Lecture Notes on the Status of

IEEE Standard 754 for Binary Floating-Point Arithmetic

Prof. W. Kahan
Elect. Eng. & Computer Science
University of California
Berkeley CA 94720-1776

Introduction:

Twenty years ago anarchy threatened floating-point arithmetic. Over a dozen commercially significant arithmetics boasted diverse wordsizes, precisions, rounding procedures and over/underflow behaviors, and more were in the works. "Portable" software intended to reconcile that numerical diversity had become unbearably costly to develop.

Eleven years ago, when IEEE 754 became official, major microprocessor manufacturers had already adopted it despite the challenge it posed to implementors. With unprecedented altruism, hardware designers rose to its challenge in the belief that they would ease and encourage a vast burgeoning of numerical software. They did succeed to a considerable extent. Anyway, rounding anomalies that preoccupied all of us in the 1970s afflict only CRAYs now.

Now atrophy threatens features of IEEE 754 caught in a vicious circle:

Those features lack support in programming languages and compilers,
so those features are mishandled and/or practically unusable,
so those features are little known and less in demand, and so
those features lack support in programming languages and compilers.

To help break that circle, those features are discussed in these notes under the following headings:

Insofar as this is a status report, it is subject to change and supersedes versions with earlier dates. This version supersedes one distributed at a panel discussion of "Floating-Point Past, Present and Future" in a series of San Francisco Bay Area Computer History Perspectives sponsored by Sun Microsystems Inc. in May 1995. A PostScript version is accessible electronically as http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps .

Representable Numbers:

IEEE 754 specifies three types or *Formats* of floating-point numbers:

Single (Fortran's REAL*4, C's `float`),	(Obligatory),
Double (Fortran's REAL*8, C's `double`),	(Ubiquitous), and
Double-Extended (Fortran REAL*10+, C's `long double`),	(Optional).

(A fourth Quadruple-Precision format is not specified by IEEE 754 but has become a *de facto* standard among several computer makers none of whom support it fully in hardware yet, so it runs slowly at best.)

Each format has representations for NaNs (Not-a-Number), $\pm\infty$ (Infinity), and its own set of finite real numbers all of the simple form

$$2^{k+1-N}\, n$$

with two integers n (signed *Significand*) and k (unbiased signed *Exponent*) that run throughout two intervals determined from the format thus:

K+1 Exponent bits: $1 - 2^K < k < 2^K$. N Significant bits: $-2^N < n < 2^N$.

Table of Formats' Parameters:

Format	Bytes	K+1	N
Single	4	8	24
Double	8	11	53
Double-Extended	≥ 10	≥ 15	≥ 64
(Quadruple	16	15	113)

This concise representation $2^{k+1-N}\, n$, unique to IEEE 754, is deceptively simple. At first sight it appears potentially ambiguous because, if n is even, dividing n by 2 (a right-shift) and then adding 1 to k makes no difference. Whenever such an ambiguity could arise it is resolved by minimizing the exponent k and thereby maximizing the magnitude of significand n ; this is " Normalization " which, if it succeeds, permits a Normal nonzero number to be expressed in the form $2^{k+1-N}\, n = \pm 2^k (1 + f)$ with a nonnegative *fraction* $f < 1$.

Besides these Normal numbers, IEEE 754 has Subnormal (Denormalized) numbers lacking or suppressed in earlier computer arithmetics; Subnormals, which permit Underflow to be Gradual, are nonzero numbers with an unnormalized significand n and the same minimal exponent k as is used for 0 :

Subnormal $2^{k+1-N}\, n = \pm 2^k (0 + f)$ has $k = 2 - 2^K$ and $0 < |n| < 2^{N-1}$, so $0 < f < 1$.

Thus, where earlier arithmetics had conspicuous gaps between 0 and the tiniest Normal numbers $\pm 2^{2-2^K}$, IEEE 754 fills the gaps with Subnormals spaced the same distance apart as the smallest Normal numbers:

```
   Subnormals    [--- Normalized Numbers ----- - - -  -  -  -  -   -  -  ->
                 |                |                                |
0-!-!-+-!-+-+-+-+-!-+-+-+-+-+-+-+-+-!---+---+---+---+---+---+---+---!------ - -
  | |   |         |                |                                |
Powers of 2 :  2^(2-2^K)         2^(3-2^K)                        2^(4-2^K)
```

–+– Consecutive Positive Floating-Point Numbers –+–

IEEE 754 encodes floating-point numbers in memory (not in registers) in ways first proposed by I.B. Goldberg in *Comm. ACM* (1967) 105-6 ; it packs three fields with integers derived from the sign, exponent and significand of a number as follows. The leading bit is the sign bit, 0 for + and 1 for - . The next K+1 bits hold a biased exponent. The last N or N-1 bits hold the significand's magnitude. To simplify the following table, the significand n is dissociated from its sign bit so that n may be treated as nonnegative.

Encodings of $\pm 2^{k+1-N}$ n into Binary Fields :

Number Type	**Sign** Bit	K+1 bit **Exponent**	Nth bit	N-1 bits of **Significand**
NaNs:	?	binary 111...111	1	binary 1xxx...xxx
SNaNs:	?	binary 111...111	1	nonzero binary 0xxx...xxx
Infinities:	±	binary 111...111	1	0
Normals:	±	$k-1+2^K$	1	nonnegative $n-2^{N-1} < 2^{N-1}$
Subnormals:	±	0	0	positive $n < 2^{N-1}$
Zeros:	±	0	0	0

Note that +0 and -0 are distinguishable and follow obvious rules specified by IEEE 754 even though floating-point arithmetical comparison says they are equal; there are good reasons to do this, some of them discussed in my 1987 paper " Branch Cuts ... ." The two zeros are distinguishable arithmetically only by either division-by-zero (producing appropriately signed infinities) or else by the CopySign function recommended by IEEE 754 / 854. Infinities, SNaNs, NaNs and Subnormal numbers necessitate four more special cases.

IEEE Single and Double have no Nth bit in their significant digit fields; it is " implicit." 680x0 / ix87 Extendeds have an explicit Nth bit for historical reasons; it allowed the Intel 8087 to suppress the normalization of subnormals advantageously for certain scalar products in matrix computations, but this and other features of the 8087 were later deemed too arcane to include in IEEE 754, and have atrophied.

Non-Extended encodings are all " Lexicographically Ordered," which means that if two floating-point numbers in the same format are ordered (say $x < y$), then they are ordered the same way when their bits are reinterpreted as Sign-Magnitude integers. Consequently, processors need no floating-point hardware to search, sort and window floating-point arrays quickly. (However, some processors reverse byte-order!) Lexicographic order may also ease the implementation of a surprisingly useful function NextAfter(x, y) which delivers the neighbor of x in its floating-point format on the side towards y .

Algebraic operations covered by IEEE 754, namely + , - , · , / , √ and Binary <-> Decimal Conversion with rare exceptions, must be *Correctly Rounded* to the precision of the operation's destination unless the programmer has specified a rounding other than the default. If it does not Overflow, a correctly rounded operation's error cannot exceed half the gap between adjacent floating-point numbers astride the operation's ideal (unrounded) result. Half-way cases are rounded *to Nearest Even*, which means that the neighbor with last digit 0 is chosen. Besides its lack of statistical bias, this choice has a subtle advantage; it prevents prolonged drift during slowly convergent iterations containing steps like these:

While (...) do { y := x+z ; ... ; x := y-z } .

A consequence of correct rounding (and Gradual Underflow) is that the calculation of an expression X•Y for any algebraic operation • produces, if finite, a result $(X\bullet Y)\cdot(1+ß)+\mu$ where $|\mu|$ cannot exceed half the smallest gap between numbers in the destination's format, and $|ß| < 2^{-N}$, and $ß\cdot\mu = 0$. ($\mu \neq 0$ only when Underflow occurs.) This characterization constitutes a weak model of roundoff used widely to predict error bounds for software. The model characterizes roundoff *weakly* because, for instance, it cannot confirm that, in the absence of Over/Underflow or division by zero, $-1 \le x/\sqrt{(x^2+y^2)} \le 1$ despite five rounding errors, though this is true and easy to prove for IEEE 754, harder to prove for most other arithmetics, and can fail on a CRAY Y-MP.

The following table exhibits the span of each floating-point format, and its precision both as an upper bound 2^{-N} upon relative error ß and in " Significant Decimals."

Span and Precision of IEEE 754 Floating-Point Formats :

Format	Min. Subnormal	Min. Normal	Max. Finite	2^{-N}	Sig. Dec.
Single:	1.4 E-45	1.2 E-38	3.4 E38	5.96 E-8	6 - 9
Double:	4.9 E-324	2.2 E-308	1.8 E308	1.11 E-16	15 - 17
Extended:	≤ 3.6 E-4951	≤ 3.4 E-4932	≥ 1.2 E4932	≤ 5.42 E-20	≥ 18 - 21
(Quadruple:	6.5 E-4966	3.4 E-4932	1.2 E4932	9.63 E-35	33 - 36)

Entries in this table come from the following formulas:

Min. Positive Subnormal:		$2^{3 - 2^K - N}$
Min. Positive Normal:		$2^{2 - 2^K}$
Max. Finite:		$(1 - 1/2^N)\ \ 2^{2^K}$
Sig. Dec.,	at least:	floor((N-1) $\text{Log}_{10}(2)$) sig. dec.
	at most:	ceil(1 + N $\text{Log}_{10}(2)$) sig. dec.

The precision is bracketed within a range in order to characterize how accurately conversion between binary and decimal has to be implemented to conform to IEEE 754. For instance, " 6 - 9 " Sig. Dec. for Single means that, in the absence of OVER/UNDERFLOW, ...

> If a decimal string with at most 6 sig. dec. is converted to Single and then converted back to the same number of sig. dec., then the final string should match the original. Also, ...
>
> If a Single Precision floating-point number is converted to a decimal string with at least 9 sig. dec. and then converted back to Single, then the final number must match the original.

Most microprocessors that support floating-point on-chip, and all that serve in prestigious workstations, support just the two REAL*4 and REAL*8 floating-point formats. In some cases the registers are all 8 bytes wide, and REAL*4 operands are converted on the fly to their REAL*8 equivalents when they are loaded into a register; in such cases, immediately rounding to REAL*4 every REAL*8 result of an operation upon such converted operands produces the same result as if the operation had been performed in the REAL*4 format all the way.

But Motorola 680x0-based Macintoshes and Intel ix86-based PCs with ix87-based (not Weitek's 1167 or 3167) floating-point behave quite differently; they perform all arithmetic operations in the Extended format, regardless of the operands' widths in memory, and round to whatever precision is called for by the setting of a control word.

Only the Extended format appears in a 680x0's eight floating-point flat registers or an ix87's eight floating-point stack-registers, so all numbers loaded from memory in any other format, floating-point or integer or BCD, are converted on the fly into Extended with no change in value. All arithmetic operations enjoy the Extended range and precision. Values stored from a register into a narrower memory format get rounded on the fly, and may also incur OVER/UNDERFLOW. (Since the register's value remains unchanged, unless popped off the ix87's stack, misconstrued ambiguities in manuals or ill-considered " optimizations " cause some compilers sometimes wrongly to reuse that register's value in place of what was stored from it; this subtle bug will be re-examined later under " Precisions of Rounding " below.)

Since the Extended format is optional in implementations of IEEE 754, most chips do not offer it; it is available only on Intel's x86/x87, Pentium, P6 and their clones by AMD and Cyrix, on Intel's 80960 KB, on Motorola's 68040/60 or earlier 680x0 with 68881/2 coprocessor, and on Motorola's 88110, all with 64 sig.

bits and 15 bits of exponent, but in words that may be 80 or 96 or 128 bits wide when stored in memory. This format is intended mainly to help programmers enhance the integrity of their Single and Double software, and to attenuate degradation by roundoff in Double matrix computations of larger dimensions, and can easily be used in such a way that substituting Quadruple for Extended need never invalidate its use. However, language support for Extended is hard to find.

Multiply-Accumulate, a Mixed Blessing:

The IBM Power PC and Apple Power Macintosh, both derived from the IBM RS/6000 architecture, purport to conform to IEEE 754 but too often use a " Fused " Multiply-Add instruction in a non-conforming way. The idea behind a Multiply-Add (or " MAC " for " Multiply-Accumulate ") instruction is that an expression like $\pm a*b \pm c$ be evaluated in one instruction so implemented that scalar products like

$$a_1*b_1 + a_2*b_2 + a_3*b_3 + ... + a_L*b_L$$

can be evaluated in about L+3 machine cycles. Many machines have a MAC. Beyond that, a *Fused* MAC evaluates $\pm a*b \pm c$ with just one rounding error at the end. This is done not so much to roughly halve the rounding errors in a scalar product as to facilitate fast and correctly rounded division without much hardware dedicated to it.

