numerical programs. Its advocates expect old programs to run at least as well on KCS as on previous arithmetic systems, and evidence is mounting to justify that expectation. Therefore the critical questions concern new programs.

I claim that the cost of writing a new numerical program,

given the mathematical algorithm, is proportional mostly to the incidence of

- Subroutine calls
- Comparisons, tests, thresholds
- Jumps, branches, cases.

In other words, the "straight-line code" that evaluates mathematical expressions contributes relatively weakly to programming costs, especially when compiled from higher-level languages that allocate storage automatically. Therefore good design must attempt to trade off greater complexity in the underlying implementation of arithmetic (or software) against enhanced simplicity of use as revealed by greater generality, a larger domain of application, or fewer cases to test, fewer branches to think about. This trade-off implies an awareness of the purposes served by floating point arithmetic, purposes whose diversity is complicated by the sometimes paradoxical styles of approximate reasoning in a domain where "One man's Negligible is another man's All".

B. The Importance of Being Accurate

Regardless of whether John Rice and I and others, who claim that floating point computation is far more nearly ubiquitous than most computer users and builders think, are correct, correct computation does not matter much. If most of the numerical results printed or displayed matter at all, why are

they so soon discarded, often without being seen? And if numerical results don't matter, neither do their errors. Besides, nobody believes that high-rise buildings will topple into underflows nor that aircraft will fold their wings in flight over rounding errors; everybody who deals with numbers daily distrusts them. We expect numerical errors to occur and to make themselves known to the skeptical onlooker before they can do irreparable damage.

The foregoing paragraph is not a parody; it merely reflects an attitude both prevalent and obsolescent. Of course nobody can object to incorrect results until he discovers they are incorrect; then what does he do? He may ignore them, or correct them; or he may look for somebody to blame, preferably somebody else.

As long as most consumers of numerical results were the principal authors of their own programs, they had nobody else to blame except in rare instances, and then the fellowship of our guild precluded blaming one another except for the most flagrant negligence. But times change. The consumers of numerical results are increasingly dependent upon software vendors, upon professional programmers who sell their wares without communicating more than superficially an understanding of their product. It must be done this way or else we would all drown in the details we pay someone else to dispatch. Moreover, proprietary interests are increasingly often defended by providing software in a form that can, for a fee, be executed but not read, thereby denying purchasers any opportunity to

verify the suitability of the program for their purposes even when they are willing to wade through the details. Therefore numerical software must increasingly come to be specified not by a list of its instructions but by its vendor's representations, and equally so must the vendor bear the blame for misrepresentations.

Just as doctors and hospitals practice now a kind of "defensive medicine" in jurisdictions where malpractice suits are common, so shall "defensive programming" become more common, and with the same inflationary impact upon costs, despite attempts to define and share risks. We are moving into an era where the correctness of results is invisible and only the correctness of programs matters.

C. Our Faith in Interval Arithmetic

Many of the earliest and still most enthusiastic advocates of Interval Arithmetic acclaim its infallibility; it will not give an answer that is wrong without a warning, namely a wide interval, that the answer is ill determined. But who would rather have a warning than an answer accurate enough? If the objective were merely to compute correct errorbounds we could all save money by computing "∞". The proper goal of a numerical procedure is an estimate whose error is negligible regardless of how big or small that error may be. This goal cannot possibly be accomplished by using only Interval

Arithmetic; we all know this, and therefore the unanimous endorsement of those optional provisions for Interval Arithmetic within the proposed IEEE standard must stem from other perceptions.

The proceedings of this symposium illustrate how diverse are the applications of Interval Arithmetic, but always in conjunction with ordinary arithmetic and, most important, always allowing for intermediate estimates and calculations that may fall far from their ultimate objectives. Some calculation is best done carelessly but fast; later it will be tidied up with a little bit of the most meticulous kind of calculation. Interval arithmetic can be used in either phase or both; its potential usefulness is not disputed.

