Marketing versus Mathematics

and other Ruminations on the Design of Floating-Point Arithmetic

Prof. W. Kahan
Math. Dept., and
Elect. Eng. & Computer Sci.
Univ. of Calif. @ Berkeley

(… upon receiving the IEEE's Emanuel R. Piore Award, 15 Aug. 2000)

Abstract: ~~Marketing is Bad; Mathematics is Good.~~

No; that is not my message. Rather, in so far as our technological activities have become ever more mathematical as well as market-oriented, mathematics and marketing have come to depend upon each other, and therefore must appreciate each other's needs, more than ever. This growing interdependence is illustrated in what follows by the design and implementation of floating-point arithmetic.

This document is at `http://www.cs.berkeley.edu/~wkahan/MktgMath.pdf`

(The title "Marketing versus Mathematics" was chosen by Prof. Alan Jay Smith.)

Contents:

Mathematics for Marketing and *vice-versa* page 3
A Good Example: The *Intimidation* Factor 5
Good and Bad in Marketing and in Mathematics 7
A Good Example: HP-92, …, HP-12C Financial Calculators 9
Market-motivated Mathematics is a Gamble 12
A Bad Example: QPRO 4.0 and QPRO for Windows 13
Why is Floating-Point almost all Binary nowadays? 18
Example: … literal constant " 25.4 " ; what's its nationality? 19
What is worth learning from the story about " 25.4", … ? 23
Besides its massive size, what distinguishes today's market for floating-point … ? 24
Case study: Kernighan-Ritchie C vs. ANSI C & Java 26
Example: Find the nearest point … 27
Old Kernighan-Ritchie C works better than ANSI C or Java **!** 28
The Intel 8087 Numeric Coprocessor's Marketing Vicissitudes 29
Floating-Point Arithmetic for a Mass Market … What was Best? 30
How do Errors get Amplified? By Singularities Near the Data. … 33
Testing floating-point software … A Frightening Example 36
What else should we remember about the foregoing examples? 42
A Bad Example of a Programming Language Ostensibly for a Mass Market 43
Appendix: How Directed Roundings May Help Locate Numerical Instability. 45
Over/Underflow Undermines Complex Number Division in Java 46
Four Rules of Thumb for Best Use of Modern Floating-point Hardware 47
Conclusion & Acknowledgement 48

Marketing is to the Laws of Economics as
High Technology is to the Laws of Mathematics, but …

Some Laws of Economics can be suspended for a while, though not repealed.

Every violation of a Law of Mathematics is always punished sooner or later.
(Alas, the punishment does not always befall the perpetrator.)

Why do I care? How are Marketing and Mathematics connected?

Without market forces on its side, technology is impotent.

By itself, technological superiority has never guaranteed commercial success, despite what R.W. Emerson may have said about building a better mousetrap.

Marketing is needed both before and after technological innovation,
first to descry what a targeted market needs and desires (though desire is fickle),
then to appraise how much a proposed innovation is worth to that market.

The crucial first step is understanding that market's needs.

Afterwards, the follow-through will inevitably entail mathematical analysis too.

How does mathematical analysis figure in a "Creative Activity"?

Much of what passes for imagination, innovation and invention
is actually perspiration, exploration and
painstaking elimination of the unfeasible and the unsuitable.

"… when you have eliminated the impossible, whatever remains,
however improbable, must be the truth …"
Sherlock Holmes in *The Sign of Four* by Sir Arthur Conan Doyle.

Only after the needs of a targeted market are well analyzed and understood
can unsuitable proposals be rejected before too late,
freeing up time to pursue a successful design.

Therefore, successful technical marketing depends upon mathematics,
and vice-versa.

How can Mathematics depend upon Marketing ? We'll see later.

A Good Example: The *Intimidation* Factor, or what a mathematician learned from a marketing man

In the early 1980s, the HP-15C Shirt-Pocket Programmable Calculator …
for students of Engineering, Physics, Chemistry, Statistics, …
all of the first two years of College Math. but Divs, Grads & Curls:

- Easy to program despite Reverse Polish (Stack) notation
- All elementary transcendental functions, x**!** , Combinatorial, …
- Potent and easy-to-use [SOLVE] and [INTEGRATE] keys
- Complex arithmetic as convenient as Real
- [+], [−], [x], [/], [1/x] keys work on small matrices too; Determinant, Norms, …

Since I expected freshmen who got the calculator to encounter its nonelementary mathematical capabilities before learning about them in classrooms, I proposed that its manual be augmented by some tutorial material about inverse functions, integrals, complex variables, matrices, and roundoff; — in short, a math. text.

