Ruminations on the Design of Floating-Point Arithmetic

Prof. W. Kahan
Univ. of Calif. @ Berkeley

Abstract:

Floating-point can be understood better now than half a century ago. But the benefit of better understanding cannot reach computer programmers and users, most of them unwitting users of floating-point, unless floating-point providers (implementors of hardware and of programming languages) attend to vastly many details. These seem unfairly burdensome because nobody else cares about more than a few of them. Which few? Each detail matters in somebody's program upon which others (perhaps we) may depend; but who knows? Disregarding any one detail will undermine the coherence of the rest of them and thus harm the whole community of floating-point users. The details descend from specifications for floating-point hardware and for its use by programming languages most of which, like *Java*, still disregard inconvenient details although they are implied by a few accidental constraints (like bus-widths) and by a short list of guiding principles paramount among which is *Intellectual Economy*. Alas, this crucial principle too much resembles pornography:

I don't know what it is, but I hope to recognize it if I see it.

What about computers matters most? Speed?

About every 18 months it doubles.

What matters more than speed is
Throughput : Tasks/sec, Transactions/sec, ...
which is not quite the same as speed.

Unreliability degrades Throughput :

$$\text{Malfunctions per month} = \frac{\text{Speed, in Operations per month}}{\text{No. of Operations between Malfunctions}}$$

How many malfunctions per month can you tolerate?

Hardware is getting more reliable as well as faster.
Software is not, so it imposes its own
Law of Diminishing Returns upon hardware speed.

Who notices this?

Whether it be noticed varies a lot with the application, the personality, and the ability correctly to locate blame for a malfunction:
"The fault, dear Brutus, is not in our stars,
but in ourselves, ..." (*Julius Caesar* I.i.139)
or is it due to some anonymous programmer?

Witness: expensive wreckage on Mars.

Why is software so unreliable? Reliability costs too much.

Why does reliability cost so much, often more than it would be worth ?

Establishing software reliability combines the methods of mathematical and experimental sciences in a way antithetical to their natures:
Mathematics and Science flourish around precious truths
eternal or at least fairly durable, as software is not.

Too few people master the analytical skills needed to achieve reliability.
A minority among holders of Computer Science degrees are actually Scientists.
Quantity must make up for a lack of Quality; therefore
many a software house employs rather more people
testing software than creating it.

Business: Reliability is a cost-, not a revenue-center.
Engineering: Reliability is a matter of degree, subject to trade-offs;
utter reliability is solely Death's prerogative.

If we yearn to improve software's reliability,
we must (re)design computer hardware and software,
programming languages most of all,
to cut the price paid for reliability
without significantly degrading functionality,
especially speed.

What remains to be seen is how this desideratum -
– reduce reliability's price without harming performance -
translates into strictures upon floating-point providers.

Details.

What undermines the reliability of floating-point?

It is too much like everything else.

Advertisements rarely mean just what they say;
they speak *Puffery* to wishful thinking, so …

- What you see is not what you get.
- What you get is not what you want.
- What you want is not what you need.
- What you need is no longer available.

Similarly, floating-point programs rarely mean just what they say; and programmers fail frequently to say just what they mean, as if wishful thinking suffused floating-point computation:

- What you see can't be what you get
  (because of binary-decimal conversion).
- What you get can't be what you request
  (because of roundoff and over/underflow).
- What you request can't be what you desire
  (but at best an approximation to it).
- What you desire can't be known
  (because of uncertain data and mathematical models).

Why should providers of floating-point in hardware and languages be held to a higher standard?

Why should providers of floating-point in hardware and languages be held to a higher standard?

Here is why:

A clear mind is needed to cope well with uncertainty,
be it in Nature or in the nature of floating-point computation.
Fuzzy thinking makes matters worse.

Uncertainty is intrinsic in floating-point computation because it approximates computation with Real numbers which are idealizations of arbitrarily refineable approximations

We may say that a (computed) number is "Uncertain" or "In Error", but that is just a manner of speaking.
Real numbers are not uncertain. We are uncertain about them.

Example: given two representations of Real numbers, say

$$1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + \ldots \quad \text{and} \quad \int_1^2 dx/x ,$$

we cannot decide whether they represent the same number without proving a theorem, for which no routine way exists.

Every Real number has infinitely many representations.
Every such representation represents just one Real number.

Many people think otherwise, claiming to be able to distinguish "3" from "3.0" from "3.0000000000000000".

They are mistaken. *Their fuzzy thinking makes matters worse.*
For example ...

"25.4" appears as a literal constant in a program.
What is its nationalty?

And what is the color of the bear?

" 25.4 " appears as a literal constant in a program.
What is its nationalty?

Table 1: How many Millimeters in an Inch?

Inch nationality	Era	Millimeters
British Imperial	Early 1800s	25.39954...
	Late 1800s	25.39997...
U.S.A.	Early 1900s	25.40005...
	Late 1900s	25.4 exactly

Regardless of the program's nationality, compilation may well convert its " 25.4 " from decimal to binary and round it to a slightly different number. Which? That varies with the language and its compilers, all on the same computer:

As a 4-Byte float 25.4E0 = 25.399999619... ,
As an 8-Byte double 25.4D0 = 25.39999999999999858... ,
As a 10-Byte long-double 25.4T0 = 25.3999999999999999653... .

Considering historical variations, such discrepancies seem too tiny for journalists to notice compared with mi. *vs.* km. and knots *vs.* kmph.

Tiny discrepancies may go unnoticed by the program's users too ...
Geodetic surveys under diverse colonial administrations and U.N. ...
Different units of length must be taken into account by the program ...
Predictions of landslides, **earthquakes**, volcanic activities ...
A small correlation appears between geodetic markers' speeds and the administrations under which they moved. Bad things ensue.

Blame the programmer, provided she can be found. "Experts" testify "25.4" should have been written "25.400000000000" or "25.4D0" .

Do you agree ?

Blame the programmer, provided she can be found. "Experts" testify "25.4" should have been written "25.400000000000" or "25.4D0".

But, "corrected" by having "D0" appended to its noninteger decimal literals, the program fails badly and mysteriously on rare occasions.

Failures arise from the appearance of " 2.54 " in the program, but in a part written by someone else who worked with cm. instead of mm.

25.4 = 10 · 2.54 exactly, and
25.4E0 = 10 · 2.54E0 *exactly*, and
25.4T0 = 10 · 2.54T0 *exactly* ; but
25.4D0 ≠ 10 · 2.54D0 quite exactly because ...

25.4D0 – 25.4 ≈ $-1.42_{10}-15$ but
2.54D0 – 2.54 ≈ $+3.55_{10}-17$. Not exactly a factor of 10 .