IV. ANTITHEOREMS

What must be disputed is a certain attitude toward computation, as if it were a chain no stronger than its weakest link. In fact, many modern calculations more resemble webs in which some strands are far stronger than others but also weak strands must contribute essential strength to the whole. Rather than philosophize further on this theme, I shall offer certain statements that illustrate the intellectual poverty of the chain-like attitude to computation. The statements are couched as theorems which are widely believed and, alas, wide-

ly taught. The theorems are false; that is why they are called anti-theorems.

Anti-Theorem 1: If all the intermediate results at some intermediate stage of a computation are wrong, in the sense that they are very different from what would have been calculated using infinite precision and range, and if the final result depends strongly upon those intermediate results, then the final result must be wrong too.

Counterexample: The following program calculates the analytic function

$$f(x) = (e^x - 1)/x = \sum_1^\infty x^{n-1}/n! \quad .$$

Real procedure f(x): real value x.
x := e^x ... rounded. x is overwritten to save memory space.
if x ≠ 1 then x := (x - 1)/ln x.
Return f:= x end... correct if no over/underflow occurs.

At that point in the program where (x - 1) and ln x are about to participate in division there need be no record of the original value of x. But if that original value was, say, $x = 1.49_{10} - 9$ on a 10 sig.dec. machine then calculated values

(x - 1) and ln x will be respectively $1.00_{10} - 9$ and $9.9999\ 99995_{10} - 10$ quite different from the "correct" values $1.4900\ 00001_{10} - 9$ and $1.49_{10} - 9$. Still we find f = 1.0000 00001 correctly in either case. Of course the errors have cancelled, but this is not a trivial accomplishment considering that one might naively have written a shorter program

f(x) := if x = 0 then 1 else $(e^x - 1)/x$

which suffers rather than benefits from cancellation. Just such a naive program accounts for egregious errors produced by certain popular financial calculators (not those made by H-P).

Modern machine computation would be uneconomical if anti-theorem 1 were true. Only because it is false can we solve large systems of linear equations, calculate accurate eigenvalues and eigenvectors, and solve accurately certain initial value problems which demand large numbers of tiny steps. Alas, because anti-theorem 1 is false, naive users of interval arithmetic are doomed too often to obtain grotesquely pessimistic error bounds. Therefore there will always be a small but steady demand for error-analysts to practice their dreary profession provided they continue to expose bad algorithms' big errors and, more important, supplant bad algorithms with provably good ones regardless of anti-theorem 1.

Anti-theorem 2: The accuracy of a computation can always be improved in the face of roundoff by carrying more figures; successive re-computations of a continuous function in single-precision, double-precision, quadruple-precision, ... must converge to the value that would be produced by an ideal computation free from roundoff.

Counterexample: Execute the following program on any computer with built-in (rather than programmed) floating point arithmetic, rounded or chopped, binary or ternary or decimal or ...

X := 2.0/3.0 ; Y := |3.0(X - 0.5) - 0.5|/25.

Z := if Y = 0 then 1 else $(e^Y - 1)/Y$.

In infinite precision we should get Z = f(0) = 1 (cf. anti-theorem 1) but every computer I know produces instead Z = 0 regardless of the precision carried during the computation. This example may look artificial, but it imitates life; cf. my "Survey of Error Analysis" (6). Whether such examples are common is impossible to discover because their victims, finding no numerical disagreements, cannot know that something disagreeable has happened.

Evidently no numerical procedure, regardless of the precision with which it is executed, can be entirely trustworthy unless either

- it is executed in interval arithmetic, or
- it passes the conscientious scrutiny of a competent error analyst.

Both options are so often so pessimistic and so costly that most people prefer to take their chances with computations carried out with precisions believed, rightly or wrongly, to exceed by far what is necessary. Their attitude makes sense; they would rather believe the error to be negligible than know how big it isn't. Their attitude will not change until the cost of error analysis, whether performed during their computation or before, declines to something near its perceived value.

Anti-theorem 3: Short computations free from overflow, underflow or cancellation must be correct. Specifically, if a computation involves only positive numbers, none of them beyond the machine's range, and performs at most 100 rounded arithmetic operations drawn from the set {+, x, /, $\sqrt{\ }$} (no subtraction, hence no cancellation), then the result of the computation cannot differ by more than about 100 ulps*) from what would have been calculated using infinite precision.