A competent HP marketing man (I have forgotten his name, alas) asked me to contemplate how a potential buyer will react when the storekeeper shows him a box containing a tiny shirt-pocket calculator dwarfed by its instruction manual.

(My proposed tutorial became a separately purchaseable *HP-15C Advanced Functions Handbook* containing also a 39-page graduate micro-course on Error-Analysis.)

The *Intimidation* Factor continued …

Intimidation rears up nowadays when software comes with a 400 KB executable and a 3 MB *Help*-file written in `html` hypertext. The trouble is less with the size of that *Help*-file than with the burden upon a reader forced to wade through too big a portion of it to learn coherently (if he ever can) what the software does, whence its information comes and whither it goes, and where some of it stays (in hidden files?). No wonder people disincline to "Read the F— Manual."

The `html` format for *Help*-files invites a *Stream-of-Consciousness* style of disorganization, as if no single mind ever grasped what a reader needs to know.

Disorganized technical documentation, albeit comprehensive,
signals a failure of the marketing department
since it alone (if any) can hire English majors.

Not all marketing is competent marketing.

But bashing the marketing department is too much like shooting fish in a barrel.

It is more constructive for technically and, especially, mathematically trained personnel to collaborate with marketing people, and for these to accept such collaboration and, better, to know when to ask for it. This last requires a little mathematical competency within the marketing department.

How do you tell good marketing from not so good before it is too late?

I wish I knew. Instead, I can only offer examples of good and bad.

"Things are seldom what they seem; skim milk masquerades as cream."
Act II, *H.M.S. Pinafore* (1878) Gilbert & Sullivan

Not all mathematics is competent mathematics.

Some of it, even if logically correct, merely fills long-needed gaps.

How do you tell the good math. from the not so good?

Good mathematics promotes Intellectual Economy by helping us understand more while obliging us to memorize less without violating …

Einstein's Dictum: "Everything should be as simple as possible, but no simpler."

Disregard whatever I say that flunks this test.

A Good Example: HP-92, …, HP-12C Financial Calculators

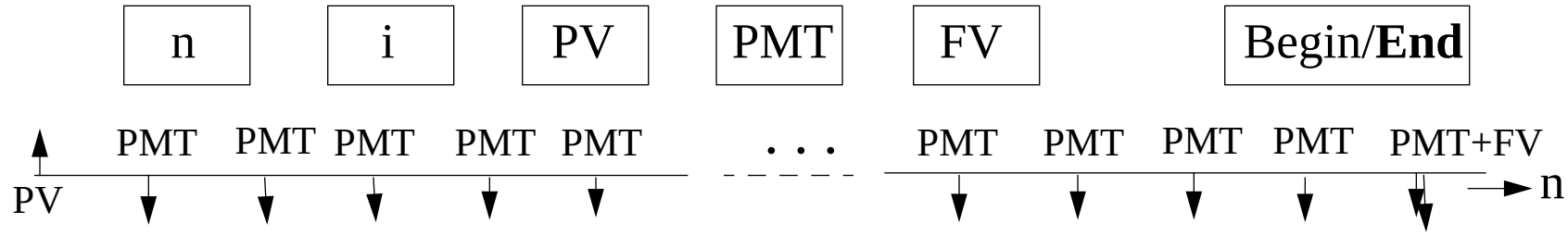
n	i	PV	PMT	FV	Begin/End

n = number of payment periods all of "equal" length (month, year, …)
i = interest rate (%) compounded in each period
PV = "Present Value" or special ("fee") payment at the beginning of the first period
PMT = each of n equal PayMenTs, one per period
[Begin/End] selects the beginning or end of every period for payments PMT
FV = "Future Value" or special ("balloon") payment at the end of the last period

Before 1976, these terms were used but interpreted differently for loans, leases, mortgages, amortizations, annuities, savings plans, …; consequently users of financial calculators and software had to look up different protocols for each kind of transaction. Besides, calculators and software could not handle some variants of these transactions, like balloon payments at the end, or fees at the beginning. Mistakes were common, and so were violations of *Truth in Lending* regulations.

Since relatively few "experts" understood the protocols, the market for do-it-yourself financial calculators was small, and many a purchaser was dissatisfied.

HP-92, …, HP-12C Financial Calculators continued …



In 1976 Roy Martin, a mathematician and programmer interested in marketing, proposed simpler interpretations of PV, PMT and FV that seem obvious now:

Give each a positive sign for cash flowing in,
a negative sign for cash flowing out.