These coincidences collide with another: Distances measured in inches to the nearest 1/32" are representable exactly in binary if not too big. If d is such a distance, 10·d is representable exactly in binary too. ...

25.4·d and 2.54·(10·d) are exactly the same.
(25.4E0)·d and (2.54E0)·(10·d) round to the same value ,
(25.4T0)·d and (2.54T0)·(10·d) round to the same value. But
(25.4D0)·d and (2.54D0)·(10·d) round differently occasionally.
For example, after they are rounded to 53 sig. bits the values of

(25.4D0)·(3.0) and (2.54D0)·(30.0)

differ by an ULP (Unit in the Last Place stored, $1/2^{46} \approx 1.42_{10}-14$).

The difference between (25.4D0)·**d** and (2.54D0)·(10·**d**) suffices to put these values on different sides of a threshold, leading to different and inconsistent branches in subprograms that treat the same datum **d** .

The difference between (25.4D0)·**d** and (2.54D0)·(10·**d**) suffices to put these values on different sides of a threshold, leading to different and inconsistent branches in subprograms that treat the same datum **d** .

"Corrected" with constant literals "25.4D0" and "2.54D0" , the program would go berserk over a sprinkling of otherwise innocuous data **d** that the "incorrect" program with literals "25.4" and "2.54" had handled perfectly, almost.

Could you have debugged the "Corrected" program?

Protest! " To think binary approximations of 25.4 and 2.54 can always be one exactly 10 times the other is overly naivé."

This protest is mistaken. Here is how to make 10 happen:

```
c0  := 2.54  rounded to the precision intended for all variables;
f0  := 4·c0 ;          ... exact in Binary floating-point.
f1  := f0 + c0 ;       ... rounds to very nearly 5·c0 .
c1  := f1 – f0 ;       ... exact in any decent floating-point.
c10 := 10·c1 ;         ... exact in Binary or Decimal, not Hex.
```

Now, unless the compiler has "optimized" f0 and f1 away, c10 and c1 turn out to be almost as good binary approximations of 25.4 and 2.54 as are to be had, and one exactly 10 times the other.

Programmers aware of the importance of such a relationship and the need for its defence can enforce it.

The program to predict earthquakes is a fictional didactically motivated over-simplified composite of the most common and most commonly misdiagnosed bugs in programs that use a little floating-point.

2.54 25.4 2.54D0 25.4D0 c1 c10

What is worth remembering about these numbers?

1: Floating-point software depends upon the preservation of certain
Mathematical Relationships,
much as software generally depends upon loop-invariants etc., except floating-point cannot preserve *all* Mathematical Relationships *exactly*.

We depend upon Precision and Accuracy to preserve all relationships as well as possible, trying to preserve *all* because the relationships that don't matter so much are almost never known in advance.

Precision ... statement of desire or intent. Accuracy ... accomplishment.

Sometimes Precision and Accuracy can't do enough, and then special steps must be taken to preserve vital relationships like
Symmetry, Monotonicity, Correlation, 10, ...
These steps often appear like silly pet tricks ...
redundant variables, extra parentheses, computations of zero, ...
to compiler writers, who then too often "Optimize" them away.

2: Error-analysts know Binary floating-point is best for error-analysis.
But Decimal floating-point is best for everyone else.
Only with Decimal floating-point can what you see be what you get.

Binary vs. Decimal almost doesn't matter for Integers because they are not rounded off.

But Binary will not soon go away,
and trying to hide it makes matters worse. ...

Trying to hide the Binary behind the Decimal makes matters worse....

Case Study:

Extracted from "Bug Watch", p. 30 of *PC World*, 20 May 1993:

" . . . . . .

QPRO 4.0 & QPRO for Windows
(Borland's *Quattro Pro* Spreadsheets)

Users report and Borland confirms a bug in @ROUND ; it may round decimal numbers ending with the digit 5 inconsistently. For example,

@ROUND(31.875, 2) should round to two decimal places;
it should always yield 31.88 ,
but it sometimes yields 31.87 .

PC World's tests confirm this bug.

Borland recommends using the @INT function instead as follows:

To round the number in cell A1 to two decimal places, enter
@INT((A1 + 0.005)*100)/100 .

. . . . . ."

This cure is worse than the disease.

@ROUND actually works correctly for numbers near		31.875 ,
rounding	numbers slightly less to	31.87 ,
	numbers equal or slightly bigger to	31.88 .

@INT((A1 + 0.005)*100)/100 works incorrectly, yielding 31.88 for numbers A1 equal to or slightly bigger than 31.8749999999999822... .

Trouble arises partly because Quattro displays at most 15 sig. dec.; numbers from 31.87499999999947... to 31.875000000000046... all display as " 31.875000...000 ". **What you see is not what you get.**

More trouble arises because of "Cosmetic Rounding" designed to hide the trouble misdiagnosed by Borland, Quattro's users, and *PC World*.

More trouble arises because of "Cosmetic Rounding" designed to hide the trouble misdiagnosed by Borland, Quattro's users, and *PC World.*

Quattro's pious fraud:

Ideally @INT(x) = the integer nearest x and no bigger in magnitude.
We expect @INT(1.00...001) = 1 and @INT(0.999...999) = 0 .
Whenever 0 < x < 2 we expect @INT(x) = (x >= 1) .
But ...

Range of Stored Values x	Displayed x	@INT(x)	(x >= 1)
1 – 14/2^53 to 1 – 6/2^53	0.999999999999999	0	0
1 – 5/2^53	1.000000000000000	0	0
1 – 4/2^53 to 1 – 1/2^53	1.000000000000000	1	0
1 to 1 + 21.2^52	1.000000000000000	1	1

The discrepancy in the second-last line exposes a serious bug.

Why should roundoff contaminate the arithmetic expression @INT(x) or the logical expression (x >= 1) ? How would you debug this?

Apparently @INT(x) first rounds its argument x to 53 sig. bits, multiplies it by $1 + 2^{-51}$ and then rounds the product to 53 sig. bits before discarding its fractional part. Why multiply by $1 + 2^{-51}$?

Without it, for many an argument x displayed as an integer N though it is slightly smaller, @INT(x) would yield not N but N–1 . But the fudge factor fails to prevent this, and deepens @INT's mystery.

No mention of roundoff nor of Binary floating-point appears in almost 1200 pages of documentation that came with Quattro Pro 4.0 .

What is worth remembering about Quattro's @INT ?

1: Binary floating-point is a nuisance to people who think Decimal.

2: Avoid cosmetic rounding; it merely obscures what it tries to hide.

Trouble is caused not by displaying no more than a user wishes to see, but by failing to display faithfully as much as a user asks to see.

Even if what you see is not what you get, you will wish occasionally to see enough of it to distinguish it from everything else and to reproduce it exactly.