Counterexample: To replace x by |x| execute the instructions

x := x^2

x := $\sqrt{x}$

x := $\sqrt{x}$ 50 square roots

...

x := $\sqrt{x}$

*) An ulp is a Unit in the Last Place of a computed result.

$x := x^2$
$x := x^2$
49 multiplications.
...
$x := x^2$

But on most computers that round off their square root correctly this program calculates $|x| = 1$ for all numbers x between 0.5 and 2 and usually gets $|x|$ very wrong otherwise. The final error can be as big as 200,000,000,000,000. ulps, or worse.

This counterexample is neither an isolated example nor a loophole overlooked in the anti-theorem's statement. On the contrary, the results of computations (involving millions rather than hundreds of operations) are frequently wrong, when they are very wrong, just because of the kind of insidious error-amplification illustrated here rather than because of cancellation or over/underflow or even mere accumulation of multitudinous errors. Well-known instances are associated with the solution of linear systems via (possibly ill-conditioned) triangular factors, and with (possibly unstable) recurrences used to solve initial value problems or calculate special functions or eigenvectors, etc.

Many people, clever and otherwise competent, cannot accept that anti-theorem 3 is false. Their intuition, conditioned by experience with very short computations of their own devising or with long computations based upon the standard library of stable algorithms, confirms the anti-theorem. Consequently

they tend to believe in a kind of computational justice; if every part of a computation is calculated about as accurately as conscientious and virtuous programming allows, then the whole computation deserves to be correct. Alas, virtue is its own and often sole reward. The unacknowledged failure of anti-theorem 3, in conjunction with the previous two, has the most profound consequences for the design of arithmetic hardware.

V. THE ANTI-THEOREMS' IMPACT

Since computation can come out wrong when done right (AT3) and right when done wrong (AT1) and nobody can tell which (AT2) except an error analyst, why bother to build arithmetic hardware Carefully when Fast is good enough?

The reason for carefully designed arithmetic, and for the sometimes paradoxical indifference to what may appear to be calamitous aberrations within what is none the less claimed to be carefully designed arithmetic, cannot be found within the chain-like model of computation. The reason can be found in the web-model. Computation preserves a web of relationships, some of them strongly (like the commutativity of addition and multiplication despite roundoff) and others weakly (like monotonicity), and other relationships leave only suggestive shadows in our memories. With experience we learn which relationships matter.

Usefully to illustrate the web-model of computation is impractical except by proffering an error analysis that nobody wants to read, though that thought has never inhibited publication. What I offer instead is an enlargement upon one aspect of the KCS proposal, its most controversial.

VI. WHAT IS GRADUAL UNDERFLOW?

It is a scheme which reduces the effect of underflow to something comparable with the uncertainty due to roundoff in a wide range of numerical computations including most of those with matrices, quadrature, ordinary differential equations, zero-finding, convergence acceleration, ... To understand gradual underflow we have to understand a little about formats used for floating point arithmetic.

Normalized floating point numbers look like

$$\pm(D_0.d_1d_2\ldots d_n) \times B^e \qquad \text{e.g. } \pm(3.14159) \times 10^5$$

where the radix B can be 2(binary), 8(octal), 10(decimal) or 16(hexadecimal) on diverse American computers; then the n+1 "significant digits" D_0, d_1, $d_2\ldots d_n$ are each drawn from the set {0,1,2,...,B-1} except that $D_0 \neq 0$; and the exponent e is an integer confined to some interval $\check{e} \leq e \leq \hat{e}$; e.g. hand-held calculators use B = 10 and $\check{e} = -99 \leq e \leq \hat{e} = 99$. The foregoing format excludes zero unless a special case is set aside