To compute $q = x/y$ correctly rounded, it suffices to have hardware approximate the reciprocal $1/y$ to several sig. bits by a value t looked up in a table, and then improve t by iteration thus:

$$t := t + (1 - t*y)*t .$$

Each such iteration doubles the number of correct bits in t at the cost of two MACs until t is accurate enough to produce $q := t*x$. To round q correctly, its remainder $r := x - q*y$ must be obtained exactly; this is what the " Fused " in the Fused MAC is for. It also speeds up correctly rounded square root, decimal <-> binary conversion, and some transcendental functions. These and other uses make a Fused MAC worth putting into a computer's instruction set. (If only division and square root were at stake we might do better merely to widen the multiplier hardware slightly in a way accessible solely to microcode, as TI does in its SPARC chips.)

A Fused MAC also speeds up a grubby "Doubled-Double" approximation to Quadruple-Precision arithmetic by unevaluated sums of pairs of Doubles. Its advantage comes about from a Fused MAC's ability to evaluate any product a*b exactly; first let $p := a*b$ rounded off; then compute $c := a*b - p$ exactly in another Fused MAC, so that $a*b = p + c$ exactly without roundoff. Fast but grubby Double-Double undermines the incentive to provide Quadruple-Precision correctly rounded in IEEE 754's style.

Fused MACs generate anomalies when used to evaluate $a*b \pm c*d$ in two instructions instead of three. Which of a*b and c*d is evaluated and therefore rounded first? Either way, important expectations can be thwarted. For example, multiplying a complex number by its complex conjugate should produce a real number, but it might not with a Fused MAC. If $\sqrt{(q^2 - p*r)}$ is real in the absence of roundoff, then the same is expected for

SQRT(q*q - p*r)

despite roundoff, but perhaps not with a Fused MAC. Therefore Fused MACs cannot be used indiscriminately; there are a few programs that contain a few assignment statements from which Fused MACs must be banned.

By design, a Fused MAC always runs faster than separate multiplication and add, so compiler writers with one eye on benchmarks based solely upon speed leave programmers no way to inhibit Fused MACs selectively within expressions, nor to ban them from a selected assignment statement.

Ideally, some locution like redundant parentheses should be understood to control the use of Fused MACs on machines that have them. For instance, in Fortran, ...

(A*B) + C*D and C*D + (A*B) should always round A*B first;
(A*B) + (C*D) should inhibit the use of a Fused MAC here.

Something else is needed for C , whose Macro Preprocessor often insinuates hordes of redundant parentheses. Whatever expedient is chosen must have no effect upon compilations to machines that lack a Fused MAC; a separate compiler directive at the beginning of a program should say whether the program is intended solely for machines with, or solely for machines without a Fused MAC.

Exceptions in General.

Designers of operating systems tend to incorporate all trap-handling into their handiwork, thereby burdening floating-point exception-handling with unnecessary and intolerable overheads. Better designs should incorporate all floating-point trap-handling into a run-time math. library, along with logarithms and cosines, which the operating system merely loads. To this end, the operating system has only to provide default handlers (in case the loaded library neglects trap-handling) and secure trap re-vectoring functions for libraries that take up that duty and later, at the end of a task, relinquish it.

To Disable an exception's trap is to let the numeric (co)processor respond to every instance of that exception by raising its Flag and delivering the result specified as its " Default " in IEEE 754. For example, the default result for 3.0/0.0 is ∞ with the same sign as that 0.0 . The raised flag stays raised until later set down by the program, perhaps after it has been sensed. IEEE 754 allows for the possibility that raised flags be non-null pointers, but most microprocessors keep one or two bits per flag in a Status register whose sensing and clearing fall outside the scope of these notes. The same goes for bits in a Control register that Enable/Disable traps; see manuals for your chip and for the programming environment (e.g. compiler) that concerns you.

```
+-------------------------------------------------------------------+
|  CAUTION:  Do not change  ( enable or disable )  exception traps  |
|       in a way contrary to what is expected by your programming   |
|       environment or application program,  lest unpredictable     |
|       consequences ensue for lack of a handler or its action.     |
+-------------------------------------------------------------------+
```

Disputes over exceptions are unavoidable. In fact, one definition for " Exception " is ...

" Event for which any policy chosen in advance will subsequently
give some reasonable person cause to take exception."

A common mistake is to treat exceptions as errors and punish the presumed perpetrators; usually punishment falls not upon deserving perpetrators but upon whatever interested parties happen to be in attendance later when exceptions arise from what was perpetrated, error or not.

```
+-------------------------------------------------------------------+
|    Exceptions that reveal errors are merely messengers.  What     |
|    turns an exception into an error is bad exception-handling.    |
+-------------------------------------------------------------------+
```

Attempts to cope decently with all exceptions inevitably run into unresolved dilemmas sooner or later unless the computing environment provides what I call " Retrospective Diagnostics.". These exist in a rudimentary form in Sun Microsystems' operating system on SPARCs. The idea is to log (in the sense of a ship's log) every suspicious event that is noticed during a computation. These events are logged not by time of occurrence (which could fill a disk very soon) but by site in a program. A hashing scheme ensures that events repeated at the same site will perhaps update but certainly not add entries to the log. Neither need an exception that occurs while its flag is still raised by a previous exception of the same kind add a new entry to the log. After a program finishes (if it ever does), its user may be notified discreetly if a dangerous flag like INVALID is still raised; then that flag can serve as a pointer to its entry in the log. The log cannot grow intolerably long, so unusual entries stand out and point to whatever software module put them there. The user of that software can then identify the module in question and ask its supplier whether an entry is something to worry about.

Exception: **INVALID** operation.

Signaled by the raising of the INVALID flag whenever an operation's operands lie outside its domain, this exception's default, delivered only because any other real or infinite value would most likely cause worse confusion, is NaN , which means " Not a Number." IEEE 754 specifies that seven invalid arithmetic operations shall deliver a NaN unless they are trapped:

real √(Negative) , 0*∞ , 0.0/0.0 , ∞/∞,
REMAINDER(Anything, 0.0) , REMAINDER(∞, Anything) ,
∞ - ∞ when signs agree (but ∞ + ∞ = ∞ when signs agree).

Conversion from floating-point to other formats can be INVALID too, if their limits are violated, even if no NaN can be delivered.

NaN also means " Not any Number " ; NaN does not represent the set of all real numbers, which is an interval for which the appropriate representation is provided by a scheme called " Interval Arithmetic."

NaN must not be confused with " Undefined." On the contrary, IEEE 754 defines NaN perfectly well even though most language standards ignore and many compilers deviate from that definition. The deviations usually afflict relational expressions, discussed below. Arithmetic operations upon NaNs other than SNaNs (see below) never signal INVALID, and always produce NaN unless replacing every NaN operand by any finite or infinite real values would produce the same finite or infinite floating-point result independent of the replacements.

For example, 0*NaN must be NaN because 0*∞ is an INVALID operation (NaN). On the other hand, for hypot(x, y) := √(x*x + y*y) we find that hypot(∞, y) = +∞ for all real y , finite or not, and deduce that hypot(∞, NaN) = +∞ too; naive implementations of hypot may do differently.

NaNs were not invented out of whole cloth. Konrad Zuse tried similar ideas in the late 1930s; Seymour Cray built " Indefinites " into the CDC 6600 in 1963; then DEC put " Reserved Operands " into their PDP-11 and VAX. But nobody used them because they trap when touched. NaNs do not trap (unless they are " Signaling " SNaNs, which exist mainly for political reasons and are rarely used); NaNs propagate through most computations. Consequently they do get used.

Perhaps NaNs are widely misunderstood because they are not needed for mathematical analysis, whose sequencing is entirely logical; they are needed only for computation, with temporal sequencing that can be hard to revise, harder to reverse. NaNs must conform to mathematically consistent rules that were deduced, not invented arbitrarily, in 1977 during the design of the Intel 8087 that preceded IEEE 754. What had been missing from computation but is now supplied by NaNs is an opportunity (not obligation) for software (especially when searching) to follow an unexceptional path (no need for exotic control structures) to a point where an exceptional event can be appraised after the event, when additional evidence may have accrued. Deferred judgments are *usually* better judgments but *not always*, alas.

Whenever a NaN is created from non-NaN operands, IEEE 754 demands that the INVALID OPERATION flag be raised, but does not say whether a flag is a word in memory or a bit in a hardware Status Word. That flag stays raised until the program lowers it. (The Motorola 680x0 also raises or lowers a transient flag that pertains solely to the last floating-point operation executed.) The " Sticky " flag mandated by IEEE 754 allows programmers to test it later at a convenient place to detect previous INVALID operations and compensate for them, rather than be forced to prevent them. However, ...

```
+-----------------------------------------------------------------+
|  Microsoft's  C  and  C++  compilers defeat that purpose of the  |
|  INVALID  flag by using it exclusively to detect floating-point  |
|  stack overflows,  so programmers cannot use it  ( via library   |
|  functions  _clear87  and  _status87 )  for their own purposes.  |
+-----------------------------------------------------------------+
```

This flagrant violation of IEEE 754 appears not to weigh on Microsoft's corporate conscience.
So far as I know, Borland's C , C++ and Pascal compilers do not abuse the INVALID flag that way.

. . . .

While on the subject of miscreant compilers, we should remark their increasingly common tendency to reorder operations that can be executed concurrently by pipelined computers. *C* programmers may declare a variable `volatile` to inhibit certain reorderings. A programmer's intention is thwarted when an alleged " optimization " moves a floating-point instruction past a procedure-call intended to deal with a flag in the floating-point status word or to write into the control word to alter trapping or rounding. Bad moves like these have been made even by compilers that come supplied with such procedures in their libraries. (See _control87 , _clear87 and _status87 in compilers for Intel processors.) Operations' movements would be easier to debug if they were highlighted by the compiler in its annotated re-listing of the source-code. Meanwhile, so long as compilers mishandle attempts to cope with floating-point exceptions, flags and modes in the ways intended by IEEE Standard 754, frustrated programmers will abandon such attempts and compiler writers will infer wrongly that unexercised capabilities are unexercised for lack of demand. . . . .

IEEE 754's specification for NaN endows it with a field of bits into which software can record, say, how and/or where the NaN came into existence. That information would be extremely helpful for subsequent " Retrospective Diagnosis " of malfunctioning computations, but no software exists now to employ it. Customarily that field has been copied from an operand NaN to the result NaN of every arithmetic operation, or filled with binary 1000...000 when a new NaN was created by an untrapped INVALID operation. For lack of software to exploit it, that custom has been atrophying.

680x0 and ix87 treat a NaN with any nonzero binary 0xxx...xxx in that field as an SNaN (Signaling NaN) to fulfill a requirement of IEEE 754. An SNaN may be moved (copied) without incident, but any other arithmetic operation upon an SNaN is an INVALID operation (and so is loading one onto the ix87's stack) that must trap or else produce a new nonsignaling NaN. (Another way to turn an SNaN into a NaN is to turn 0xxx...xxx into 1xxx...xxx with a logical OR.) Intended for, among other things, data missing from statistical collections, and for uninitialized variables, SNaNs seem preferable for such purposes to zeros or haphazard traces left in memory by a previous program. However, no more will be said about SNaNs here.

Were there no way to get rid of NaNs, they would be as useless as Indefinites on CRAYs; as soon as one were encountered, computation would be best stopped rather than continued for an indefinite time to an Indefinite conclusion. That is why some operations upon NaNs must deliver non-NaN results. Which operations?

Disagreements about some of them are inevitable, but that grants no license to resolve the disagreements by making arbitrary choices. Every real (not logical) function that produces the same floating-point result for all finite and infinite numerical values of an argument should yield the same result when that argument is NaN. (Recall hypot above.)

The exceptions are C predicates " `x == x` " and " `x != x` ", which are respectively 1 and 0 for every infinite or finite number x but reverse if x is Not a Number (NaN); these provide the only simple unexceptional distinction between NaNs and numbers in languages that lack a word for NaN and a predicate IsNaN(x). Over-optimizing compilers that substitute 1 for `x == x` violate IEEE 754.

IEEE 754 assigns values to all relational expressions involving NaN . In the syntax of C , the predicate `x != y` is True but all others, `x < y` , `x <= y` , `x == y` , `x >= y` and `x > y` , are False whenever x or y or both are NaN, and then all but `x != y` and `x == y` are INVALID operations too and must so signal. Ideally, expressions `x !< y` , `x !<= y` , `x !>= y` , `x !> y` and `x !>=< y` should be valid and quietly True if x or y or both are NaN , but arbiters of taste and fashion for ANSI Standard C have refused to recognize such expressions. In any event, `!(x < y)` differs from `x >= y` when NaN is involved, though rude compilers " optimize " the difference away. Worse, some compilers mishandle NaNs in all relational expressions.

Some language standards conflict with IEEE 754. For example, APL specifies 1.0 for 0.0/0.0 ; this specification is one that APL's designers soon regretted. Sometimes naive compile-time optimizations replace expressions `x/x` by 1 (wrong if x is zero, ∞ or NaN) and `x - x` by 0 (wrong if x is ∞ or NaN) and `0*x` and `0/x` by 0 (wrong if ...), alas.