To distinguish one 8-Byte Double from its neighbors can require as many as 17 sig. dec., not just the 15 which is Quattro's maximum. Fewer than 17 are too few to distinguish adjacent Doubles like

$$1024 - 2^{-43} = 1023.9999999999998863\ldots \quad \text{and}$$
$$1024 - 2^{-42} = 1023.9999999999997726\ldots .$$

Displaying all 17 sig. dec. has its ugly aspects:

Enter " 0.8 ".	See " 0.8000000000000004 "	displayed.
Compute 32200/32.2 .	See " 999.99999999999989 "	displayed.

This is part of a user's education, not to be denied him.

Where does that number " 17 " come from?

Span and Precision of IEEE 754 Floating-Point Formats :

Format	Min. Subnormal	Min. Normal	Max. Finite	Roundoff 2^{-N}	Sig. Dec.
4- Byte Single:	1.4 E-45	1.2 E-38	3.4 E 38	6.0 E-8	6 — 9
8-Byte Double:	4.9 D-324	2.2 D-308	1.8 D 308	1.1 D-16	15 — 17
≥10-Byte Extended:	≤ 3.6 T-4951	≤ 3.4 T-4932	≥ 1.2 T 4932	≤ 5.4 T-20	≥ 18 — 21
(16-Byte Quadruple:	6.5 Q-4966	3.4 Q-4932	1.2 Q 4932	9.6 Q-35	33 — 36)
Soft. Doubled-Double		2.2 D-308	1.8 D 308	about 1D-32	about 32

16 Byte Quadruple is a *de facto* standard, not official, and not yet in hardware.
16 Byte Doubled-Double is not standardized at all, and my be rounded roughly.

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

Rounded result of one operation = (1 ± ß)·(Ideal result)

Lesser Sig. Dec. survive Dec. —> Bin. —> Dec. Conversion
Greater Sig. Dec. survive Bin. —> Dec. —> Bin. Conversion

Each IEEE 754 Binary Floating-Point Standard format can representat ±∞ (Infinity), NaNs (Not-a-Number), and its own set of finite real numbers each of the simple form $2^{k+1-N}\,n$ with two integers n (its signed *Significand*) and k (its unbiased signed *Exponent*) that run throughout two unbroken intervals determined from the format thus:
K+1 Exponent bits: $1 - 2^K < k < 2^K$. N Sig. bits: $-2^N < n < 2^N$.

Formats' Parameters:

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

IEEE 754's simple representation for *all* its finite numbers, namely

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

with its signed *Significand* n and its
unbiased signed *Exponent* k
that run throughout two unbroken intervals

K+1 Exponent bits: $1 - 2^K < k < 2^K$,

N "Significant" bits: $-2^N < n < 2^N$,

and the standard's prescription for careful rounding "to nearest"

avoid perplexing anomalies afflicting earlier commercially important computer arithmetics some of whose numbers could be ...

- Equal, but their ratio is 1.25
- Different, but their difference is 0.0
- Different and both huge, but their difference is 0.0
- Different, with a difference X–Y that is NOT the negative of Y–X
- Changed if multiplied by 1.0D0
- Nonzero for add, subtract and compare,
  but 0.0 for multiply and *divide*
- Added, and multiplied by 0.97 , say,
  but not by 1.0 without *Overflowing* .
- Product $0.5 \cdot X$ could differ from quotient X/2.0 .
- Small difference X – 1.0 could differ from (X – 0.5) – 0.5 .

These bizarre departures from expected mathematical relationships for rounded arithmetic used to thwart program portability and debugging.

How could vendors excuse these anomalies?

They invoked "Backward Error-Analysis" to excuse most of them.

Backward Error-Analysis

Initiated by A. Turing (1948-9), W. Givens (1954), F.L. Bauer (1957) and J.H. Wilkinson (1957), joined by W. Kahan (1958).

The idea: Compare

Roundoff's contribution to a slightly wrongly computated result

with the

Effect upon desired result of end-figure perturbations in given data.

i.e.,
- We desire $f(x)$; we compute $F(X)$.
- If backward error-analysis succeeds, we find that $f(X) - F(X)$ is scarcely worse than $f(X) - f(X + \Delta X)$ for some unknown ΔX comparable with $X-x$ for all we know.
- Then the program $F(\ldots)$ is deemed "Numerically Stable" even if $F(X)$ is utterly different from $f(X)$, for which we blame the "Ill-Condition" of $f(\ldots)$ at X .
- Backward error-analysis often succeeded.

This approach to errors led to an efflorescence of numerically stable software supplanting centuries' accumulation of numerical methods whose misbehavior could now be corrected or at least explained.

What began as explanation was turned into exculpation in the hands of hardware and software vendors who alleged ...

" Not knowing x you will accept $f(X)$ for some stored X indistinguishable from x for practical purposes, and so you shall accept also $F(X) = f(X + \Delta X)$ for any $X+\Delta X$ about as indistinguishable from the unknown x as X is."

The vendors failed to see the flaw in their reasoning.

Almost all their customers failed to see the flaw too.

Can you see the flaw?

" Not knowing x you will accept $f(X)$ for some stored X
indistinguishable from x for practical purposes, and so you shall
accept also $F(X) = f(X + \Delta X)$ for any $X+\Delta X$ about as
indistinguishable from the unknown x as X is."

Where is the flaw in this reasoning?

There are two logical flaws.
The first arises from the *Intransitivity of Accuracy*,
a failure that sets floating-point software apart from all others:

Suppose progams $R(x) \approx r(x)$ and $Q(y) \approx q(y)$ each approximates its target function as well as possible, perhaps "correctly rounded". Still, it is possible for $P(x) := Q(R(x))$ to approximate $p(x) := q(r(x))$ arbitrarily badly. Accuracy can be lost by functional composition, thus thwarting a decomposition technique nonnumerical programmers enjoy.

Example: Cf. x vs. $(-\log(\exp(-x^{-4})))^{-1/4}$ for $4000 < x < 12000$.
This example is explored on my web page in the note "Matlab's Loss is Nobody's Gain", together with a more perplexing example in which P is accurate although Q and R are not!

Cases where accuracy would survive composition if R computed r accurately are precious. Degrading R as badly as that flawed appeal to Backward Error Analysis allows destroys those cases' accuracy: $q(r(x + \Delta x))$ may now approximate $p(x)$ poorly where $Q(R(x))$ could reasonably have been expected to approximate $p(x)$ well.
e.g.: $\log(\arccos(x))$ for $|x|$ barely smaller than 1

The second flaw arises from a common practice: computing several functions $f(x)$, $g(x)$, $h(x)$, ... from the same datum x . If computed accurately, these functions can be related in ways that matter later. But if $F(x) \approx f(x + \Delta_1 x)$, $G(x) \approx g(x + \Delta_2 x)$, $H(x) \approx h(x + \Delta_3 x)$, ... for tiny but uncorrelated perturbations $\Delta_k x$, then the computed functions F, G, H, ... can violate utterly the relations honored by f, g, h, ...