for $0 = \pm(0.00\ldots0) \times (B^{\check{e}}$ or $B^0)$; whether zero is called normalized or not (because its $D_0 = 0$) does not matter. Unnormalized floating point numbers, with $D_0 = 0$ but $e > \check{e}$, are usually outlawed as redundant in so far as their values can be represented equally well by normalized numbers (but the B5500 is one computer which treats unnormalized and normalized numbers indistinguishably). Denormalized numbers are characterized by $D_0 = 0$ and $e = \check{e}$; for instance on a 6 sig. dec. calculator

$$(0.03142) \times 10^{-99}$$

is the best convenient approximation for $\pi/10^{101}$. Note that denormalized numbers look unnormalized at first until you notice that the exponent $e = \check{e}$ is minimal and realize that no denormalized number can be represented by a normalized number in the same format whereas an unnormalized number can be supplanted exactly by another, normalized or denormalized, with a lesser exponent e.

But most computers have outlawed denormalized numbers too. Those computers "flush" any attempt to compute a floating point number whose normalized form would otherwise underflow (have $e < \check{e}$); for instance, most hand-held calculators will neither allow $(\pi/10^{50})/10^{51}$ to be represented as 3.14159×10^{-101} nor denormalize it to 0.03142×10^{-99} but will instead display zero or 1×10^{-99} or "Error". Only computers that underflow gradually possess denormalized numbers. Among such computers are the Electrologica X-8, Intel 8086-8088 with 8087, Motorola 6809 with 6839 (and IBM 7094, IBM 360-370, Burroughs B5500, DEC 20/PDP 10 with appropriate trap-

handling software not currently distributed by their manufacturers).

How does gradual underflow help? Its most obvious effect is to preserve the following relation: if $0 < x < y$ then $x/y < 1$ and $y-x > 0$. The last inequality can be falsified by computers that flush underflows to zero, but not by those that underflow gradually. The effect upon program logic, where a test of one relation is presumed to imply others, is immediately clear, especially when we expect "$y-x = 0$" to imply "$f(y) - f(x) = 0$" unless f is a pathological function (involving, say, division by zero). But the effect goes much deeper. Consider two nearby representable numbers x and y and their difference $z = x - y$, and suppose x and y are so close to each other that $|z| \leq |x|$ and $|z| \leq |y|$. Then z must be representable exactly despite roundoff and, if underflow is gentle, despite underflow. But just as poorly designed arithmetic units can contaminate small differences unnecessarily with roundoff (as do CDC 6000 class computers and most TI calculators), so will flushing underflows contaminate them. This contamination is a noise that may defeat the feedback loop intended to stabilize a computation. For instance, to solve the equation $f(s) = t$ for s given f and t, we are obliged to calculate the discrepancy $t-f(s)$ and feed it back to alter s, as in Newton's method:

$$\text{new } s = s + (t - f(s))/f'(s).$$

The noise in $t-f(s)$, whether caused by roundoff or underflow, is the principal limitation upon the accuracy to which a solution s can be calculated. Gradual underflow makes much less noise here than does flushing.

Gradual underflow reduces every instance of underflow to an amount absolutely no bigger than a rounding error in the last significant digit of the smallest normalized number. This means very often that a program which is proved correct in the face of roundoff can easily be proved correct despite underflow too provided underflow is gradual, whereas when underflow is flushed that program may have to be augmented by tests against format-dependent thresholds to provide a defense against flushing. This is why gradual underflow is valuable for matrix multiplication and inversion, operations so common that their enhancement by gradual underflow is enough to justify providing that feature. Moreover, subroutines programmed conscientiously can often be designed more easily, thanks to gradual underflow, to cope with their own underflows automatically; consequently the naive users of such subroutines can expect to see underflow messages significantly less often on machines that underflow gradually than on machines that flush.