Ideally, certain other *Real* expressions unmentioned by IEEE 754 should signal INVALID and deliver NaNs ; some examples in Fortran syntax are ...

(Negative)**(Noninteger) , LOG(Negative) , ASIN(Bigger than 1) ,
SIN(∞) , ACOSH(Less than 1) , ..., all of them INVALID .

These expressions do behave that way if implemented well in software that exploits the transcendental functions built into the 680x0 and ix87 ; here i387 and successors work better than 8087 and 80287.

A number of real expressions are sometimes implemented as INVALID by mistake, or declared Undefined by ill-considered language standards; a few examples are ...

0.0**0.0 = ∞**0.0 = NaN**0.0 = 1.0 , not NaN ;
COS(2.0**120) = -0.9258790228548378673038617641... , not NaN.

More examples like these will be offered under DIVIDE by ZERO below.

Some familiar functions have yet to be defined for NaN . For instance max{x, y} should deliver the same result as max{y, x} but almost no implementations do that when x is NaN . There are good reasons to define max{NaN, 5} := max{5, NaN} := 5 though many would disagree.

Differences of opinion persist about whether certain functions should be INVALID or defined by convention at internal discontinuities; a few examples are ...

1.0**∞ = (-1.0)**∞ = 1.0 ?	(NaN is better.)
ATAN2(0.0, 0.0) = 0.0 or +π or -π ?	(NaN is worse.)
ATAN2(+∞, +∞) = π/4 ?	(NaN is worse.)
SIGNUM(0.0) = 0.0 or +1.0 or -1.0 or NaN ?	(0.0 is best.)
SGN(0.0) = 0 .	(Standard BASIC)
SIGN(+0.0) = SIGN(-0.0) = +1.0 .	(Fortran Standard)
CopySign(1.0, ±0.0) = ±1.0 respectively.	(IEEE 754/854)

As time passes, so do disputes over the value that should be assigned to a function at a discontinuity. For example, a consensus is growing that x**0 = 1 for every x , including 0 , ∞ and NaN . If some day we agree that 1**x = 1 for every x , then 1**NaN = 1 will follow; but for the time being NaN is the preferred alternative for 1**NaN , and for 1**∞ too provided it signals. It seems unlikely that 0 will ever be preferred to NaN for sign(NaN). And yet, unwise choices continue to be inflicted upon us with the best of intentions, so the struggle to correct them is unending.

> Between 1964 and 1970 the U.S. National Bureau of Standards changed its definition of arccot(x) from π/2 - arctan(x) to arctan(1/x) , thereby introducing a jump at x = 0 . This change appears to be a bad idea, but it is hard to argue with an arm of the U.S. government.

Some programmers think invoking language locutions that enable the trap to abort upon INVALID operations is the safe way to avoid all such disputes; they are mistaken. Doing so may abort searches prematurely. For example, try to find a positive root x of an equation like

(TAN(x) - ASIN(x))/x**4 = 0.0

by using Newton's iteration or the Secant iteration starting from various first guesses between 0.1 and 0.9 . In general, a root-finder that does not know the boundary of an equation's domain must be doomed to abort, if it probes a wild guess thrown outside that domain, unless it can respond to NaN by retracting the wild guess back toward a previous guess inside the domain. Such a root-finder is built into current Hewlett-Packard calculators that solve equations like the one above far more easily than do root-finders available on most computers.

Exception: **DIVIDE by ZERO**.

This is a misnomer perpetrated for historical reasons. A better name for this exception is

" Infinite result computed Exactly from Finite operands. "

An example is 3.0/0.0 , for which IEEE 754 specifies an Infinity as the default result. The sign bit of that result is, as usual for quotients, the exclusive OR of the operands' sign bits. Since 0.0 can have either sign, so can ∞; in fact, division by zero is the only algebraic operation that reveals the sign of zero. (IEEE 754 recommends a non-algebraic function CopySign to reveal a sign without ever signaling an exception, but few compilers offer it, alas.)

Ideally, certain other real expressions should be treated just the way IEEE 754 treats divisions by zero, rather than all be misclassified as errors or " Undefined "; some examples in Fortran syntax are ...

(±0.0)**(NegativeNonInteger) = +∞ ,
(±0.0)**(NegativeEvenInteger) = +∞ ,
(±0.0)**(NegativeOddInteger) = ±∞ resp.,
ATANH(±1.0) = ±∞ resp.,
LOG(±0.0) = -∞ .

The sign of ∞ may be accidental in some cases; for instance, if TANdeg(x) delivers the TAN of an angle x measured in degrees, then

TANdeg(90.0 + 180*Integer)

is infinite with a sign that depends upon details of the implementation. Perhaps that sign might best match the sign of the argument, but no such convention exists yet. (For x in radians, accurately implemented TAN(x) need never be infinite !)

Compilers can cause accidents by evaluating expressions carelessly. For example, when y resides in a register, evaluating x-y as -(y-x) reverses the sign of zero if y = x ; evaluate it as -y + x instead. Simplifying x+0 to x misbehaves when x is -0 . Doing that, or printing -0 without its sign, can obscure the source of a -∞.

Operations that produce an infinite result from an infinite operand or two must not signal DIVIDE by ZERO. Examples include

∞ + 3 , ∞*∞ , EXP(+∞) , LOG(+∞) , ...

Neither can

3.0/∞ = EXP(-∞) = 0.0 , ATAN(±∞) = ±π/2 ,

and similar examples be regarded as exceptional. Unfortunately, naive implementations of complex arithmetic can render ∞ dangerous; for instance, when $(0 + 3\imath)/\infty$ is turned naively into $(0 + 3\imath)(\infty - \imath 0)/(\infty^2 + 0^2)$ it generates a NaN instead of the expected 0 ; MATLAB suffers from this affliction.

If all goes well, infinite intermediate results will turn quietly into correct finite final results that way. If all does not go well, Infinity will turn into NaN and signal INVALID. Unlike integer division by zero, for which no integer infinity nor NaN has been provided, floating-point division by zero poses no danger provided subsequent INVALID signals, if any, are heeded; in that case disabling the trap for DIVIDE by ZERO is quite safe.

...... Digression on Division-by-Zero ......

Schools teach us to abhor Division-by-Zero and to stand in awe of the Infinite. Actually, adjoining Infinity to the real numbers adds nothing worse than another exception to the familiar cancellation laws

$$(1/x)x = x/x = 1 , \quad x-x = 0 ,$$

among which the first is already violated by x = 0 . That is a small inconvenience compared with the circumlocutions we would resort to if Infinity were outlawed. Two examples to show why are offered below.

The first example shows how Infinity eases the numerical solution of a differential equation that appears to have no divisions in it. The problem is to compute y(10) where y(t) satisfies the Ricatti equation

$$dy/dt = t + y^2 \quad \text{for all} \quad t \ge 0 , \quad y(0) = 0 .$$

Let us pretend not to know that y(t) may be expressed in terms of Bessel functions J... , whence y(10) = -7.53121 10731 35425 34544 97349 58··· . Instead a numerical method will be used to solve the differential equation approximately and as accurately as desired if enough time is spent on it.

Q(θ, t, Y) will stand for an *Updating Formula* that advances from any estimate $Y \approx y(t)$ to a later estimate $Q(\theta, t, Y) \approx y(t+\theta)$. Vastly many updating formulas exist; the simplest that might be applied to solve the given Ricatti equation would be Euler's formula:

$$Q(\theta, t, Y) := Y + \theta\cdot(t + Y^2) .$$

This " First-Order " formula converges far too slowly as *stepsize* θ shrinks; a faster " Second-Order " formula, of Runge-Kutta type, is Heun's :

$$f := t + Y^2 ; \qquad q := Y + \theta\cdot f ;$$
$$Q(\theta, t, Y) := Y + (f + t+\theta + q^2)\cdot\theta/2 .$$

Formulas like these are used widely to solve practically all ordinary differential equations. Every updating formula is intended to be iterated with a sequence of stepsizes θ that add up to the distance to be covered; for instance, Q(...) may be iterated N times with constant stepsize θ := 10/N to produce $Y(n\cdot\theta) \approx y(n\cdot\theta)$ thus:

$$Y(0) := y(0) ;$$
$$\text{for } n = 1 \text{ to } N \text{ do } Y(n\cdot\theta) := Q(\theta, (n-1)\cdot\theta, Y((n-1)\cdot\theta)) .$$

Here the number N of *timesteps* is chosen with a view to the desired accuracy since the error Y(10) - y(10) normally approaches 0 as N increases to Infinity. Were Euler's formula used, the error in its final estimate Y(10) would normally decline as fast as 1/N ; were Heun's, ... $1/N^2$. But the Ricatti differential equation is not normal; no matter how big the number N of steps, those formulas' estimates Y(10) turn out to be huge positive numbers or overflows instead of -7.53··· . Conventional updating formulas do not work here.

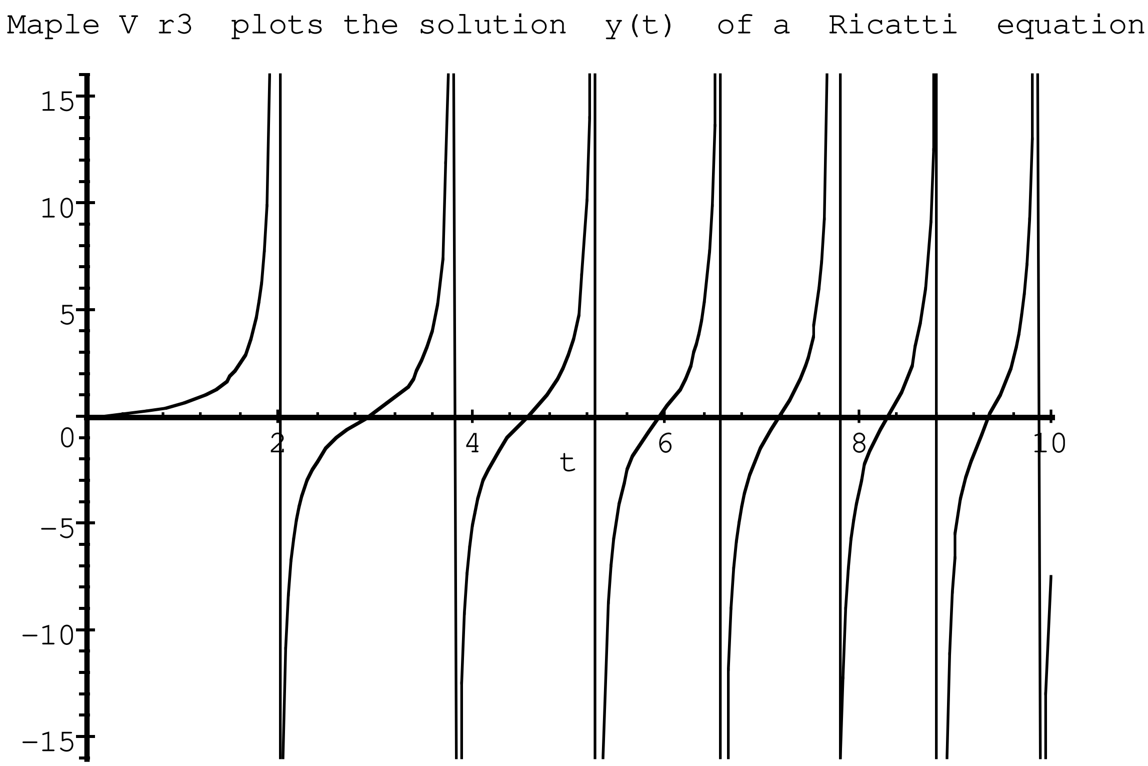
The simplest unconventional updating formula Q available turns out to be this rational formula:

$$Q(\theta, t, Y) := Y + (t + \tfrac{1}{2}\theta + Y^2)\cdot\theta/(1 - \theta\cdot Y) \qquad \text{if} \quad |\theta\cdot Y| < \tfrac{1}{2} ,$$
$$:= (1/\theta + (t + \tfrac{1}{2}\theta)\cdot\theta)/(1 - \theta\cdot Y) - 1/\theta \qquad \text{otherwise.}$$

The two algebraically equivalent forms are distinguished to curb rounding errors. Like Heun's, this Q is a second-order formula. (It can be compounded into a formula of arbitrarily high order by means that lie beyond the scope of these notes.) Iterating it N times with stepsize θ := 10/N yields a final estimate Y(10) in error by roughly $(105/N)^2$ even if Division-by-Zero insinuates an Infinity among the iterates Y(n·θ) . Disallowing Infinity and Division-by-Zero would at least somewhat complicate the estimation of y(10) because y(t) has to pass through Infinity seven times as t increases from 0 to 10 . (See the graph on the next page.)