"ALRIGHT! SO SOME PROGRAMS WON'T WORK AT FIRST. WE'LL DEBUG THEM AND FIND ANOTHER WAY. THAT'S WHY WE ARE PAID THE BIG BUCKS." So say many experts.

Not true. If you share that wrong idea of Backward Error Analysis, there are programs you can't debug since you won't write them.

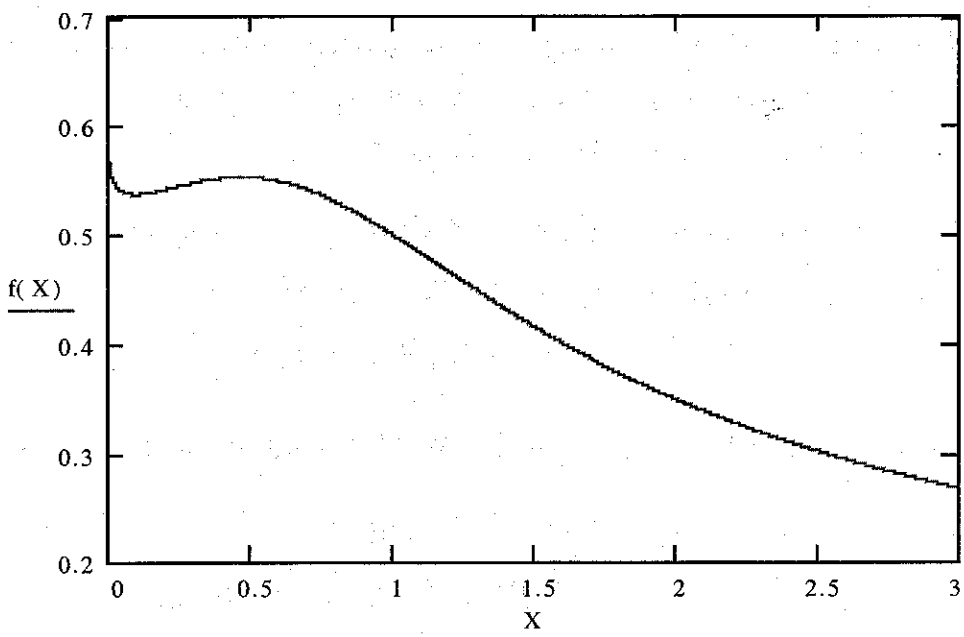
Example:

$$f(x) := \begin{cases} -\arctan(\log(x))/\arccos(x)^2 & \text{for } 0 < x < 1\,, \\ 1/2 & \text{for } x = 1\,, \\ \arctan(\log(x))/\operatorname{arccosh}(x)^2 & \text{for } x > 1\,. \end{cases}$$

This function $f(x)$ is smooth around $x = 1$ where its Taylor series is

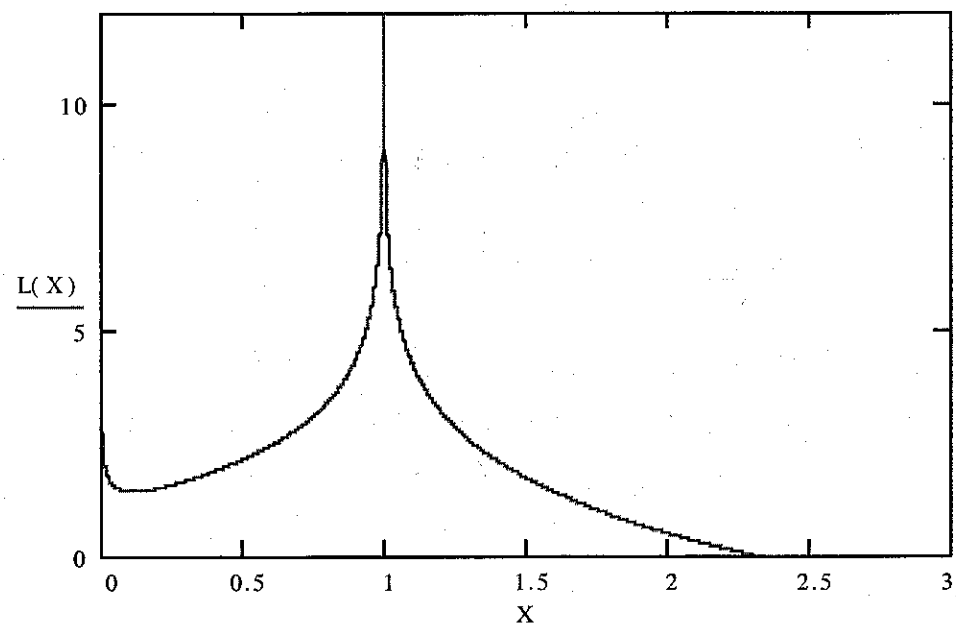
$$Tf(x) = 1/2 - (x-1)/6 + (x-1)^2/20 + 124(x-1)^3/945 + \ldots$$

Figure *f* : $f(x)$ for $0 < x < 3$



If LOG and ACOS are as bad as flawed Backward Error Analysis allows, then $\mathrm{LOG}(x) \approx \log(x + \Delta_1 x)$ and $\mathrm{ACOS}(x) \approx \arccos(x + \Delta_2 x)$ for independent end-figure perturbations $\Delta_k x$. Now an application of the Calculus reveals for x near 1 that all sig. bits can be lost from $-\arctan(\log(x))/\arccos(x)^2$. Similarly from $\arctan(\log(x))/\mathrm{arccosh}(x)^2$.

Figure L: L(x) = #(sig. bits lost to independently perturbed arguments)



To avoid this loss, F(x) is programmed using a small threshold h determined somehow from the accuracy desired and the number of terms retained in the Taylor series Tf ; say ...

$$
\begin{aligned}
F(x) := \; & -\mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOS}(x)^2 && \text{for } 0 < x < 1 - h\,, \\
& \mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOSH}(x)^2 && \text{for } x > 1 + h\,, \\
& 0.5 - (x-1)/6 + (x-1)^2/20 + 124(x-1)^3/945 && \text{otherwise.}
\end{aligned}
$$

For no choice of h etc. is the program above so accurate as the obvious

$$
\begin{aligned}
F(x) := \; & -\mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOS}(x)^2 && \text{for } 0 < x < 1\,, \\
& \mathrm{ATAN}(\mathrm{LOG}(x))/\mathrm{ACOSH}(x)^2 && \text{for } x > 1\,, \\
& 0.5 && \text{otherwise.}
\end{aligned}
$$

on every commercially significant computer (CRAYs too) today.