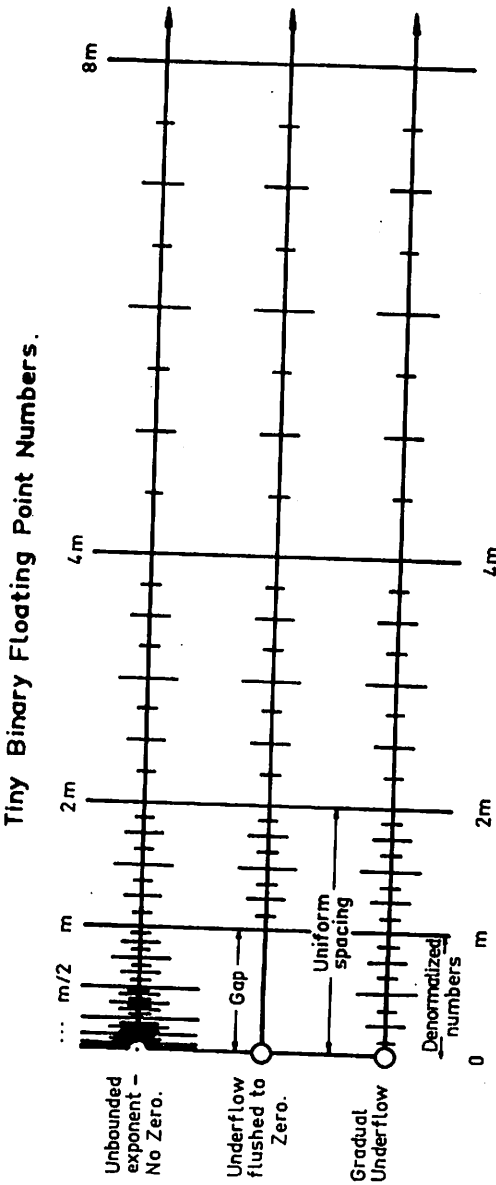
Some kinds of underflow cannot be cured by gradualness, nor by flushing. For intance, long chains of multiplications and divisions unrelieved by alternate additions generally require scaling to defend against over/underflow unless they can be calculated with the aid of a (possibly temporary) exponent field extension (see SHARE SSD 159 item C 4537, Dec. 1966). So gradual underflow is no panacea. All that can

be claimed is that it improves a large and recognizable class of programs without making others worse, without complicating our concept of representable numbers, and without much of a penalty in hardware or speed except possibly on pipelined parallel array-oriented machines where a different approach (the KCS "Extended" format described by J. Coonen (2)) is more appropriate for reasons having little to do with underflow.

Before you rush out to implement gradual underflow on your computer be sure that it has or can be given an Underflow flag. The best way to test whether an expression has underflowed, when underflow is gradual, is to test the flag rather than to test whether the expression is zero or denormalized; see Coonen's article.

ADDENDUM

The proposed standard attempts to protect programmers accustomed to machines that flush to zero from being ambushed by gradual underflow. To this end, attempted divisions by denormalized numbers, and multiplications and divisions that magnify them excessively, are inhibited (those operations are regarded as invalid and yield NaNs) unless the program has previously instituted the "Normalizing Mode," thereby placing in evidence the programmer's awareness of the issue. Ultimately, most programs, like the examples that follow, will invoke that mode.



Each vertical tic —|— stands for a 4-sig. bit binary floating point number. The underflow threshold m is a power of 1/2 depending upon the allowed range of exponents; every floating point number ≥ bigger than m, but none smaller, is representable as a normalized floating point number. On some machines (IBM 7094, DEC PDP-10, DEC PDP-11, ...) m is a normalized number too, on others (H-P 3000) not. Flushing underflows to zero introduces a gap between m and 0 much wider than between m and the next larger number. Gradual Underflow fills that gap with Denormalized numbers as densely packed between m and 0 as are normalized numbers between m and 2m. Doing so relegates underflow in most computations to a status comparable with roundoff among the normalized numbers.

VII. EXAMPLES

Example 1. Complex Absolute Value Cabs (X+iY) = $\sqrt{(X^2+Y^2)}$

The simplest program exploits the Extended format to dispel the nuisance of over/underflow and to suppress roundoff below 1 ulp (Unit in the Last Place) of the result:

```
Real procedure   Cabs (X, Y):
                 Real values X, Y ; Extended S, anonymous
                 variables;
                 Save & set modes Normalizing, Affine ... to
                 distinguish ±∞.
                 S := √(X^2 + Y^2) ... evaluated in Extended format.
                 Restore Modes; Return Cabs := S end
```

The foregoing program cannot mislead nor be misled by pathologies like underflow or invalid operation; for instance Cabs (∞+i∞) = ∞ with no warning flag set. The only possible new pathology is overflow of the calculated Cabs when that overflow is deserved; intermediate over/underflow of $X^2 + Y^2$ is precluded by its extended range.