What becomes complicated is not the program so much as the process of developing and verifying a program that can dispense with Infinity. First, find a very tiny number ε barely small enough that $1 + 10\sqrt{\varepsilon}$ rounds off to 1 . Next, modify the foregoing rational formula for Q by replacing the divisor (1 - θ·Y) in the " otherwise " case by ((1 - θ·Y) + ε) . Do not omit any of these parentheses; they prevent divisions by zero. Then perform an error-analysis to confirm that iterating this formula produces the same values Y(n·θ) as would be produced without ε except for replacing infinite values Y by huge finite values.

Survival without Infinity is always possible since " Infinity " is just a short word for a lengthy explanation. The price paid for survival without Infinity is lengthy cogitation to find a not-too-lengthy substitute, if it exists.



End of first example.

The second example that illustrates the utility of Infinity is part of the fastest program known for computing a few eigenvalues of a real symmetric matrix. This part can be programmed to run well on every commercially significant computer that conforms to IEEE 754, but not in any single higher-level language that all such computers recognize.

$$\text{Let} \quad T = \begin{bmatrix} a[1] & b[2] & & & & \\ b[2] & a[2] & b[3] & & & \\ & b[3] & a[3] & b[4] & & \\ & & b[4] & a[4] & \dots & \\ & & & \dots & \dots & b[n] \\ & & & & b[n] & a[n] \end{bmatrix} .$$

No $b[j] = 0$.

Let $q[1] := 0$ and

$$q[j] := b[j]^2 > 0$$

for $1 < j \le n$.

Every real symmetric matrix reduces quickly to a tridiagonal form like T with the same eigenvalues $\tau[1] < \tau[2] < \ldots < \tau[n]$. The task is to find some of them specified either by an interval in which they lie or by their indices. Typically a few dozen eigenvalues may be sought when the dimension n is in the thousands. For this task the fastest and most accurate algorithm known is based upon the properties of two functions $f = f(\sigma)$ and $k = k(\sigma)$ defined for real variable σ thus:

```
σ := σ + 0 ;                                  |  If σ = 0 , this ensures it is +0 ,
 k := n ;  f := 1 ;                           |
 FOR j = 1, 2, 3, ..., n IN TURN,             |
   DO { f := ( σ - a [ j ] ) - q [ j ]/f ;    |  Note: This loop has no explicit
          k := k - SignBit(f)       } ;       |        tests nor branches.
 k(σ) := k ;  f (σ) := f .                    |
```

(The function SignBit(f) := 0 if $f > 0$ or if f is +0 or -0 ,
:= 1 if $f < 0$ or if f is -0 ;
its value at f = -0 may not matter but its value at $f = -\infty$ does. It can be computed by using logical right-shifts or by using
(f < 0.0) in C , or
0.5 - SIGN(0.5, f) in Fortran, or
0.5 - CopySign(0.5, f) from IEEE 754 .
However, the use of shifts or CopySign is mandatory on computers that depart from IEEE 754 by flushing UNDERFLOWed subtractions to -0.0 instead of UNDERFLOWing Gradually, q. v. below.
Through an unfortunate accident, the arguments of CopySign are reversed on Apple computers, which otherwise conform conscientiously to IEEE 754; they require SignBit(f) := 0.5 - CopySign(f, 0.5) .)

The function $f(\sigma)$ is a *continued fraction*:

$$f(\sigma) = \sigma - a[n] - \cfrac{b[n]^2}{\sigma - a[n-1] - \cfrac{b[n-1]^2}{\sigma - a[n-2] - \cfrac{b[n-2]^2}{\sigma - a[n-3] - \ldots - \cfrac{b[2]^2}{\sigma - a[1]}}}} \; .$$

The eigenvalues $\tau[j]$ of T are the zeros of $f(\sigma)$, separated by the poles of $f(\sigma)$ at which it interrupts its monotonic increasing behavior to jump from $+\infty$ to $-\infty$. The integer function $k(\sigma)$ counts the eigenvalues on either side of σ thus:

$$\tau[1] < \tau[2] < \ldots < \tau[k(\sigma)] \le \sigma < \tau[k(\sigma)+1] < \ldots < \tau[n] , \quad \text{and}$$
$$\tau[k(\sigma)] = \sigma \quad \text{just when} \quad f(\sigma) = 0 .$$

Evidently the eigenvalues $\tau[j]$ of T are the n values of σ at which $k(\sigma)$ jumps, and may be located by Binary Search accelerated by interpolative schemes that take values of $f(\sigma)$ into account too. Although rounding errors can obscure $f(\sigma)$ severely, its monotonicity and the jumps of $k(\sigma)$ are practically unaffected, so the eigenvalues are obtained more accurately this way than any other. And quickly too.

If Infinity were outlawed, the loop would have to be encumbered by a test and branch to prevent Division-by-Zero. That test cannot overlap the division, a slow operation, because of sequential dependencies, so the test would definitely slow down the loop even though zero would be detected extremely rarely. The test's impact would be tolerable if the loop contained only one division, but that is not what happens.

Because division is so slow, fast computers pipeline it in such a way that a few divisions can be processed concurrently. To exploit this pipelining, we search for several eigenvalues simultaneously. The variable σ becomes a small array of values, as do f and k , and every go-around the loop issues an array of divisions. In this context the tests, though " vectorized " too, degrade speed by 25% or more, much more on machines with multiple pipelines that can subtract and shift concurrently with division, regardless of what else a branch would entail whenever a zero divisor were detected. By dispensing with those tests, this program gains speed and simplicity from Infinity even if Division-by-Zero never happens.

End of second example.

How many such examples are there? Nobody knows how many programs would benefit from Infinity because it remains unsupported by programming language standards, and hence by most compilers, though support in hardware has been abundant for over a decade. To get some idea of the impediment posed by lack of adequate support, try to program each of the foregoing two examples in a way that will compile correctly on every machine whose hardware conforms to IEEE 754. That ordeal will explain why few programmers experiment with Infinity, whence few programs use it.

In my experience with a few compilers that support IEEE 754 on PCs and Macintoshes, Infinity and NaNs confer their greatest benefits by simplifying test programs, which already tend to grossly worse complexity than the software they are designed to validate. Consequently my programs enjoy enhanced reliability because of IEEE 754 regardless of whether it is in force where they run.

...... End of Digression ......

Exception: **OVERFLOW.**

This happens after an attempt to compute a finite result that would lie beyond the finite range of the floating-point format for which it is destined. The default specified in IEEE 754 is to approximate that result by an appropriately signed Infinity. Since it is approximate, OVERFLOW is also INEXACT. Often that approximation is worthless; it is almost always worthless in matrix computations, and soon turns into NaN or, worse, misleading numbers. Consequently OVERFLOW is often trapped to abort seemingly futile computation sooner rather than later.

Actually, OVERFLOW more often implies that a different computational path should be chosen than that no path leads to the desired goal. For example, if the Fortran expression x / SQRT(x*x + y*y) encounters OVERFLOW before the SQRT can be computed, it should be replaced by something like

(s*x) / SQRT((s*x)*(s*x) + (s*y)*(s*y))

with a suitably chosen tiny positive Scale-Factor s . (A power of 2 avoids roundoff.) The cost of computing and applying s beforehand could be deemed the price paid for insurance against OVERFLOW. Is that price too high?

The biggest finite IEEE 754 Double is almost 1.8 e308 , which is so huge that OVERFLOW occurs extremely rarely if not yet rarely enough to ignore. The cost of defensive tests, branches and scaling to avert OVERFLOW seems too high a price to pay for insurance against an event that hardly ever happens. A lessened average cost will be incurred in most situations by first running without defensive scaling but with a judiciously placed test for OVERFLOW (and for severe UNDERFLOW); in the example above the test should just precede the SQRT. Only when necessary need scaling be instituted. Thus our treatment of OVERFLOW has come down to this question: how best can OVERFLOW be detected?

The ideal test for OVERFLOW tests its flag; but that flag cannot be mentioned in most programming languages for lack of a name. Next best are tests for Infinities and NaNs consequent upon OVERFLOW, but prevailing programming languages lack names for them; suitable tests have to be contrived. For example, the C predicate `(z != z)` is True only when z is NaN and the compiler has not " optimized " overzealously; `(1.0e37 /(1 + fabs(z)) == 0)` is True only when z is infinite; and `(z-z != 0)` is True only when z is NaN or infinite, the INVALID trap has been disabled, and optimization is not overzealous.

A third way to detect OVERFLOW is to enable its trap and attach a handler to it. Even if a programming language in use provides control structures for this purpose, this approach is beset by hazards. The worst is the possibility that the handler may be entered inadvertently from unanticipated places. Another hazard arises from the concurrent execution of integer and floating-point operations; by the time an OVERFLOW has been detected, data associated with it may have become inaccessible because of changes in pointers and indices. Therefore this approach works only when a copy of the data has been saved to be reprocessed by a different method than the one thwarted by OVERFLOW, and when the scope of the handler has been properly localized; note that the handler must be detached before and reattached after functions that handle their own OVERFLOWs are executed.

The two costs, of saving and scoping, must be paid all the time even though OVERFLOW rarely occurs. For these reasons and more, other approaches to the OVERFLOW problem are to be preferred, but a more extensive discussion of them lies beyond the intended scope of this document.

When OVERFLOW's trap is enabled, the IEEE 754 default Infinity is not generated; instead, the result's exponent is " wrapped," which means in this case that the delivered result has an exponent too small by an amount $2^{K-1} \cdot 3$ that depends upon its format:

Double-Extended	... too small by 24576 ;	2^{24576} = 1.3 E 7398
Double	... too small by 1536 ;	2^{1536} = 2.4 E 462
Single	... too small by 192 ;	2^{192} = 6.3 E 57

(Though required by IEEE 754, the latter two are not performed by ix87 nor 680x0 nor some other machines without help from suitable trap-handling software.)

In effect, the delivered result has been divided by a power of 2 so huge as to turn what would have overflowed into a relatively small but predictable quantity that a trap-handler can reinterpret. If there is no trap handler, computation will proceed with that smaller quantity or, in the case of format-converting FSTore instructions, without storing anything. The reason for exponent wrapping is explained after UNDERFLOW.

Exception: **UNDERFLOW**.

This happens after an attempt to approximate a nonzero result that lies closer to zero than the intended floating-point destination's " Normal " positive number nearest zero. 2.2 e-308 is that number for IEEE 754 Double. A nonzero Double result tinier than that must by default be rounded to a nearest Subnormal number, whose magnitude can run from 2.2 e-308 down to 4.9 e-324 (but with diminishing precision), or else by 0.0 when no Subnormal is nearer. IEEE 754 Extended and Single formats have different UNDERFLOW thresholds, for which see the table " Span and Precision of IEEE 754 Floating-Point Formats " above. If that rounding incurs no error, no UNDERFLOW is signaled.

Subnormal numbers, also called " Denormalized," allow UNDERFLOW to occur Gradually; this means that gaps between adjacent floating-point numbers do not widen suddenly as zero is passed. That is why Gradual UNDERFLOW incurs errors no worse in absolute magnitude than rounding errors among Normal numbers. No such property is enjoyed by older schemes that, lacking Subnormals, flush UNDERFLOW to zero abruptly and suffer consequent anomalies more fundamental than afflict Gradual UNDERFLOW.

For example, the C predicates `x == y` and `x-y == 0` are identical in the absence of OVERFLOW only if UNDERFLOW is Gradual. That is so because `x-y` cannot UNDERFLOW Gradually; if `x-y` is Subnormal then it is Exact. Without Subnormal numbers, x/y might be 0.95 and yet x-y could UNDERFLOW abruptly to 0.0 , as could happen for x and y tinier than 20 times the tiniest nonzero Normal number. Consequently, Gradual Underflow simplifies a theorem very important for the attenuation of roundoff in numerical computation:

> If p and q are floating-point numbers in the same format, and if $1/2 \le p/q \le 2$,
> then p - q is computable exactly (without a rounding error) in that format. But
> if UNDERFLOW is not Gradual, and if p - q UNDERFLOWs, it is not exact.

More generally, floating-point error-analysis is simplified by the knowledge, first, that IEEE 754 rounds every finite floating-point result to its best approximation by floating-point numbers of the chosen destination's format, and secondly that the approximation's absolute uncertainty (error bound) cannot increase as the result diminishes in magnitude. Error-analysis, and therefore program validation, is more complicated, sometimes appallingly more so, on those currently existing machines that do not UNDERFLOW Gradually.

Though afflicted by fewer anomalies, Gradual UNDERFLOW is not free of them. For instance, it is possible to have `x/y == 0.95` coexist with `(x*z)/(y*z) == 0.5` because `x*z` and probably also `y*z` UNDERFLOWed to Subnormal numbers; without Subnormals the last quotient turns into either an ephemeral 0.0 or a persistent NaN (INVALID 0/0). Thus, UNDERFLOW cannot be ignored entirely whether Gradual or not.

UNDERFLOWs are uncommon. Even if flushed to zero they rarely matter; if handled Gradually they cause harm extremely rarely. That harmful remnant has to be treated much as OVERFLOWs are, with testing and scaling, or trapping, etc. However, the most common treatment is to ignore UNDERFLOW and then to blame whatever harm is caused by doing so upon poor taste in someone else's choice of initial data.