Then the protocol became uniform for all transactions:

- Set the [Begin/End] switch as required for the intended transaction;
- Enter known values for four of the five variables n, i, PV, PMT, FV ;
- Press the fifth key to display the not-yet-known variable's value.

All the calculator had to do next was solve one equation for that fifth unknown:

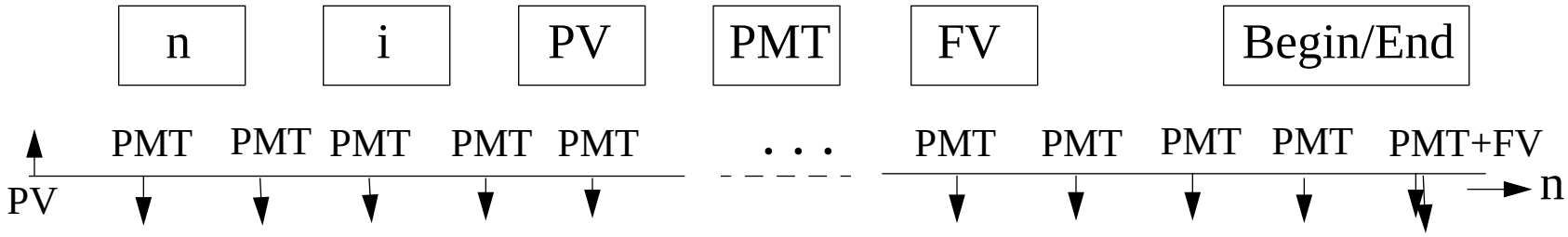
$$PV + PMT/(1+I)^{1-ß} + PMT/(1+I)^{2-ß} + \ldots + PMT/(1+I)^{n-ß} + FV/(1+I)^{n} = 0$$

where I = i/100 and ß = 1 for Beginning, 0 for End.

But nobody knew how to do so except in special cases or else intolerably slowly.

A calculator restricted to these special cases would have limited appeal.

HP-92, …, HP-12C Financial Calculators continued …



$$PV + PMT/(1+I)^{1-ß} + PMT/(1+I)^{2-ß} + \ldots + PMT/(1+I)^{n-ß} + FV/(1+I)^{n} = 0$$

(where I = i/100 and ß = 1 for Beginning, 0 for End . n can be many thousands.)

A calculator that solved this equation for only the known special cases would be a marketing failure for lack of follow-through. Instead, Roy Martin engaged a better mathematician (me) to help devise an equation-solving algorithm* that handled *all* cases quickly (under 250 arithmetic operations) and accurately, and could fit into the ROM available for micro-code. The resulting HP-92 became the progenitor of a very successful line of Hewlett-Packard financial calculators including the HP-12C, a shirt-pocket programmable calculator still being sold after 18 years and now the standard accessory for real estate brokers, financial planners, … . (Roy's write-up is pp. 22-8 of the Oct. 1977 HP Journal.)

*For the market-motivated mathematics behind such an algorithm see my web-page
`http://www.cs.berkeley.edu/~wkahan/Math128/RealRoots.pdf`

Market-motivated mathematics is a gamble:

It may reveal what you seek; - **A CLEAR WIN**

it may reveal that what you seek is not to be had; - **PARTIAL WIN**

it may fail to determine whether what you seek is feasible. - **A DRAW**

Mathematics cannot lose so long as it enhances your understanding of the issues.

Failure to follow through with market-motivated mathematical analysis is like a failure of due diligence, and can spoil market prospects and incur extra costs.

Here is a Bad Example,

extracted from "Bug Watch", p. 30 of *PC World*, 20 May 1993,
concerning what were then Borland's *Quattro Pro* spreadsheets:

(Extract begins.) "· ·

QPRO 4.0 and QPRO for Windows

Users report and Borland confirms a bug in the @ROUND function in QPRO 4.0 and QPRO for Windows: @ROUND may round decimal numbers ending with the digit 5 inconsistently. The number 31.875 rounded to two decimal places, for example, should always yield 31.88 but instead is sometimes rounded to 31.87 .

Although Borland says the problem is more likely to occur on systems lacking a math coprocessor, *PC World* was able to duplicate the problem on coprocessor-equipped systems.

Instead of using the @ROUND function, Borland recommends using the @INT function as shown in the following example:

To round the number in cell A1 to two decimal places, enter

@INT((A1 + 0.005)*100)/100 .