But what if the Extended format is unimplemented? Then we must cope somehow with data |X| and |Y| which lie outside the interval bounded by the square roots of the overflow and underflow thresholds. For definiteness suppose

$0 \neq x \geq y \geq 0$, so that we may consider the formula $\sqrt{(x^2 + y^2)} = x\sqrt{(1 + (y/x)^2)}$ in which $y/x \leq 1$. Here underflow of y/x or $(y/x)^2$ renders it negligible compared with 1, so over/underflow poses no unwarranted hazard; but roundoff is a nuisance because this formula, evaluated entirely in single precision, can suffer a final error almost as big as (but no bigger than) about (2+3B)/4 ulps where B is the radix (B = 2 for binary, 10 for decimal, etc.). For instance, in decimal the last formula produces Cabs (96,28) = 99.999...994 instead of 100. A somewhat more accurate formula is used in the next program which exhibits also the code needed to cope properly with every pathology (0, ∞, NaN, etc.):

```
Real procedure Cabs (X, Y):
               Real values X, Y;  real r.
               Save & clear Flags Overflow, Underflow, Invalid,
               Divide by zero.
               Save & set Modes Normalizing, Affine.
               X := |X| ; Y := |Y| ;  if Y > X then Swap (X,Y)...
                                      Y ≤ X.
               Clear Invalid Flag ... in case comparison in-
               volves a NaN.
               r := X/Y ...  r ≥ 1  or else invalid 0/0 or ∞/∞.
               r := Y/(r + √(1+r^2)) ... may underflow gradually.
               If Invalid Flagged then r := 0.
               Restore Flags; Restore Modes.
               Return Cabs := X + r   end.
```

On a binary machine this program's error cannot exceed $\frac{3}{2}$ ulp, with a comparable bound for other radices, but the error can exceed 1 ulp; e.g. using 10 sig.decimals yields (cf. R→P on hp-65) Cabs (4684660,4684659) = 6625109.001 instead of 6625109.

Most people accept the foregoing program's slight inaccuracy since a smaller error costs too much (see below). But note here the beneficial role played by gradual underflow in the last value of r; if underflow were flushed to zero instead, the calculated Cabs could be quite wrong whenever X and Y were not both much bigger than the underflow threshold. Lacking gradual underflow, the conscientious programmer is forced to complicate the foregoing program with radix-dependent scaling operations.

Finally, for perfectionists who can tolerate no error in Cabs so large as 1 ulp but who have been denied the convenience of an Extended format, here is a better program:

Real procedure Cabs:

Real values X, Y; real r, s.
Constants $\alpha = \sqrt{2}$, $\gamma = 1+\sqrt{2}$ rounded, $\beta = (1+\sqrt{2} - \gamma)$ to several sig.dec. ... e.g. to 10 sig.dec.
$\alpha = 1.4142\ 13562$, $\gamma = 2.414213562$,
$\beta = 3.7309505_{10}-10$.
Save & clear Flags Overflow, Underflow, Invalid, Divide by zero.
Save & set Modes Normalizing, Affine.
X := |X|; Y := |Y|; if Y > X then swap (X,Y) ... $Y \leq X$.

r := X - Y.
If r ≤ Y then begin r := r/Y ; s := r(r+2)
r := ((s/(α+√(2+s))+r)+β)+γ
end
else begin Clear Invalid Flag; r := X/Y;
r := r + √(1+r²) end.
r := Y/r; if Invalid Flagged then r := 0.
Restore Flags; Restore Modes.
Return Cabs := X + r end.

Surely this cannot be preferable to the first Cabs program above.

Example 2. Complex Divide x + iy = (a+ib)/(c+id).