What is worth remembering from the foregoing examples?

1: ERROR-ANALYSIS IS TRICKY.
2: BACKWARD ERROR ANALYSIS IS TRICKIER.

It is mistaken to treat numbers as uncertain when in fact the uncertainty is in our minds, not in the numbers Those stored in our computers may differ from the numbers we desire, but the ones in the computer are the only ones we know and must be treated for what they are, exactly, or else we shall be unable to replace them by numbers we prefer.

3: Most programmers will decline to perform any error-analysis.

The numerical stability of many programs is disgustingly obvious; these need not concern us, fortunately.

The numerical instability of many programs is so delightfully obvious that they are used at most a few times; these need not concern us.

Unreliability arises from programs whose numerical instability is unobvious and infrequent, and usually very hard to diagnose.

Numerical computation has always been this way. Here is an example:

EDSAC's old arccos program

EDSAC at Cambridge, England, was one of the earliest electronic computers, built out of vacuum tubes and delay lines.

Case Study: EDSAC's original arccos:

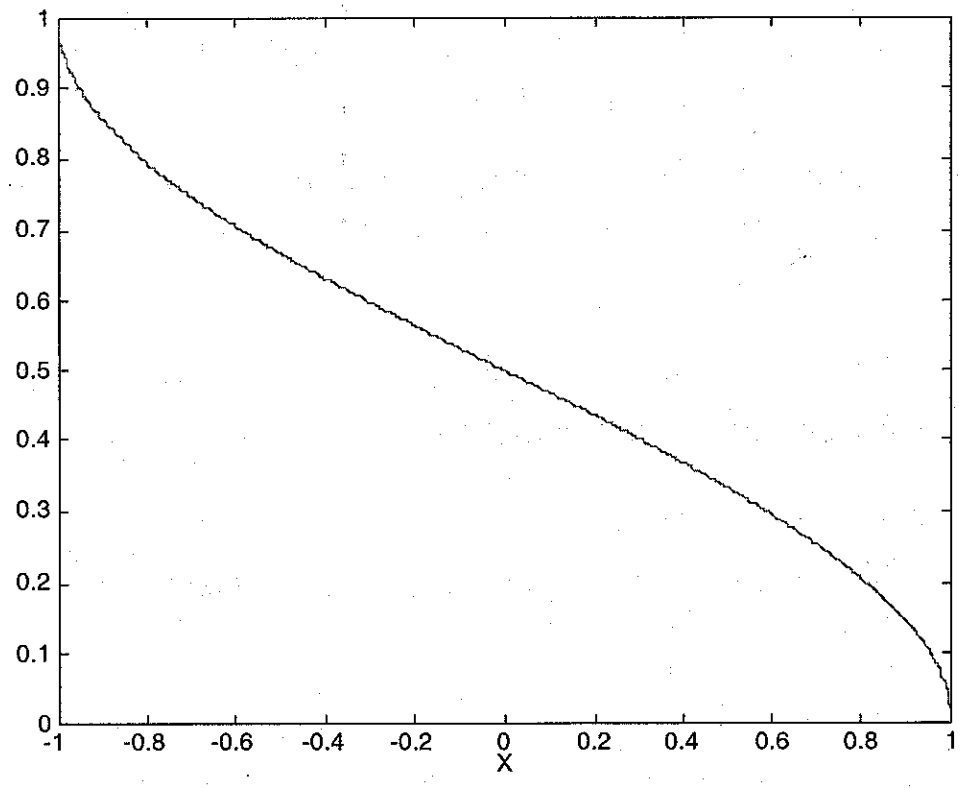
This is the earliest instance I could find of an electronic computer program in full service for over a year before users noticed its "treacherous nature". Its errors were of the worst kind:

- Too small to be obvious but
  too big to be tolerable
  (half the figures carried could be lost),
- Too rare to be discovered by the customary desultory testing, but
  too nearly certain to afflict unwitting users at least weekly.

The program served users of EDSAC at Cambridge, England, from 1949 to 1951 until A. van Wijngaarden exposed its "treachery".

The program tried to compute $B(x) := \arccos(x)/\pi$ for $-1 \le x < 1$.

Figure B: $B(x) = \arccos(x)/\pi$ changes fast as x varies near ± 1 .