No fix is available for the bug; Borland plans to repair it in the next major release of both packages.

. . . . . ." (Extract ends.)

This recommended cure is worse than the disease.

The recommended cure is worse than the disease.

@ROUND(A1, 2) actually works correctly for numbers A1 near … 31.875 ,
rounding numbers A1 slightly less than 31.875 down to … 31.87 ,
numbers A1 equal or slightly bigger than 31.875 up to … 31.88 .

@INT((A1 + 0.005)*100)/100 malfunctions, yielding 31.88 wrongly for five numbers A1 strictly between 31.874999999999822 and 31.875 .

Trouble arises partly because Quattro displays at most 15 sig. dec., so that 29 numbers from 31.8749999999947… to 31.87500000000046… all display as " 31.87500…00 " but 14 @ROUND up to 31.88 and 15 down to 31.87 .

What QPRO shows you (decimal) is not what you get (binary).

This is a marketing department's blunder. They boasted about Quattro Pro's "fully integrated WYSIWYG display" feature without mentioning that it does not work for numbers the way a user should expect. In almost 1200 pages of documentation supplied with QPRO there is no mention of binary floating-point arithmetic nor of roundoff. Instead the reader can too easily misinterpret a few references to 15 or 16 sig. dec of precision as indications that no more need be said about QPRO's arithmetic. Actually much more needs to be said because some of it is bizarre. …

Decimal displays of Binary nonintegers cannot always be WYSIWYG.

Trying to pretend otherwise afflicts both customers and implementors with bugs that go mostly misdiagnosed, so "fixing" one bug merely spawns others. …

QPRO's @INT Bug, a Pious Fraud Exposed:

@INT(x) should yield the integer nearest x and no bigger in magnitude. *E.g.*,
we expect @INT(1.00…001) = 1 and @INT(0.999…999) = 0 .

More generally, whenever $0 < x < 2$ we expect @INT(x) == $(x \geq 1)$.

But QPRO's @INT does something unexpected:

Range of Stored Values x	x displays as …	@INT(x)	(x ≥ 1)	Notes
$1 - 14/2^{53}$ to $1 - 6/2^{53}$	0.999999999999999	0	0	
$1 - 5/2^{53}$	1.000000000000000	0	0	SAD
$1 - 4/2^{53}$ to $1 - 1/2^{53}$	1.000000000000000	1	0	BAD!
1 to $1 + 21/2^{52}$	1.000000000000000	1	1	

The discrepancy marked "BAD!" insinuates into spreadsheets inconsistencies almost impossible to debug. How can it be explained?

QPRO's @INT bug, a Pious Fraud Explained:

Range of Stored Values x	x displays as …	@INT(x)	(x ≥ 1)	Notes
$1 - 14/2^{53}$ to $1 - 6/2^{53}$	0.999999999999999	0	0	
$1 - 5/2^{53}$	1.000000000000000	0	0	SAD
$1 - 4/2^{53}$ to $1 - 1/2^{53}$	1.000000000000000	1	0	BAD!
1 to $1 + 21/2^{52}$	1.000000000000000	1	1	

Perhaps users of earlier versions of *Quattro* complained about a bug: For many an argument x displayed as an integer N , function @INT(x) yielded not N but N–1 whenever x was actually slightly smaller than N . To "fix" this bug, implementors of *Quattro* chose to round @INT(x)'s argument x to 53 sig. bits, multiply this by $1 + 1/2^{51}$, round the product to 53 sig. bits, and finally discard its fractional part. The line marked "SAD" shows that this kludge did not quite work; the line marked "BAD!" shows how @INT's mystery was deepened instead of hidden.

The right quick fix for the @ROUND and @INT (and other) bugs is to allow *Quattro*'s users to display up to 17 sig. dec. instead of no more than 15 .

The correct cure for the @ROUND and @INT (and some other) bugs is not to fudge their argument but to increase from 15 to 17 the maximum number of sig. dec. that users of QPRO may see displayed. Then distinct floating-point numbers in memory must appear distinct when displayed to 17 sig. dec.
(16 sig. dec. are too few to distinguish seven consecutive 8-byte floating-point numbers between $1024 - 4/2^{43} = 1023.9999\ 9999\ 9999\ 5453\ldots$ and $1024 + 4/2^{43} = 1024.0000\ 0000\ 0000\ 4547\ldots$.)

But no such cure can be liberated from little annoyances:

"0.8" entered would display as 0.80000000000000004 to 17 sig. dec.;

"0.0009875" entered would display as 0.000987500000000001 to 16 sig. dec.,

annoying users who did not expect QPRO to alter what they entered.