UNDERFLOWs resemble ants; where there is one there are quite likely many more, and they tend to come one after another. That tendency has no direct effect upon the i387-486-Pentium nor M68881/2, but it can severely retard computation on other implementations of IEEE 754 that have to trap to software to UNDERFLOW Gradually for lack of hardware to do it. They take extra time to Denormalize after UNDERFLOW and/or, worse, to prenormalize Subnormals before multiplication or division. (Gradual UNDERFLOW requires no prenormalization before addition or subtraction of numbers with the same format, but computers usually do it anyway if they have to trap Subnormals.) Worse still is the threat of traps, whether they occur or not, to machines like the DEC Alpha that cannot enable traps without hampering pipelining and/or concurrency; such machines are slowed also by Gradual UNDERFLOWs that do not occur!

Why should we care about such benighted machines? They pose dilemmas for developers of applications software designed to be portable (after recompilation) to those machines as well as ours. To avoid sometimes severe performance degradation by Gradual UNDERFLOW, developers will sometimes resort to simple-minded alternatives. The simplest violates IEEE 754 by flushing every UNDERFLOW to 0.0 , and computers are being sold that flush by default. (DEC Alpha is a recent example; it is advertised as conforming to IEEE 754 without mention of how slowly it runs with traps enabled for full conformity.) Applications designed with flushing in mind may, when run on ix87s and Macs, have to enable the UNDERFLOW trap and provide a handler that flushes to zero, thereby running slower to get generally worse results! (This is what MathCAD does on PCs and on Macintoshes.) Few applications are designed with flushing in mind nowadays; since some of these might malfunction if UNDERFLOW were made Gradual instead, disabling the ix87 UNDERFLOW trap to speed them up is not always a good idea.

A format's usable exponent range is widened by almost its precision N to fully $\pm 2^K$ as a by-product of Gradual Underflow; were this its sole benefit, its value to formats wider than Single could not justify its price. Compared with Flush-to-Zero, Gradual Underflow taxes performance unless designers expend time and ingenuity or else hardware. Designers incapable of one of those expenditures but willing to cut a corner off IEEE 754 exacerbate market fragmentation, which costs the rest of us cumulatively far more than whatever they saved.

...... Digression on Gradual Underflow ......

To put things in perspective, here is an example of a kind that, when it appears in benchmarks, scares many people into choosing Flush-to-Zero rather than Gradual UNDERFLOW. To simulate the diffusion of heat through a conducting plate with edges held at fixed temperatures, a rectangular mesh is drawn on the plate and temperatures are computed only at intersections. The finer the mesh, the more accurate is the simulation. Time is discretized too; at each timestep, temperature at every interior point is replaced by a positively weighted average of that point's temperature and those of nearest neighbors. Simulation is more accurate for smaller timesteps, which entail larger numbers of timesteps and tinier weights on neighbors; typically these weights are smaller than 1/8 , and timesteps number in the thousands.

When edge temperatures are mostly fixed at 0 , and when interior temperatures are mostly initialized to 0 , then at every timestep those nonzero temperatures next to zeros get multiplied by tiny weights as they diffuse to their neighbors. With fine meshes, large numbers of timesteps can pass before nonzero temperatures have diffused almost everywhere, and then tiny weights can get raised to large powers, so UNDERFLOWs are numerous. If UNDERFLOW is Gradual, denormalization will produce numerous subnormal numbers; they slow computation badly on a computer designed to handle subnormals slowly because the designer thought they would be rare. Flushing UNDERFLOW to zero does not slow computation on such a machine; zeros created that way may speed it up.

When this simulation figures in benchmarks that test computers' speeds, the temptation to turn slow Gradual UNDERFLOW Off and fast Flush-to-Zero On is more than a marketing manager can resist. Compiler vendors succumb to the same temptation; they make Flush-to-Zero their default. Such practices bring to mind calamitous explosions that afflicted high-pressure steam boilers a century or so ago because attendants tied down noisy over-pressure relief valves the better to sleep undisturbed.

```
+--------------------------------------------------------------------+
|  Vast numbers of  UNDERFLOWs  usually signify that something about  |
|  a program or its data is strange if not wrong;  this signal should |
|  not be ignored,  much less squelched by flushing  UNDERFLOW  to  0.|
+--------------------------------------------------------------------+
```

What is strange about the foregoing simulation is that exactly zero temperatures occur rarely in Nature, mainly at the boundary between colder ice and warmer water. Initially increasing all temperatures by some negligible amount, say 10^{-30} , would not alter their physical significance but it would eliminate practically all UNDERFLOWs and so render their treatment, Gradual or Flush-to-Zero, irrelevant.

To use such atypical zero data in a benchmark is justified only if it is intended to expose how long some hardware takes to handle UNDERFLOW and subnormal numbers. Unlike many other floating-point engines, the i387 and its successors are slowed very little by subnormal numbers; we should thank Intel's engineers for that and enjoy it rather than resort to an inferior scheme which also runs slower on the i387-etc.

...... End of Digression ......

When UNDERFLOW's trap is enabled, the IEEE 754 default Gradual Underflow does not occur; the result's exponent is “ wrapped ” instead, which means in this case that the delivered result has an exponent too big by an amount $2^{K-1}\cdot 3$ that depends upon its format:

Double-Extended	... too big by 24576 ;	$2^{24576} = 1.3$ E 7398
Double	... too big by 1536 ;	$2^{1536} = 2.4$ E 462
Single	... too big by 192 ;	$2^{192} = 6.3$ E 57

(Though required by IEEE 754, the latter two wraps are not performed by ix87 nor 680x0 nor some other machines without help from suitable trap-handling software.)

In effect, the delivered result has been multiplied by a power of 2 so huge as to turn what would have underflowed into a relatively big but predictable quantity that a trap-handler can reinterpret. If there is no trap handler, computation will proceed with that bigger quantity or, in the case of format-converting FSTore instructions, without storing anything.

Exponent wrapping provides the fastest and most accurate way to compute extended products and quotients like

$$Q = \frac{(a_1 + b_1)\cdot(a_2 + b_2)\cdot(a_3 + b_3)\cdot(\ldots)\cdot(a_N + b_N)}{(c_1 + d_1)\cdot(c_2 + d_2)\cdot(c_3 + d_3)\cdot(\ldots)\cdot(c_M + d_M)}$$

when N and M are huge and when the numerator and/or denominator are likely to encounter premature OVER/UNDERFLOW even though the final value of Q would be unexceptional if it could be computed. This situation arises in certain otherwise attractive algorithms for calculating eigensystems, or Hypergeometric series, for example.

What Q requires is an OVER/UNDERFLOW trap-handler that counts OVERFLOWs and UNDERFLOWs but leaves wrapped exponents unchanged during each otherwise unaltered loop that computes separately the numerator's and denominator's product of sums. The final quotient of products will have the correct significant bits but an exponent which, if wrong, can be corrected by taking counts into account. This is by far the most satisfactory way to compute Q , but for lack of suitable trap-handlers it is hardly ever exploited though it was implemented on machines as diverse as the IBM 7094 and /360 (by me in Toronto in the 1960s; see Sterbenz (1974)), a Burroughs B5500 (by Michael Green at Stanford in 1966), and a DEC VAX (in 1981 by David Barnett, then an undergraduate at Berkeley). Every compiler seems to require its own implementation.

Exception: **INEXACT**.

This is signaled whenever the ideal result of an arithmetic operation would not fit into its intended destination, so the result had to be altered by rounding it off to fit. The INEXACT trap is disabled and its flag ignored by almost all floating-point software. Its flag can be used to improve the accuracy of extremely delicate approximate computations by " Exact Preconditioning," a scheme too arcane to be explained here; for an example see pp. 107-110 of Hewlett-Packard's HP-15C *Advanced Functions Handbook* (1982) #00015-90011. Another subtle use for the INEXACT flag is to indicate whether an equation f(x) = 0 is satisfied exactly *without* roundoff (in which case x is exactly right) or *despite* roundoff (in which not–so–rare case x may be arbitrarily erroneous).

A few programs use REAL variables for integer arithmetic. M680x0s and ix87s can handle integers up to 65 bits wide including sign, and convert all narrower integers to this format on the fly. In consequence, arithmetic with wide integers may go faster in floating-point than in integer registers at most 32 bits wide. But then when an integer result gets too wide to fit exactly in floating-point it will be rounded off. If this rounding went unnoticed it could lead to final results that were all unaccountably multiples of, say, 16 for lack of their last few bits. Instead, the INEXACT exception serves in lieu of an INTEGER OVERFLOW signal; it can be trapped or flagged.

Well implemented Binary-Decimal conversion software signals INEXACT just when it is deserved, just as rational operations and square root do. However, an undeserved INEXACT signal from certain transcendental functions like X**Y when an exact result is delivered accidentally can be very difficult to prevent.

Directions of Rounding:

The default, reset by turning power off-on, rounds every arithmetic operation to the nearest value allowed by the assigned precision of rounding. When that nearest value is ambiguous (because the exact result would be one bit wider than the precision calls for) the rounded result is the " even " one with its last bit zero. Note that rounding to the nearest 16-, 32- or 64-bit integer (float-to-int store) in this way takes both 1.5 and 2.5 to 2 , so the various INT, IFIX, ... conversions to integer supported by diverse languages may require something else. One of my Fortran compilers makes the following distinctions among roundings to nearest integers:

IRINT, RINT, DRINT	round to nearest even, as ix87 FIST does.
NINT, ANINT, DNINT	round half-integers away from 0 .
INT, AINT, DINT	truncate to integers towards 0 .

Rounding towards 0 causes subsequent arithmetic operations to be truncated, rather than rounded, to the nearest value in the direction of 0.0 . In this mode, store-to-int provides INT etc. This mode also resembles the way many old machines now long gone used to round.

The " Directed " roundings can be used to implement Interval Arithmetic, which is a scheme that approximate every variable not by one value of unknown reliability but by two that are guaranteed to straddle the ideal value. This scheme is not so popular in the U.S.A. as it is in parts of Europe, where some people distrust computers.

Control-Word control of rounding modes allows software modules to be re-run in different rounding modes without recompilation. This cannot be done with some other computers, notably DEC Alpha, that can set two bits in every instruction to control rounding direction at compile-time; that is a mistake. It is worsened by the designers' decision to take rounding direction from a Control-Word when the two bits are set to what would otherwise have been one of the directed roundings; had Alpha obtained only the round-to-nearest mode from the Control-Word, their mistake could have been transformed into an advantageous feature.

All these rounding modes round to a value drawn from the set of values representable either with the precision of the destination or selected by rounding precision control to be described below. The sets of representable values were spelled out in tables above. The direction of rounding can also affect OVER/UNDERFLOW ; a positive quantity that would OVERFLOW to $+\infty$ in the default mode will turn into the format's biggest finite floating-point number when rounded towards $-\infty$. And the expression " X - X " delivers +0.0 for every finite X in all rounding modes except for rounding directed towards $-\infty$, for which -0.0 is delivered instead. These details are designed to make Interval Arithmetic work better.

Ideally, software that performs Binary-Decimal conversion (both ways) should respect the requested direction of rounding. David Gay of AT&T Bell Labs has released algorithms that do so into the public domain (Netlib); to use less accurate methods now is a blunder.

Precisions of Rounding:

IEEE 754 obliges only machines that compute in the Extended (`long double` or REAL*10) format to let programmers control the precision of rounding from a Control-Word. This lets ix87 or M680x0 emulate the roundoff characteristics of other machines that conform to IEEE 754 but support only Single (C's `float`, or REAL*4) and Double (C's `double`, or REAL*8), not Extended. Software developed and checked out on one of those machines can be recompiled for a 680x0 or ix87 and, if anomalies excite concerns about differences in roundoff, the software can be run very nearly as if on its original host without sacrificing speed on the 680x0 or ix87. Conversely, software developed on these machines but without explicit mention of Extended can be rerun in a way that portends what it will do on machines that lack Extended. Precision Control rounds to 24 sig. bits to emulate Single, to 53 sig. bits to emulate Double, leaving zeros in the rest of the 64 sig. bits of the Extended format.

The emulation is imperfect. Transcendental functions are unlikely to match. Although Decimal -> Binary conversion must round to whatever precision is set by the Control-Word, Binary -> Decimal should ideally be unaffected since its precision is determined solely by the destination's format, but ideals are not always attained. Some OVER/UNDERFLOWs that would occur on those other machines need not occur on the ix87 ; IEEE 754 allows this, perhaps unwisely, to relieve hardware implementors of details formerly thought unimportant.

Few compilers expose the Control-Word to programmers. Worse, some compilers have revived a nasty bug that emerged when Double-Precision first appeared among Fortran compilers; it goes like this: Consider

```
S = X
T = ( S - Y )/( . . . )
```

in a Fortran program where S is SINGLE PRECISION, and X and Y are DOUBLE or EXTENDED PRECISION variables or expressions computed in registers. Compilers that supplant S by X in the second statement save the time required to reload S from memory but spoil T . Though S and X differ by merely a rounding error, the difference matters.