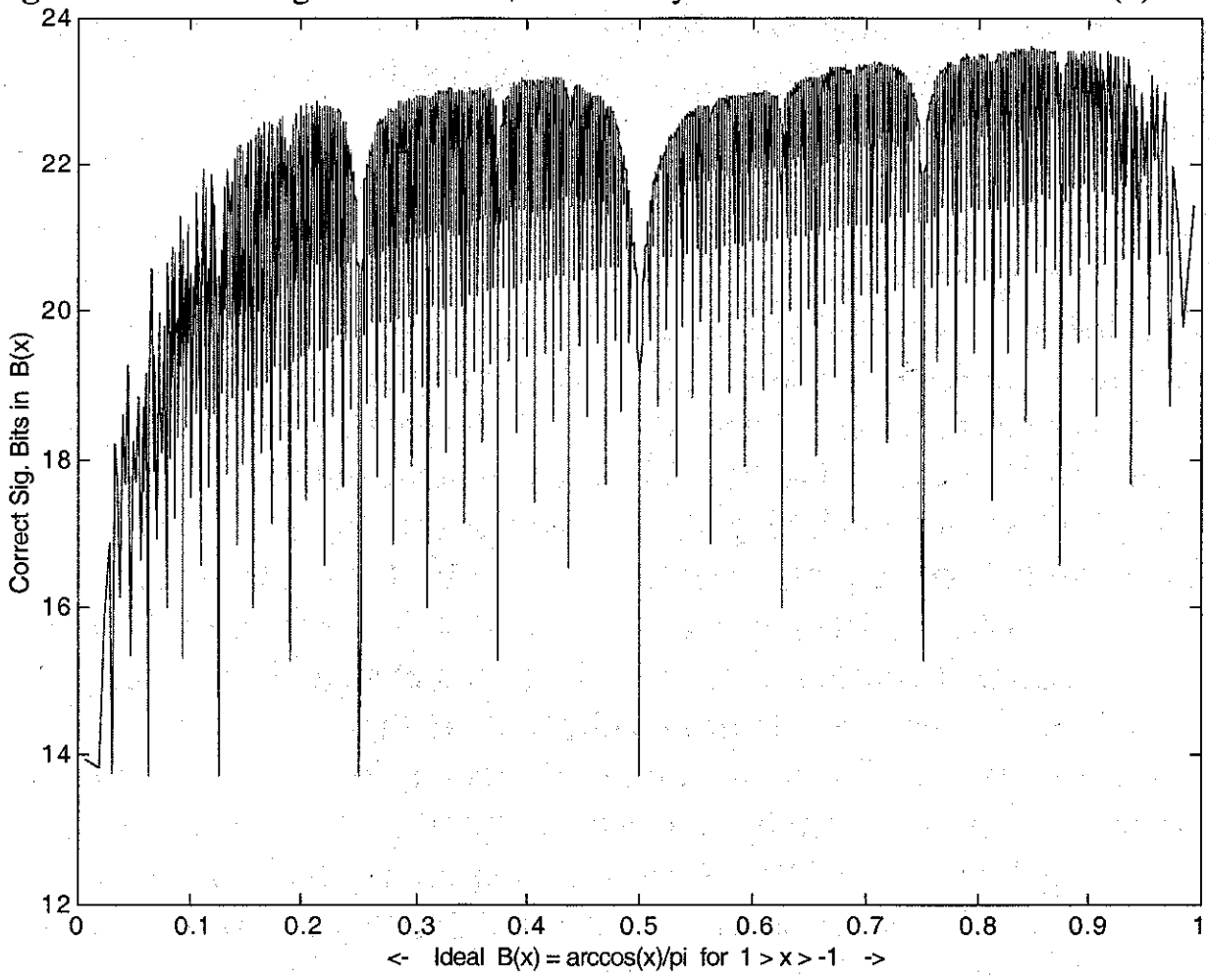


Here is the simple program for $B(x) := B$, somewhat edited:

Set $x_1 := x = \cos(B\pi)$; $ß_0 := 0$; $B_0 := 0$; $t_0 := 1$;
While $(B_{j-1} + t_{j-1} > B_{j-1})$ do (for $j := 1, 2, 3, \ldots$ in turn)
{ $t_j := t_{j-1}/2$; $\ldots = 1/2^j$.
$\mu_j := \mathrm{SignBit}(x_j)$; $\ldots = 0$ or 1 according as $x_j \geq 0$ or not.
$ß_j := |\mu_j - ß_{j-1}|$; $\ldots = 0$ or 1 according as $\mu_j = ß_{j-1}$ or not.
$B_j := B_{j-1} + ß_j \cdot t_j$; $\ldots = \sum_{1 \leq k \leq j} ß_k/2^k < 1$, a binary expansion
$x_{j+1} := 2 \cdot x_j^2 - 1$ } . $\ldots = \cos(2^j \cdot \arccos(x)) = \cos(2^j \cdot B\pi)$.

No subscript appears in the actual program, nor does j . The equation $x_{j+1} = \cos(2^j \cdot B\pi)$ follows by induction from $\cos(2\Theta) = 2 \cdot \cos^2(\Theta) - 1$.

Figure C: Of 24 Sig. Bits Carried, how many are Correct in EDSAC's B(x) ?



Rounding errors occur only in the last statement $x := 2 \cdot x^2 - 1$, with astonishing consequences on rare occasions. Figure C plots the worst errors for each of 2048 batches of a million arguments, all the 4-Byte `floats` x between -1 and $+1$, showing a narrow downward spike in accuracy wherever $\arccos(x)/\pi$ is very nearly (but not exactly) a small odd integer multiple of a power of 1/2 . The bigger that integer, the narrower and shallower the spike. Evidently up to almost half the sig. bits carried by the arithmetic can be lost in the computed result B(x) for such arguments x which, alas, arise frequently in practice.

First explanation, presented in 1951 by A. van Wijngaarden (1953), included an estimate under 1% of the probability that a random test might expose the loss of more than 3 or 4 sig. bits. According to Wilkes (1971), testers performed about 100 more-or-less random tests, laboriously comparing EDSAC's values with published tables; their probability of overlooking the program's faults exceeded 1/3 . They were slightly unlucky.

People have argued that the error in the program's B(x) is barely worse than uncertainty inherited from the argument x . It seems reasonable to think that if x is not worth distinguishing from $x + \Delta x$ then B(x) is not worth distinguishing from $B(x+\Delta x)$ nor from a computed value nearly as close to B(x) . Since x too is a computed value unlikely to be accurate down to its last bit, they argue that we ought to tolerate errors in B(x) not much bigger than variations in $B(x + \Delta x)$ arising from perturbations Δx of the order of an ULP (Unit in the Last Place) of x . For instance, B(x) is nearly 0 when x is very nearly 1 ; then tiny end-figure alterations of order ε in x can change B(x) utterly by as much as $\sqrt{(2\varepsilon)}/\pi$, orders of magnitude more than the alteration to x ; see Fig. B. End-figure alterations of order ε in x near -1 can change B(x) near 1 also by as much as $\sqrt{(2\varepsilon)}/\pi$, thus losing almost half the sig. bits carried. Based upon this reasoning, the program for B(x) is deemed about as accurate as the function B(x) deserves, or so Morrison (1956, p. 206) seems to imply.

The foregoing plea for *Tolerance* is plausible but

"irrelevant, incompetent and immaterial." ... Perry Mason

We have seen why such pleas are misguided. Irrelevance becomes clear when x is near $1/\sqrt{2} = 0.7071\ldots$ and B(x) is near 0.25 ; then tiny changes in x induce similar tiny changes in B(x) , and yet EDSAC's program can lose almost half the sig. bits carried. This loss must be blamed not upon the function B(x) but upon the program, which dislikes certain innocuous arguments x like $1/\sqrt{2}$.

EDSAC's simple bit-by-bit algorithm is a precursor of CORDIC and "pseudo-multiply/divide" versions used with processor chips too old, too cheap, too tiny or too sparing of power to afford a big multiplier array. Versions on the Intel 8087/80387/486DX are numerically stable.

.....

Citations:

D.R. Morrison (1956) "A Method for Computing Certain Inverse Functions" pp. 202-8 of *MTAC* (*Math. Tables and Aids to Computation*) vol. **X**; also (1957) correction on p. 314 of **XI**; and on p. 204 of **XI** a warning "Note ..." by Wilkes and Wheeler who, however, seem to have overlooked Morrison's explicit error-bound at the top of his p. 206, perhaps because he seems to have been indifferent to the loss of half the figures he carried.

Adrian van Wijngaarden (1953) "Erreurs d'arrondiment dans les calculs systématiques" pp. 285-293 of *XXXVII: Les machines à calculer et la penseé humaine*, proceedings of an international conference held in Paris, 8-13 Jan. 1951, organized by the Centre National de la Recherche Scientifique. This is among the earliest published error-analyses of computer programs, and one of the first to mention floating-point if only in passing. At that time, floating-point error-analysis was widely deemed intractable.