The simplest program, and the best, exploits the Extended format again:

Procedure Complex divide (a, b, c, d; x,y):
Real values a, b, c, d; Real output x,y.
Extended S, T, U, anonymous variables.
Save and set modes Normalizing, Affine.
S := $c^2 + d^2$; T := ac + bd ; U := bc - ad.
Restore modes; x := T/S ; y := U/S ; Return end.

But without extended variables to hold intermediate results the foregoing program is fatally vulnerable to over/underflow in S, T and U from which may follow completely unreasonable values for x and y. A program appropriate for systems with no extended variables can be adapted from R. L. Smith's algorithm described in vol. 2 of D. E. Knuth's "The Art of Computer Programming". His algorithm assumes that $|d| \leq |c| \neq 0$; otherwise either compute $x+iy = (b-ia)/(d-ic)$ instead or deal appropriatly with the case $c = d = 0$. Next let $r = d/c$, so $|r| \leq 1$, and $s = c + dr$; then $x = (a+br)/s$ and $y = (b-ar)/s$. But now observe how gradual underflow comes into play here; if all of a, b, c, d, were only moderately larger than the underflow threshold, and if one or more of dr, br and ar underflowed, then flushing underflows to zero must produce plausible but occasionally utterly wrong values for x and y. The reader should experiment with these formulas on suitably chosen data, including also ∞ and NaN and denormalized numbers, and then try to write a satisfactory program. Next try to get along without gentle underflow, if you can.

ANNOTATED BIBLIOGRAPHY

1. ACM SIGNUM Newsletter Special Issue, Oct. 1979, is devoted to the proposed floating point standard and includes
   - Draft 5.11 of the standard, by Coonen, Kahan, Palmer, Pittman and Stevenson
   - Expository article by Kahan and Palmer about KCS
   - Critique by Fraley and Walther, authors of the FW proposal.
   - Counter proposal by Payne and Strecker, the PS proposal.
   - other shorter comments.
2. Coonen, J., "An Implementation Guide to a Proposed Standard for Floating Point Arithmetic" in the IEEE/CS journal "Computer", Jan. 1980, pp. 68 - 79. This lays out the KCS proposal.
3. "Electronic Design", Feb. 1980; contains an article by Burdick et al. about National Semiconductor's 16000 series.
4. "Electronics" for May 4, 1980 (published by Mc Graw-Hill). Contains an article by Palmer et al. describing the i8087, and another article that peeks at Motorola's plans for the 6809 and 68000.
5. IEEE/CS working group on Floating Point Standardization. The minutes of the meetings are available at irregular intervals from the distributing secretary, currently
   Dr. David Hough.
   APPLE Computer Co.
   10260 Brandley Drive
   Cupertino, Calif. 95104, U.S.A.
6. Kahan, W., "A Survey of Error Analysis" in "Information Processing 71" (North Holland), the proceedings of IFIP'71 in Ljubljana. An exposé of arithmetic design, plus refe-

rences to some of the author's earlier unpublished works and SHARE report.

7. Laveuve, S., "Definition einer Kahan-Arithmetik und ihre Implementierung in Triplex-Algol 60". Interner Bericht 75/6, Institut für Praktische Mathematik, Universität Karlsruhe, Feb. 1975

8. Palmer, J., "The INTEL Standard for Floating Point Arithmetic" in Proc. COMPSAC (Chicago, 1977). Announces a precursor to ~~IC~~. KCS.

9. Proceedings of "ELECTRO 80", Boston May 13-15, sessions 14 and 18. Contains more about i8087, KCS pro and con, VAX by Payne and Bhandarkar, AMD 9512, etc. .

Added in proof: Aug. 1980

A revised draft 7.0 of the proposed IEEE floating point standard is about to be promulgated. It includes the working group's recent decision to make directed roundings obligatory instead of an implementor's option, although nothing is said about the way higher-level languages are to convey this capability to programmers.

Acknowledgement: Much of the author's work reported herein has been supported by a research contract N 00014-76-C-0013 with the U.S. Office of Naval Research, and by contract DE-AT03-76SF00034 with the U.S. Department of Energy.