The Baleful Influence of Benchmarks:

Hardware and compilers are increasingly being rated entirely according to their performance in benchmarks that measure only speed. That is a mistake committed because speed is so much easier to measure than other qualities like reliability and convenience. Sacrificing them in order to run faster will compel us to run longer. By disregarding worthwhile qualities other than speed, current benchmarks penalize conscientious adherence to standards like IEEE 754; worse, attempts to take those qualities into account are thwarted by political constraints imposed upon programs that might otherwise qualify as benchmarks.

For example, a benchmark should compile and run on every commercially significant computer system. This rules out our programs for solving the differential equation and the eigenvalue problem described above under the Digression on Division-by-Zero. To qualify as benchmarks, programs must prevent exceptional events that might stop or badly slow some computers even if such prevention retards performance on computers that, by conforming conscientiously to IEEE 754, would not stop.

The Digression on Gradual Underflow offered an example of a benchmark that lent credibility to a misguided preference for Flush-to-Zero, in so far as it runs faster than Gradual Underflow on some computers, by disregarding accuracy. If Gradual Underflow's superior accuracy has no physical significance there, neither has the benchmark's data.

Accuracy poses tricky questions for benchmarks. One hazard is the ...

> Stopped Clock Paradox: Why is a mechanical clock more accurate stopped than running?
> A running clock is almost never exactly right, whereas a stopped clock is exactly right twice a day.
> (*But WHEN is it right? Alas, that was not the question.*)

The computational version of this paradox is a benchmark that penalizes superior computers, that produce merely excellent approximate answers, by making them seem less accurate than an inferior computer that gets exactly the right answer for the benchmark's problem accidentally. Other hazards exist too; some will be illustrated by the next example.

Quadratic equations like

$$p\,x^2 - 2\,q\,x + r = 0$$

arise often enough to justify tendering a program that solves it to serve as a benchmark. When the equation's roots x1 and x2 are known in advance both to be real, the simplest such program is the procedure `Qdrtc` exhibited on the next page.

In the absence of premature Over/Underflow, `Qdrtc` computes x1 and x2 at least about as accurately as they are determined by data { p, q, r } uncorrelatedly uncertain in their last digits stored. It should be tested first on trivial data to confirm that it has not been corrupted by a misprint nor by an ostensible correction like "x1 := (q+s)/p ; x2 := (q-s)/p" copied naively from some elementary programming text. Here are some trivial data:

{ p = Any nonzero, q = r = 0 }; x1 = x2 = 0 .

{ p = 2.0 , q = 5.0 , r = 12.0 }; x1 = 2.0 , x2 = 3.0 .

{ p = 2.0 E-37, q = 1.0 , r = 2.0 }; x1 ≈ 1.0 , x2 ≈ 1.0 E 37 .

Swapping p with q swaps { x1, x2 } with { 1/x2, 1/x1 } .

{µ*p, µ*q, µ*r} yields {x1, x2} independently of nonzero µ .

A Proposed Accuracy Benchmark

```
Procedure  Qtest( Qdrtc ):
  Parameter  n = 15 ;   ...  In time,  n  may grow.
  Real Array  r[1:n] ;  ...  Choose precision here.
  r[1] := 2^12 + 2.0 ;        ... for  24 sig. bits,
  r[2] := 2^12 + 2.25 ;           ...      and 6 hex.
  r[3] := 16^3 + 1 + 1.0/16^2 ; ...     6  hex. IBM.
  r[4] := 2^24 + 2.0 ;            ...     48 bits CRAY -
  r[5] := 2^24 + 2.25 ;           ...        rounded;
  r[6] := 2^24 + 3.0 ;            ...     48 bits chopped.
  r[7] := 94906267.0 ;            ...     53  sig. bits.
  r[8] := 94906267 + 0.25 ;       ...     53  sig. bits.
  r[9] := 2^28 - 5.5 ;            ...     PowerPC, i860.
  r[10] := 2^28 - 4.5 ;           ...     PowerPC, i860.
  r[11] := 2^28 + 2.0 ;           ...     56  sig. bits,
  r[12] := 2^28 + 2.25 ;          ...      and 14 hex.
  r[13] := 16^7 + 1 + 1.0/16^6 ; ...    14 hex. IBM.
  r[14] := 2^32 + 2.0 ;           ...     64  sig. bits.
  r[15] := 2^32 + 2.25 ;          ...     64  sig. bits.
  e := +Infinity ;
  for  j := 1  to  n  do {
    t := Qtrial( Qdrtc, r[j] ) ;   ...   Could be  NaN.
    If  ((t < e) or not(t = t))  then  e := t } ;
  Display( " Worst accuracy is ", e, " sig. bits" ) ;
   Return ;  End  Qtest.

Real Function  Log2(x) := Log(Abs(x))/Log(2.0) ;

Real Function  Qtrial( Qdrtc, r ):
  p := r-2 ;  q := r-1 ;  Qtrial := 0 ;
  Display( Nameof(Qdrtc), " for  r = ", r ) ;
  If  p ≤ 0  then {
    Display(" Qtrial(..., r)  expects  r > 2 .") }
  elseif  not((r-q)=1 & (q-p)=1)  then {
    Display("  r  is too big for  Qtrial(..., r).") }
  else {
    Call  Qdrtc( p, q, r,  x1, x2 ) ;
    e1 := -Log2( x1 - 1 ) ;          ...   Could be  NaN .
    e2 := -Log2( (x2 - 1) - 2/p ) ;... Heed parentheses!
    Qtrial := Min{ e1, e2 } ; ...   Min{NaN,NaN} is NaN .
    Display(" gets ", e1, " and ", e2, " sig. bits")
    If  not( x1 ≥ 1.0 )  then
      Display(" and root ", x1, " isn't at least  1.")}};
  Display(  ) ;  Return( Qtrial ) ;  End  Qtrial.


Procedure  Qdrtc( p, q, r,  x1, x2 ):
  Real  p, q, r, s, x1, x2 ;      ...   Choose precision here.
  s := √( q*q - p*r ) ;           ...   NaN  if  √( < 0 ).
  S := q + CopySign(s, q) ;       ...   Fortran's  SIGN  O.K.
  If S = 0 then { x1 := x2 := r/p }  else
     { x1 := r/S ;  x2 := S/p }; ...   NaNs  if not real,
  Return;  End  Qdrtc.            ...   or else it may abort.
```

The proposed benchmark program `Qtest` runs `Qdrtc` on a battery of data sets each chosen to expose the worst rounding error of which a computing system is capable. The system's precision appears next to its data r as an annotation in `Qtest`. Each datum r is expressed in a way that avoids damaging roundoff at the precision under test and, since r is a large positive number but not too large, the other two coefficients p := r-2 and q := r-1 are also computed uncontaminated by roundoff in function `Qtrial`. Therefore `Qtrial` knows the correct roots x1 = 1 and x2 = 1 + 2/p exactly and can compare them with the roots x1 and x2 computed by `Qdrtc` to determine its accuracy.

More important than accuracy are mathematical relationships implied by correlations among data. In this problem, inequalities $q^2 \ge p\,r$ and $0 < p < q < r$ and $p-q \ge q-r$ can all be confirmed directly by tests, and imply that both roots must be real and no less than 1.0 . When `Qdrtc` fails to honor those implications, `Qtrial` notices.

What should we expect would-be benchmark `Qtest` to find when it runs in 8-byte floating-point on some current computer systems? Tabulated under Precision is how many significant bits are stored in the named system's 8-byte format; different systems trade off precision and range differently, and this should be taken into account before one system is condemned for getting less accuracy than another. Next comes the worst Accuracy `Qtest` encountered; evidently as many as half the sig. bits stored in computed roots can be inaccurate. Worse, the smaller computed root can fall short of 1.0 in the sig. bit whose position is tabulated last. These findings cry out for explanation; how can some computer systems get worse accuracy than others that store the same number of sig. bits?

Expected Results from Qtest(Qdrtc) on 8-byte Floating-Point

Computer Hardware	Software System	Precision sig. bits	Accuracy sig. bits	How far < 1 sig. bit
ix86/87- & Pentium- based PCs	Fortran, C, Turbo-Basic, Turbo-Pascal	53	32	33.3
680x0-based Sun III, Macintosh	Fortran, C			
DEC VAX D	Fortran, C	56	28	29.3
ix86/87 & Macintosh	MATLAB 3.5, MathCAD 2.5	53	26.5	27.8
SGI MIPS, SPARC, DEC VAX G	Fortran, C , MATLAB 4.x			
IBM /370	Fortran, C	56	26.4	26.4
CRAY Y-MP	Fortran, C	48	24	25.3
Intel 860, PowerPC, IBM RS/6000	Fortran, C	53	NaN from √(< 0)	NaN from √(< 0)

The explanation is easy for the IBM /370 ; its hexadecimal floating-point loses two or three sig. bits compared with binary floating-point of the same width. No matter; these formerly ubiquitous machines are disappearing.

The best accuracy, 32 sig. bits, is achieved on inexpensive ix86/87-based PCs and 680x0-based Macintoshes whose hardware permits every algebraic (sub)expression, though no named variable wider than 8 bytes appears in it, to be evaluated in Extended registers 10 bytes wide, and by software systems (compilers) that neither disable nor eschew that capability regardless of whether they support named 10-byte variables. These computer systems also accept, without premature over/underflows, a wider range of input data $\{\mu*p, \mu*q, \mu*r\}$ than do the others, though this robustness cannot be explored by `Qtest` without crashing some systems.

MATLAB and MathCAD on ix86/87 and 680x0 platforms store almost every subexpression into internal 8-byte scratch variables, thereby wasting time as well as the 10-byte registers' superior accuracy and range; that is why their accuracy is no better on machines with 10-byte registers than on machines without.

The final mystery is the NaN (Not a Number) obtained from the i860, IBM RS/6000 and PowerPC instead of roots. The NaN arises from the square root of a negative number q*q - p*r , although tests performed upon input data would find that QQ := q*q and PR := p*r do satisfy QQ ≥ PR . This paradox arises out of the Fused Multiply-Accumulate instruction possessed by those machines. (The i860's MAC is only partially fused.) The paradox can be suppressed by inhibiting that instruction at compile time, but doing so generally would slow those machines; therefore, their compiler was designed to render that inhibition inconvenient and unusual. If `Qtest` were run on these machines in their unusual mode, would that constitute a fair test?

Fairness raises troublesome issues for a benchmark. What if custodians of a computer family allege unfairness? Letting them tweak a benchmark slightly to render it " fair " lets them overcompensate in devious ways very difficult to expose. For example, replace `Qdrtc` by an ostensibly algebraically equivalent procedure …

```
Procedure  PPCQdrtc( p, q, r,  x1, x2 ):
  Real  o, p, q, r, s, x1, x2 ;    ...  Choose precision here.
  S := p*r ;  o := p*r - S ;       ...  Suits PowerPC well.
  s := √((q*q - S) - o) ;          ...  NaN if √( < 0 ).
  S := q + CopySign(s, q) ;        ...  Fortran's SIGN O.K.
  If S = 0 then { x1 := x2 := r/p }  else
     { x1 := r/S ;  x2 := S/p };   ...  NaNs if not real,
  Return;  End  PPCQdrtc.          ...  or else may abort.
```

Aside from running slightly slower, `Qtest(PPCQdrtc)` differs from `Qtest( Qdrtc )` only by getting 53 sig. bits instead of NaN on the PowerPC and RS/6000, which then win the prize for accuracy. Which of `Qtest( Qdrtc )` and `Qtest( PPCQdrtc )` assesses accuracy more fairly?

In general, insisting that a benchmark exist in only one version, and that it run successfully (no NaNs !) on every machine, may cripple speed or accuracy or robustness on computers with advantageous features others lack. Permitting variety may invalidate comparison. As it is now, `Qtest( Qdrtc )` tells us something I think worth knowing regardless of whether it is admitted to the ranks of industry-approved benchmarks.

Exceptions in General, Reconsidered:

The prevailing attitude towards exceptions is changing. Previously they were declared to be errors that would abort an offending program. Abortion could be prevented only by defensive programming that tested for every error condition in advance. Punitive policies and paranoid practices persist, but now in competition with other options afforded programmers by IEEE 754 though handicapped by their near invisibility in programming languages. How might exception-handling be practiced if other options were supported properly? The Standard Apple Numerical Environment (SANE), documented in the Apple Numerics Manual (1988), is one approach. What follows is another I have implemented partially.