Maurice V. Wilkes (1971) "The Changing Computer Scene 1947 - 1957" pp. 8.1-5 of *MC-25 Informatica Symposium*, Mathematical Centre Tract #37, Mathematisch Centrum Amsterdam. This symposium celebrated A. van Wijngaarden's 25th year at the Math. Centrum.

.....

What is worth remembering about EDSAC's old program for B(x) ?

1: ERROR-ANALYSIS IS TRICKY.
2: BACKWARD ERROR ANALYSIS IS TRICKIER.

3: Most programmers will decline to perform any error-analysis,
and of those who try most may well fail.

It seems prudent to assume that an error-analyzed numerical subprogram must be exceptional, rather than commonplace, and precious not merely because of the function it computes but also because, as part of a larger numerical program being debugged, it is where the bug is far less likely to lurk than elsewhere. Look here last.

4: If a subprogram's numerical instability has so low a probability of being found when sought, why does it ***not*** have a low probability of affecting any one of the subprogram's users?

The reason turns out to be the large number of data-dependent branches in the subprogram; in EDSAC's B(x) these branches occur at the statement " μ := SignBit(x) ". In the notorious Pentium FDIV bug of 1994, the branches occurred at the selection of each quotient digit.

Both probabilities are about the same for most numerical subprograms in so far as they lack a large number of data-dependent branches. The risk, of being embarrassed by roundoff when using such a program without an error-analysis, turns out to attenuate rapidly as the precision of arithmetic increases, as we shall see. In short, ...

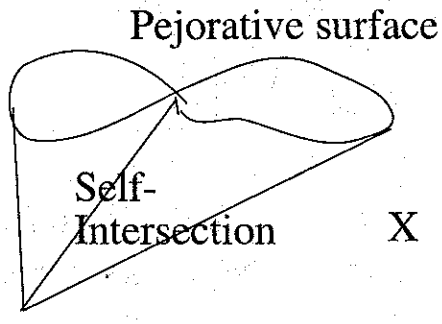
5: A good policy is to carry extra precision
for all intermediate computation.

We wish to compute $y = f(x)$. We actually compute $Y = F(X)$.
How do errors, seemingly so tiny when committed, get amplified?

Error inherited from $dx \approx X-x$: $dy := f(X) - f(x) \approx f'(x)dx$, so
$\|dy\|/\|dx\| \approx \|f'(x)\|$ at worst.

Amplification factor like $\|f'(x)\|$ is called a *Condition Number.*
When it is too big, the computation of $f(x)$ is called *Ill Conditioned.*

Ill condition can occur only near where $\|f'(x)\| = \infty$. This occurs on certain *Pejorative* surfaces (algebraic varieties, manifolds, ...) in the space of all possible data x .



Examples:

- Matrix inversion: Pejorative surface consists of Singular Matrices
  Self-Intersection contains matrices of Nullity > 1 .

- Polynomial Zeros: Pejorative Surf. = Polynomials with multiple zeros
  Self-Intersection = Polynomials with a triple zero.

- Eigenproblems: Pejorative Surf. = Matrices with a multiple eigenvalue
  Self-Intersection = Matrices with multiple eigenvalues

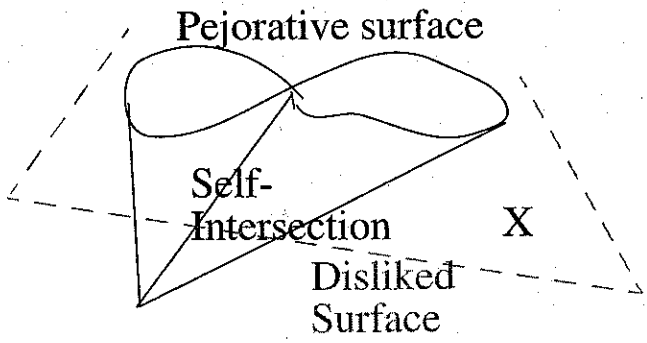
Typical Phenomenon:

Condition No. = O(distance(datum x to pejorative surface)$^{\text{power} < 0}$)

We wish to compute $y = f(x)$. We actually compute $Y = F(X)$. How do errors, seemingly so tiny when committed, get amplified?

Error inherited from $dx \approx X-x$ is amplified by Cond. No. like $\|f'(x)\|$.

Rounding errors within program $F(\ldots)$ are amplified too. Their amplification factors, like condition numbers, may grow infinite on certain surfaces. Some may be Pejorative surfaces. Others may contain data x that program $F(\ldots)$ dislikes, if it is numerically unstable there..



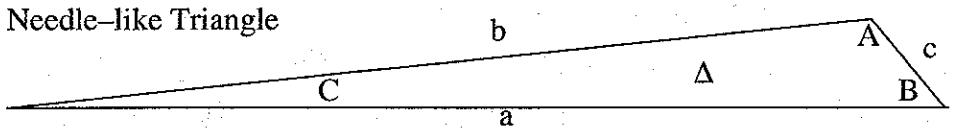
As do condition numbers, the factors amplifying roundoff grow infinite typically like a negative power of the distance from the datum x to a surface either Pejorative or merely Disliked.

Rounding error is negligible unless amplified beyond some tolerance. This can happen only for data x in some shell (slab, skin, tube, ...) obtained by thickening a Pejorative or Disliked surface by an amount proportional to some positive power of the roundoff level.

Conclusion:
The proportion (by volume, content, ...) of data x for which roundoff can be amplified intolerably is proportional to a positive integer power of the roundoff level, which in turn is an exponentially declining function of the number of sig. digits carried during arithmetic.

Carrying three more sig. dec. or ten more sig. bits reduces the incidence of embarrassment due to roundoff by a factor 1/1000 or less.