"32200/32.2" entered would display as 999.99999999999989 to 17 sig. dec.,

annoying users who expected roundoff to degrade only the last displayed digit of simple expressions, and confusing users who did not expect roundoff at all.

Decimal displays of Binary nonintegers cannot always be WYSIWYG.

For *Quattro*'s intended market, mostly small businesses with little numerical expertise, a mathematically competent marketing follow-through would have chosen either to educate customers about binary floating-point or, more likely, to adopt decimal floating-point arithmetic even if it runs benchmarks slower.

But Decimal is unlikely to supplant Binary hardware in the near future.

Why is Floating-Point almost all Binary nowadays?

Every finite conventional floating-point number has the form $s \cdot ß^{e+1-P}$ where

ß = Radix, fixed at Two for Binary, Ten for Decimal;
e = Exponent, a small signed integer within fixed limits;
s = Significand, a signed integer in the range $-ß^P < s < ß^P$; normally $|s| \geq ß^{P-1}$ too;
P = Precision, fixed at the number of "Significant Digits" carried by s .

- Binary is Better than any other radix for Error-Analysis (but not very much)
  because the relative density of floating-point numbers fluctuates by a factor of ß , which is rather less (2) for Binary than (10) for Decimal.
  E.g.: (Gaps among 1.000…000x) = 10·(Gaps among .9999…999x) .
- Binary is Intrinsically Simpler and Faster than Decimal (but not very much)
  because, to compete with the storage economy of Binary, Decimal has to be compressed, 3 dec. digits to 10 bits, when moved from arithmetic registers to memory, and decompressed when loaded back into registers.

Technical advantages may be more than offset by one marketing disadvantage:
Decimal displays of Binary nonintegers cannot always be WYSIWYG.

Even experienced programmers make mistakes frequently when they forget that Decimal and Binary nonintegers round off differently, and that converted literal Decimal constants lack qualities that are too often taken for granted.

Example: A program contains literal constant " 25.4 " ; what's its nationality?

Example: A program contains literal constant " 25.4 " ; what's its nationality?

This is an example *Contrived* out of subtleties that programmers typically overlook and misdiagnose.

How Many Millimeters in an Inch ?

Inch nationality	Era	Millimeters	Authority
British Imperial	Early 1800s	25.39954…	French
	Late 1800s	25.39997…	British
U.S.A. (now international)	Early 1900s	25.40005…	39.37 in./m.
	Late 1900s	25.4 exactly	Congress

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? It varies with the language and its compilers, all on the same IEEE 754 - compliant hardware:

As a 4-Byte float 25.4E0 = 25.399999619… ,
As an 8-Byte double 25.4D0 = 25.3999999999999858… ,
As a 10-Byte long-double 25.4T0 = 25.39999999999999999653… ,
As a 16-Byte quadruple 25.4Q0 = 25.3999999999999999999999999999877… .

Compared with historical variations, such discrepancies probably don't make headlines the way mi. *vs*. km. or knots *vs*. kmph. did when a Mars-bound spacecraft got lost.

Still, discrepancies tiny as these can provoke mysterious malfunctions in software.

Imagine a software project to collate the last two centuries' careful geodetic surveys of seismically active regions under the aegis of various past governments (some colonial), and now the U.N., undertaken to help predict volcanic eruptions and earthquakes. This software takes account of different units of length, most of them obsolete today.

What if irregularities in geodetic markers' movements seem to correlate with changes of the administrations in power when the markers moved? Do political upheavals influence continental drift? No; the irregularities are traced first to changes in units of length, and then to their associated constants like " 25.4 " converted by a compiler from decimal to binary as if they had been written " 25.4E0 ". Experts blame a programmer, who is now long gone, for not writing every such non-integer literal constant thus: " 25.4D0 " or " 25.4000000000000000 ". After source-texts are corrected and recompiled so that accuracy *improves*, severe and mysterious misbehavior arises on very rare occasions when different subprograms disagree upon the locations of maximal ground movements.

Disagreements arise because the constant " 2.54D0 " appears in a subprogram written by someone who worked with cm. instead of mm. even though compensatory factors " 10 " were introduced correctly:

$25.4 = 10 \cdot 2.54$ exactly, and
$25.4E0 = 10 \cdot 2.54E0$ *exactly*, and
$25.4T0 = 10 \cdot 2.54T0$ *exactly* ; but
$25.4D0 \neq 10 \cdot 2.54D0$ quite exactly because …

25.4 = 10 · 2.54 and 25.4E0 = 10 · 2.54E0 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. Consequently …

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 $1/2^{46} \approx 1.42_{10}-14$.