First, exception-classes must have names, preferably the same names in all languages. Venerable languages that limit names' lengths still live, so the names have to be short; here are suggestions for five-letter names for floating-point exceptions plus a few others:

Name	Description of Exception
INXCT	INeXaCT due to floating-point roundoff or over/underflow
UNFLO	floating-point UNderFLOw, Gradual or not
DIVBZ	Infinity exactly from finite operand(s); e.g., 1/0
OVFLO	floating-point OVerFLOw
INTXR	INTeger arithmetic eXception or eRror like overflow or 1/0
INVLD	INVaLiD operation, most likely one from the list that follows
ZOVRZ	0.0 / 0.0 :
IOVRI	Infinity / Infinity : These four are the rational
IMINI	Infinity - Infinity : Removable Singularities.
ZTMSI	0.0 * Infinity :
FODOM	Function computed Outside its DOMain; e.g., √(-3)
DTSTR	Attempted access outside a DaTa STRucture or array
NLPTR	De-referencing a NiL PoinTeR
UNDTA	UNinitialized DaTum or vAriable, or SNaN

These names are intended to identify such flags as may exist to signal exceptions to a program, and such modes as a programmer may choose to predetermine the program's response to exceptions.

More important than the spellings are the length and structure of the list. It must be parsimonious; if allowed to grow indefinitely it can accrete names unknown to most of us or with overlapping meanings, and then our programs would mishandle their exceptions. The list should be comprehensive enough to leave no kind of exception uncovered by a name; the list above may be incomplete. It does include names for exceptions practically undetectable on some systems; examples are UNFLO on a CRAY Y-MP, IOVRI on machines that lack Infinity, DTSTR on systems that do not check array-bounds, and INXCT on non-conformers to IEEE 754. The list is structured less to reflect how or where the exception is detected (as C's nearly useless ERRNO does) and more to reflect what may be done to remedy it. For example, expressions 0/0, ∞/∞, ∞-∞ and ∞*0 are distinguished because, to produce anything of more use than a NaN, they require different versions of l'Hospital's rule for the removal of removable singularities.

Though different named exceptions require different remedies, the list of remedies worth considering for any particular named exception class fits into a short menu for a preprogrammed exception-handling library. Selection from an adequate menu will serve applications programmers far better than coding up each his own handler. Here are five-letter names for the few exception-handling modes thought worth putting into a menu:

PAUSE, ABORT, PREMT, IEEED, PSUBS, KOUNT, ABSNT

To select mode PAUSE for handling, say, OVFLO is to request that each floating-point overflow suspend computation and invoke a debugger to display the values of variables defined at that moment, after which the onlooker may either abort computation or else resume it as if the overflow had been handled in the previously prevailing mode. PAUSE is a debugging mode applicable to every other exception-handling mode.

ABORT is a mode now common for severe exceptions; it empties buffers, closes files and returns to the operating system. PREMT pre-empts the handling of a designated exception for whatever language ambience had been over-ridden by a previously invoked mode. For example, to PREMT ZOVRZ in APL is to re-establish the definition 0/0 = 1 ; to PREMT ZOVRZ in ADA is to put 0.0/0.0 back among Arithmetic Errors that drop execution into a program module's error handler, if one has been provided, or else ABORTs. PREMT is indispensable when a language sports control structures like

ON ERROR { do something else or go somewhere else } ;

and

ABSENT ERROR { try something } ELSE { do something else } ;

but lacks locutions to distinguish among different kinds of exceptions.

(Control structures like those palliate exceptions rather than handle them all well. The main deficiency is a lack of recognizable names for modes defined implicitly by { do something else } clauses; a name known to an independently compiled subprocedure of the {try something} clause could tell it which exceptions of its own to hide and which to expose. Other deficiencies exacerbate the cost of scoping: Which variables in the {try something} clause are to be saved, and in what state, for the {do something else} clause to use after the ERROR ? A satisfactory discussion lies beyond the scope of these notes.)

IEEED names the Default Mode in which every exception mentioned by IEEE 754/854 must be handled, by default, unless a programmer asks explicitly for another mode. (This requirement is violated by a few computer systems that run much faster in some other mode, and by some compilers whose authors fear the consequences of unleashing Infinity and NaN upon a programmer who has not said he wants them.) To every named floating-point exception except INXCT and UNFLO, IEEED has assigned what I call a " presubstitution "; that is a precomputed number whose magnitude and possibly its sign will be substituted for the result of an exceptional floating-point operation. For DIVBZ and OVFLO, IEEED presubstitutes $\pm\infty$ with an appropriate sign. For the INVLDs ZOVRZ, IOVRI, IMINI, ZTMSI and FODOM, IEEED presubstitutes NaN. (For INXCT, IEEED does not presubstitute but yields an approximation in accordance with current modes of rounding direction and precision; IEEED for UNFLO is Gradual.) For example, with DIVBZ in IEEED mode, $\mathrm{LOG}(0.0) = -\infty$ is not trapped as an ERROR though it does raise the DIVBZ flag.

PSUBS is a generalization of IEEED that presubstitutes any number, computed by the program in advance, for any floating-point exception. For example, PSUBS(0.0, ±) for UNFLO replaces Gradual Underflow by Flush-to-Zero with preservation of sign; invoke PSUBS(0.0) to flush UNFLO to +0.0 . Similarly, PSUBS(y) for ZOVRZ replaces a subsequent SIN(x*y)/x by y whenever x = 0.0 . Thus programmers can remove some removable singularities with PSUBS without explicit tests nor branches. It is no panacea. Those tests and branches may have to be introduced implicitly (by the compiler ?) for vectorized machines like CRAYs. Neither can PSUBS(COS(x)) for ZOVRZ shield (SIN(x) - SIN(y))/(x-y) from damage caused by roundoff when x is too near y ; use PSUBS(1.0) and COS((x+y)/2)*(SIN((x-y)/2) / ((x-y)/2)) respectively instead.

Here is a nontrivial application of presubstitution. It simplifies an economical way to compute the continued fraction $f(x)$, introduced above during the Digression on Division by Zero, and simultaneously its derivative $f'(x)$. They might be computed for many different values x if they serve in Newton's iteration $x \longrightarrow x - f(x)/f'(x)$ for solving $f(x) = 0$. While $f(x)$ is being computed from the recurrence $f_n := (x - a_n) - q_n/f_{n-1}$, starting with $f_1 := x-a_1$ and ending with $f(x) := f_n$, another recurrence computes its derivative. Because multiplication can go rather faster than division, two divisions have been replaced by one division and two multiplications in the recurrences, but at the cost of introducing auxiliary variables $r_j = 1/f_j$, $h_j = q_j \cdot r_{j-1}$ (so $f_j = (x-a_j) - h_j$ and $f'_{j-1} = -f_{j-1} \cdot h_j'/h_j$) . Recall that every $q_j > 0$; this implies that every $f'_j \ge 1$ and, absent over/underflow, that no $h_j = 0$ unless f_{j-1} and f'_{j-1} are both infinite. Overflow or $0\cdot\infty$ could interfere with the computation of f'_j if nothing else were done about them. What we shall do about them is precompute an array of quotients $P_j := q_{j+1}/q_j$ and presubstitute.

```
Sovflo := PSUBS( 1.0E150, ± )  for  OVFLO ;    ... Save prior presubstitution for Overflow.
Sztmsi := PSUBS( 0.0 )  for  ZTMSI ;    ... Save prior presubstitution for 0·∞ .
f := x-a1 ;    f' := 1 ;
for  j = 2, 3, ..., n  in turn,
     do {      r := 1/f ;
               h := qj·r ;
               f := (x-aj) - h  ;
               Sf' := f' ;
               f' := (h·r)·f' + 1 ;    ... Presubstitution replaces 0·∞ here.
               PSUBS( Pj·Sf')  for  ZTMSI  } enddo ;
f'(x) := f' ;       f(x) := f ;
PSUBS(Sovflo) for OVFLO ;  PSUBS(Sztmsi) for ZTMSI ;  ... Restore prior presubstitutions.
```

This program has no floating-point test-and-branch. Instead, the first PSUBS replaces an overflowed f' by a huge but finite ±1.0E150, and then the PSUBS inside the do-loop accomplishes the same effect as if it and the previous assignment were replaced by ...

```
if  ( f' is finite )  then   f' := ( h·r)·f' + 1 ;
                      else   f' := Pj-1·SSf' + 1  endif
SSf' := Sf' .
```

Mode KOUNT(k) exploits exponent-wrapping to count OVER/UNDERFLOWs in an integer variable k as if it were a leftward extension of the floating-point exponent field. We have seen one application above; it was the fast and accurate evaluation of expressions like Q described under UNDERFLOW. If implemented fast enough, this mode also speeds up the comparison of complex magnitudes $|x + \imath y| = \sqrt{(x^2 + y^2)}$ via the relationship

$$|x + \imath y| < |u + \imath v| \quad \text{if and only if} \quad (x-u)\cdot(x+u) < (v-y)\cdot(v+y) .$$

(To attenuate roundoff first swap so that $|x| \ge |y|$ and $|u| \ge |v|$.)

OVFLO and UNFLO flags do not get raised in KOUNT mode.

. . . .

For nearly three decades, no other floating-point exception-handling modes than PAUSE, ABORT, PREMT, IEEED, PSUBS and KOUNT have been found both worthwhile and compatible with concurrent execution of floating-point and integer operations on very fast processors. If not due to a lack of imagination, this state of affairs justifies efforts to promulgate a modest library of exception-handling modes rather than leave every programmer to his own devices. A few more floating-point modes require support on systems that conform to IEEE 754 :

Directed Roundings (DIRND): ToNEAR, ToZERO, ToPOSV, ToNEGV
Rounding Precisions (RNDPR): ToSNGL, ToDBLE, ToEXTD

Rounding Precision modes are pertinent only to hardware that evaluates every floating-point expression in the Double-Extended (REAL*10+) format. However, they do raise a question of general interest:

What if a program attempts to invoke a nonexistent or unsupported mode?

An error-message protesting the use of an undefined name, or else the response to C's `#ifdef` command for conditional compilation, would answer this question at compile-time. At run-time the answer to an environmental inquiry concerning an unsupported mode's status might best be ABSNT, defined herewith to be the name of no mode. ABSNT is the mode of INXCT, UNFLO and DIRND on a CRAY Y-MP, for example.

Flags and modes are variables of type Flag and Mode that may be sensed, saved and set by library programs. I prefer programs that are syntactically functions but actually swap values. For example, my function Fflag(OVFLO, NewFlag) returns the current value of the OVFLO flag and resets that flag to NewFlag. Fflag(OVFLO) merely returns the value without changing it. A flag's value resembles a pointer, in that it may be either Null or some non-Null value returned by Fflag, and also resembles a Boolean value insofar as Null behaves like False, and every other flag like True. Consequently a typical pattern of use for Fflag goes like this:

```
SavOV :=  Fflag( OVFLO, Null ) ;    ...  saves & clears OVFLO flag.
X :=  expression that may  Overflow  prematurely ;
If  Fflag( OVFLO, SavOV )  then     ...  having restored OVFLO flag
            X :=  alternative expression ;
```

At the end, premature OVFLOs have been hidden, and the OVFLO flag is raised just if it was raised before or if the alternative expression X overflowed.

Similarly, Fmode(DIRND, NewDir) swaps the DIRND mode for saving, sensing and restoring. For example, if function g(z) is contrived either to return a monotonic increasing function of its argument z and of its rounding errors (this can be tricky), or else to take proper account of Fmode(DIRND, …), then a few statements like

```
SavDir :=  Fmode( DIRND, ToNEGV ) ;    ... Rounding towards -∞
xlo    :=  g( zlo ) ;                  ...  yields upper bound.
Dummy  :=  Fmode( DIRND, ToPOSV ) ;    ... Rounding towards +∞
xhi    :=  g( zhi ) ;                  ...  yields lower bound.
Dummy  :=  Fmode( DIRND, SavDir ) ;    ... Restores prior rounding.
```

guarantee that xlo ≤ g(zlo) ≤ exact g(z) ≤ g(zhi) ≤ xhi despite roundoff. This is admittedly a cumbersome way to obtain what Interval Arithmetic would deliver easily if it received the support it deserves from popular programming languages.

Here follows a simple example of flags and modes working together. The Euclidean Length (Norm) of a Double-Precision vector x is Vnrm(x[1:L]) := $\sqrt{(}$ $x[1]^2 + x[2]^2 + x[3]^2 + ... + x[L]^2$) . This simple formula arises in so many matrix computations that every matrix package like LAPACK and MATLAB contains a subprogram devoted to it. It poses two technical challenges; how may we ...

1. avoid an incorrect result caused by premature Overflow of some $x[j]^2$ or Underflow of too many of them though the true value of Vnrm is unexceptional?
2. avoid excessive accumulation of roundoff when L is huge? (For example, consider the case when every x[j] for j > 1 is barely small enough that $x[j]^2 + x[1]^2$ rounds to $x[1]^2$; then Vnrm can come out too small by about L/4 units in its last place if the additions are performed left-to-right. L usually stays below a few hundred, but often runs into several thousands.)