Case Study: Computing the area Δ of a needle-like triangle



A classical formula due to Heron of Alexandria,

$$\Delta = \sqrt{(s\cdot(s-a)\cdot(s-b)\cdot(s-c))} \quad \text{where} \quad s = (a+b+c)/2 \ ,$$

is the formula still taught in schools despite its numerical instability for needle-like triangles. A better scheme first sorts a, b, c so that $a \geq b \geq c$; this costs at most three comparisons. If $c-(a-b) < 0$ then the data are not side-lengths of a real triangle; otherwise compute its area

$$\Delta = \sqrt{(\ (a+(b+c))\cdot(c-(a-b))\cdot(c+(a-b))\cdot(a+(b-c))\ )/4} \ .$$

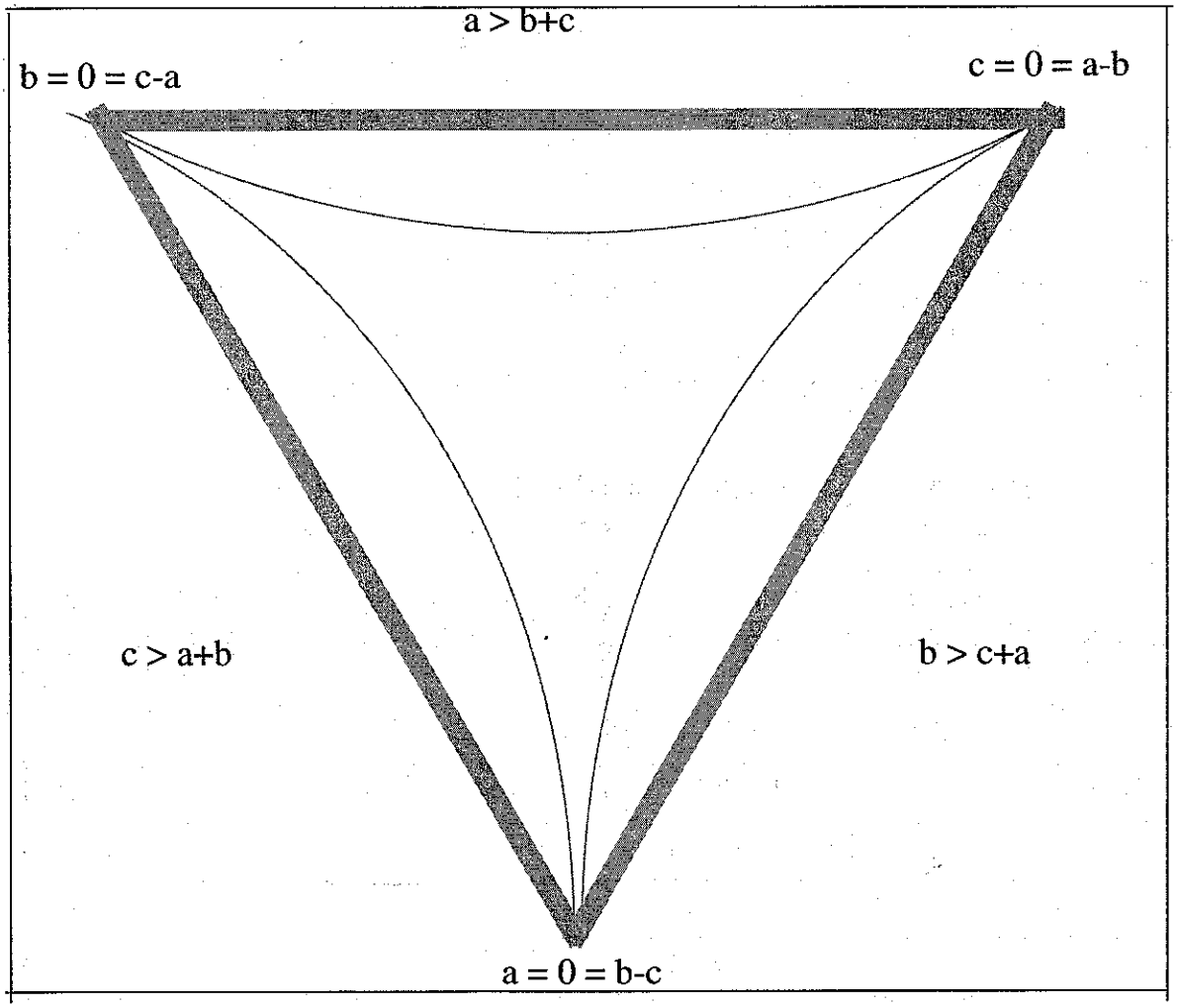
Most people don't know about this better formula, so it is little used.

Area Δ of a Triangle from the Lengths of its Sides
(calculations performed upon 4-byte `float` data)

Rounding mode	**Heron's Formula** (unstable in `float`)	**Better Formula** (stable in `float`)	**Heron's Formula** (all subexpressions `double` like K-R **C**)
a=12345679 > b=12345678 > c=1.01233995 > a–b			
to nearest	0.0	972730.06	972730.06
to +∞	17459428.0	972730.25	972730.06
to –∞	0.0	972729.88	972730.00
to 0	–0.0	972729.88	972730.00
a=12345679 ≥ b=12345679 > c=1.01233995 > a-b			
to nearest	12345680.0	6249012.0	6249012.0
to +∞	12345680.0	6249013.0	6249012.5
to –∞	0.0	6249011.0	6249012.0
to 0	0.0	6249011.0	6249012.0

Only incorrect results change drastically when the rounding mode changes, and old-fashioned Kernighan-Ritchie **C** gets fine results from an "unstable" formula by using `double` for all intermediates.

Heron's classical formula for Area Δ goes bad for a tiny fraction of triangular shapes. If the shapes are plotted in a plane region by taking side–lengths (a, b, c) as Barycentric Coordinates, these shapes whence comes chagrin lie in a narrow ribbon along the boundary:



Every point (a, b, c) in this big triangle represents a family of Similar triangles. Points near the thickened boundary represent needle-like triangles. Points between the center and the hyperbolic arcs represent triangles with all angles acute and utterly well-conditioned areas Δ ; some of them are needle-like. But Heron's formula computes Δ inaccurately in the thickened boundary no matter how well- or ill-conditioned Δ may be. The width of that thickened boundary, and also its area, is proportional to the roundoff level. This phenomenon is typical of geometric computations in two and three dimensions.

What is worth remembering about the foregoing case study?

1: Fix in advance the precision of the data and the tolerance for the result's inaccuracy. This fixes the population of possible data, and the population of tolerable results for each datum. Then, for every 10 bits or 3 dec. of extra precision carried during ALL intermediate arithmetic, the incidence of chagrin due to roundoff dwindles by a factor 0.001 or less.

Thus, the reliability of a wide range of computations can be enhanced either by successful error-analysis, or by carrying extra precision. Which is likely to cost less?

2: The most widely used hardware supports three floating-point formats. Does your compiler let you use the widest? Does your compiler let you rerun subprograms easily at a higher precision without annoyances like unwidened literal constants?

3: A numerically unstable subprogram can usually (not always) be identified by the drastic changes in its results when rerun on the same data (computed by other subprograms) in 3 different rounding modes mandated by IEEE 754. Does your compiler let you do this?

Different programming languages and compilers present different capture-cross-sections for error and unreliability to their users. In this respect, do your programming languages and compilers convey to you the benefit of current understandings of floating-point error-analysis?

And we haven't yet touched upon the handling of exceptions like division-by-zero, invalid operations like SQRT(–3.0), over/underflow, and the enigmatic "Inexact Result".