These challenges are worth overcoming only if doing so does not slow computation of Vnrm too much compared with the obvious subprogram:

```
Double  Vnrm( Double x[1:L] ) ;
  s := 0.0 ;
  For  j := 1 to L  do   s :=  s + x[j]*x[j]  ;
  Return{ Vnrm := √s }.
```

On a 680x0-based Macintosh or PC with ix87, our problem has an easy solution provided the compiler supports IEEE 754 Double-Extended (REAL*10+): begin the obvious subprogram with the declaration

```
Double-Extended  s := 0 .
```

Then the sum-of-squares s will accumulate in a register with 3 more bits of exponent range and 11 more bits of precision then Vnrm and x[...]. Thus, with no loss of speed, Over/Underflow is precluded unless Vnrm must lie out of range, and roundoff is kept below 1 + L/8192 units in its last place. These are typical benefits of an Extended format. Moreover, this subprogram honors Directed Roundings and the KOUNT mode of Over/ Underflow.

In the absence of Extended, a craftier subprogram is needed to fiddle with flags and modes during the computation of Vnrm , and particularly to ensure that the last expression computed and Returned also raises and merges only those flags that deserve alteration, or else KOUNTs. If the program to do so presented on the next page appears too baroque, compare it with slower, less accurate and more elaborate subprograms that now infest portable libraries like LAPACK.

The SANE library provides two procedures ProcEntry and ProcExit to save and restore all flags and modes, and merge old flags with new, simultaneously. However SANE makes no provision for exceptions other than those mentioned by IEEE 754 nor for modes other than ABORT and IEEED nor for computer systems that do not conform to IEEE 754. My scheme lets a programmer utter, for example,

```
If ( Fmode(INXCT, IEEED) = ABSNT ) then  Dummy := Fflag(INXCT, True)
                                   else  Dummy := Fflag(INXCT, Null);
```

after which his program repeatedly preconditions data until a rounding error renders further repetition unnecessary, on machines that detect INXCT, but preconditions just once otherwise.

```
Double  Vnrm( Double x[1:L] ) ;
  OVm := Fmode(OVFLO, IEEED) ;  UNm := Fmode(UNFLO, IEEED) ;
  OVf := Fflag(OVFLO, Null) ;   UNf := Fflag(UNFLO, Null) ;  ...  swaps!
  b := 1 ;  d := 1 ;  ...  these will be scale factors, if needed.
  s := 0 ;  c := 0 ;  ...  c will compensate for additive roundoff.
  For  j := 1 to L  Do {
     r := x[j]*x[j] ;
     t := s ;  s := (r+c) + t ;  ...  Compensate for this roundoff:
     c := ((t-s) + r) + c } ;    ...  Heed parentheses!
  OVf := Fflag(OVFLO, OVf) ;
  If ( Fflag(UNFLO, Null) & (s < 0.5^969) )             ...  Constants
               Then {  b := 2.0^996 ;  d := 0.5^996 }     ...  suit only
     Else If ( OVf )                                    ...  IEEE 754
               Then {  b := 0.5^754 ;  d := 2.0^754 } ;  ...   Double.
  If ( b ≠ 1 )  Then {  ...  Redo accumulation with scaled  x[j]'s.
     s := 0 ;  c := 0 ;

     For  j := 1 to L  Do {
        t := b*x[j] ;  r := t*t ;
        t := s :  s := (r+c) + t ;
        c := ((t-s) +r) + c } } ;
  UNf := Fflag(UNFLO, UNf) ;
  OVm := Fmode(OVFLO, OVm) ;  UNm := Fmode(UNFLO, UNm) ;
  Return{ Vnrm := d*√s } .
```

===

I need three more library programs. Two of them are swapping functions. Fpsubs(ExceptionName, NewValue) supports presubstitution. Second, Kountf(k, Initk) designates k to be the integer variable into which OVER/UNDERFLOWs will be counted, reads out the current value of k , and then resets it to the value of Initk . Those two may be embedded in Fmode. The third program inserts or updates entries in the log of Retrospective Diagnostics or reads it out, but that is a story for another day.

. . . .

The foregoing schemes to handle floating-point exceptions can be called elaborate, complicated, cumbersome, ...; add your own pejoratives. I shall rejoice if somebody shows me a simpler way to accomplish all of what my proposal tries to do. Meanwhile, onlookers who need not know about all these complications can stay in their blissful state of ignorance because IEEE 754 was designed with their state of mind in mind.

IEEE 754 establishes *partially nested computational domains*. What this means is best illustrated by examples:

Rare individuals who intend to perform floating-point computations exactly, without roundoff, must pay close attention to the INXCT exception; the rest of us ignore it because we are willing to tolerate roundoff. Someone whose concept of Real Number excludes Infinity must watch out for DIVBZ ; those of us who ignore it accept ±∞ as the uttermost among the Reals. Quantities so small as 1.0E-307 lie beneath our notice most the time, so we ignore UNFLO ; and IEEE 754 specifies that Underflow be Gradual to reduce the risk of harm from what we disdain. A few people can ignore OVFLO in a situation where any sufficiently big number will do; this belief could be tested by recomputing with OVFLO in a few modes like PSUBS(1.0E32, ±), PSUBS(1.0E64, ±), PSUBS(1.0E128, ±), ... instead of IEEED = PSUBS(Infinity, ±) . No one can ignore INVLD unless someone else has used Fflag(INVLD, ...) or Fmode(INVLD, PSUBS) or IsNaN(...) to cope with that exception.

In short, most of us can ignore most exceptions most the time provided someone else has thought about them. That “ someone else ” needs our support lest we all be obliged to face some ugly problems unaided.

Ruminations on Programming Languages:

Mediaeval thinkers held to a superstition that Thought was impossible without Language; that is how "dumb" came to change its meaning from "speechless" to "stupid." With the advent of computers, "Thought" and "Language" have changed their meanings, and now there is some truth to the old superstition: In so far as programming languages constrain utterance, they also constrain what a programmer may contemplate productively unless disrupted by bitter experience or liberated by vigorous imagination. Considering how relatively few programmers grapple daily with floating-point arithmetic, and how few of those have time to contemplate unsupported features of IEEE 754, it comes as no surprise that computer linguists receive hardly any requests to support those features.

Most computer linguists find floating-point arithmetic too disruptive. Their predilection for " referential transparency," which means that a well-formed expression's meaning should not change from one context to another, runs counter to an imperative of approximate calculation:

> The precisions with which expressions are evaluated must depend upon context because the accuracy required of an approximation depends more upon the uses to which it will be put and upon the resources available to compute it than upon alleged precisions of constituent subexpressions.

Consequently, rules promulgated in 1963, inherited from Fortran IV, for evaluating mixed-precision expressions are not optimal and never were; those rules turn pernicious when applied to more than two precisions, especially when precisions can vary at run-time. See C. Farnum's 1988 paper for better ways to handle mixed precisions.

These pernicious rules are deeply imbedded in `C++` insofar as its operator overloading explicitly disallows the expected type of a result to influence the choice of an operator now selected by consulting only the types of its operands. This restriction precludes certain troublesome ambiguities, but it also precludes fully effective `C++` implementations of intensely desirable but context-dependent programming ambiances like ...

> Mixed Precisions arbitrarily Variable at run-time.
> Interval Arithmetic arbitrarily mixable with Non-Interval Arithmetic.
> Ostensibly Coordinate-Free expressions concerning Objects in Linear Spaces.

Mixed-precision expressions should ideally be evaluated at the widest of the precision of operands in the expression not segregated by explicit coercions. Non-interval expressions must be evaluated as if they were intervals when mixed with Interval Arithmetic expressions. In ostensibly coordinate-free Linear Algebra, expressions must be evaluated in some coordinate system determinable from context if it is determined at all. These context-dependent languages become burdensomely awkward to use when the programmer is obliged to utter explicitly those coercions and conversions that a compiler could almost always determine quickly from context.

Computer linguists also dislike functions with side-effects and functions affected by implicit variables not explicit in argument lists. But floating-point operations can raise IEEE 754 exception flags as side-effects, and operations are affected implicitly by exception-handling and rounding modes eligible at run-time according to IEEE 754. Alas, that standard omitted to bind flags and modes to locutions in standard programming languages, and this omission grants computer linguists a licence for inaction.

The side-effects and implicit variables in IEEE 754 admit tractable disciplines; they are not whimsical. Moreover other computational domains exist where context-dependence, side-effects and implicit variables are rampant. Examples of side-effects and implicit variables abound in operating systems, input/output and file-handling, real-time control systems, and synchronization of parallel computing.

In short, the features of IEEE 754 that computer linguists disdain raise issues that cannot be evaded by avoiding floating-point; they have to be addressed elsewhere anyway, and in forms more obnoxious than in IEEE 754. Programmers ambitious enough to try to apply those features but left to their own devices cannot transport their meager successes to different computer systems; their situation could worsen only if palliatives were incorporated into language standards, and there is some risk of that. Thoughtful action is needed now to avert an intensification of market fragmentation that retards development of robust numerical software and diminishes the market and its rewards for all of us.

Annotated Bibliography.

IEEE standards 754 and 854 for Floating-Point Arithmetic: for a readable account see the article by W. J. Cody et al. in the IEEE Magazine *MICRO*, Aug. 1984, pp. 84 - 100.

"What every computer scientist should know about floating-point arithmetic" D. Goldberg, pp. 5-48 in *ACM Computing Surveys* vol. 23 #1 (1991). Also his "Computer Arithmetic," appendix A in *Computer Architecture: A Quantitative Approach* J.L. Hennessey and D.A. Patterson (1990), Morgan Kaufmann, San Mateo CA. Surveys the basics.

"Compiler Support for Floating-Point Computation" Charles Farnum, pp. 701-9 in *Software Practices and Experience* vol. 18 no. 7 (1988). Describes, among other things, better ways than are now customary in Fortran and C to evaluate mixed-precision expressions.

Intel *Pentium Family User's Manual, Volume 3: Architecture and Programming Manual* (1994) Order no. 241430 Explains instruction set, control word, flags; gives examples. Its flaws are listed in *Pentium Processor Specifications Update* Order No. 242480-001 (Feb. 1995)

Programming the 80386, featuring 80386/387 John H. Crawford & Patrick P. Gelsinger (1987) Sybex, Alameda CA. Explains instruction set, control word, flags; gives examples.

The 8087 Primer John F. Palmer & Stephen P. Morse (1984) Wiley Press, New York NY. Mainly of historical interest now.

User's Manuals (instruction sets, control words, flags) for ...

MC 68881 and 68882 Floating-Point Coprocessors	MC68881UM/AD (1989)
MC 68040 Microprocessor	MC68040UM/AD (1993)
Motorola PowerPC 601 Microprocessor	MPC601UM/AD (1993)

Apple *Numerics Manual, Second Edition* (1988) Addison-Wesley, Reading, Mass. Covers Apple II and 680x0-based Macintosh floating-point; what a pity that nothing like it is promulgated for the ix87 ! For PowerPC-based Macs, see Apple Tech. Library *Inside Macintosh: PowerPC Numerics* (1994); for PowerPC generally, see a forthcoming document on *Foundation Services for the* CommonPoint *Application System* from Taligent which will support Floating-Point C Extensions proposed for a new C standard now being debated by ANSI X3-J11. Copies of draft proposals concerning floating point generally and complex floating-point arithmetic in particular can still be obtained through Jim_Thomas@Taligent.com .

Sun Microsystems *Numerical Computation Guide* Part no. 800-5277-10 (Rev. A, 22 Feb. 1991) Information analogous to these notes' but aimed at Sun's customers; describes variances among compilers and hardware, offers advice, explains crude retrospective diagnostics.

"Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit" W. Kahan; ch. 7 in *The State of the Art in Numerical Analysis* ed. by M. Powell and A. Iserles (1987) Oxford. Explains how proper respect for -0 eases implementation of conformal maps of slitted domains arising in studies of flows around obstacles.

"The Effects of Underflow on Numerical Computation" J.W. Demmel, pp. 887-919 in SIAM *Jl. Scientific & Statistical Computing* vol. 5 #4 (Dec. 1984). Explains gradual underflow's advantages.

"Faster Numerical Algorithms via Exception Handling" J.W. Demmel and X. Li, pp. 983-992 in IEEE *Trans. on Computers* vol. 43 #8 (Aug. 1994). Some computations can go rather faster if OVERFLOW is flagged than if it will be trapped.

"Database Relations with Null Values" C. Zaniolo, pp. 142-166 in *Jl. Computer and System Sciences* vol. 28 (1984). Tells how best to treat a NaN (he calls it "ni" for " no information ") when it turns up in a database.

Floating-Point Computation P.H. Sterbenz (1974) Prentice-Hall, N.J. Ch. 2 includes a brief description of my early (1960s) work with Gradual Underflow, Over/Underflow Counting, and Retrospective Diagnostics on the IBM 7094.

"Handling Floating-point Exceptions in Numeric Programs" J.R. Hauser *ACM Trans. on Prog. Lang. and Syst.* vol. 8 #2 (Mar. 1996). Surveys many language-related issues and contains a substantial bibliography.