This difference between (25.4D0)·**d** and (2.54D0)·(10·**d**) suffices sometimes to put these values on different sides of a comparison, leading to different and inconsistent branches in subprograms that treat the same datum **d** . "Corrected" with more accurate constant literals "25.4D0" and "2.54D0" , the program would malfunction at a rare sprinkling of otherwise innocuous data **d** that the "incorrect" program with intended but paradoxically less accurate literals "25.4" and "2.54" had handled almost perfectly.

Could you have debugged the "Corrected" program?

Protest! " Only an overly naive programmer could expect the binary approximation of 25.4 to be exactly 10 times the binary approximation of 2.54 ."

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  f1  to very nearly  5·c0 .
c1  := f1 – f0 ;       …  exact in any  decent floating-point but not  exactly  c0 .
c10 := 10·c1 ;         …  exact in  Binary  or  Decimal  (but not  Hexadecimal).
```

Now, unless the compiler has "optimized" f0 and f1 away and taken c0 and c1 to be the same, c10 and c1 turn out to be respectively almost as good binary approximations of 25.4 and 2.54 as are to be had, and the first exactly 10 times the second. Can you see why this trick always works in Binary (though not always in Hexadecimal) floating-point? Many a programmer can't.

Programmers aware of the importance of such a relationship and the need for its defence can figure out how to enforce it. But because software tends to accrete like barnacles on a whale, rather than growing according to plan, obscure relationships and the rationale behind them tend to be forgotten as newly hired programmers take the places of the old.

The tale above about mm. per in. is a *fictional* didactically motivated over-simplified composite of common and often misdiagnosed bugs in programs that use a little floating-point. The point here is misdiagnosis. Programming languages also attract other bugs easier to diagnose. For example, a Java programmer who wrote wrongly " `C=(F-32)*(5/9)` " instead of " `C=(F-32)*5/9` " exposed a flaw worse in some languages than in himself. (`comp.lang.java.help` for 1997/07/02)

2.54 25.4 2.54D0 25.4D0 c1 c10

What is worth learning from the story about these numbers?

1: Programs transform information while preserving relationships, for instance loop-invariants, that connect input via intermediates to output strongly enough to imply its correctness. Floating-point rounding errors, as insidious as wood-rot because they are invisible in a program's text, weaken intended relationships among declared variables and literal constants, thus undermining the output's correctness. More about this later.

If unable to know in advance which relationships most need preservation, a programmer is obliged to try to preserve them *all* as well as possible by carrying *all* intermediate variables to the highest precision that does not run too slow (lest it never get run at all). On rare occasions this expedient fails, and then a programmer must resort to tricks like …

redundant variables, redundant parentheses, redundant computations of zero, …

to preserve crucial relationships like …

Symmetry, Monotonicity, Correlation, 10, Orthogonality, … .

These tricks may appear silly to a compiler writer tempted to "Optimize" them away.

2: Binary floating-point is best for mathematicians, engineers and most scientists, and for integers that never get rounded off. For everyone else Decimal floating-point is best because it is the only way *What You See* can be *What You Get*, which is a big step towards reducing programming languages' capture cross-section for programming errors.

But Binary will not soon go away. Trying to hide it (as QPRO's @INT did) can only make matters worse, especially for today's mass market.

"In every army big enough there is always somebody who does not get the message, or gets it wrong, or forgets it." … (a military maxim of unknown provenance)

Besides its massive size, what distinguishes today's market for floating-point arithmetic from yesteryears' ?

Innocence

(if not inexperience, naïveté, ignorance, misconception, superstition, …)

A mainframe costs so much to purchase and maintain that its owner can easily afford a numerical expert attached to the computer by an anklet like the one that stops an elephant from wandering away from its keeper at night. This expert can exploit the machine's idiosyncracies to extract the best performance compatible with tolerable reliability. But in a mass market computers and software cost too little to include an attached expert, so every practitioner must fend for himself. No matter how clever he may be about everything else, his amateur judgment about floating-point is predisposed towards speed in the absence of striking counter-indications. And published benchmarks pander to this predisposition.

Innocence

What else (besides a yen for Decimal) does marketing-motivated mathematics deduce from the mass market's innocence concerning floating-point arithmetic?

- Must Overdesign
- Must Majorize Past Practice
- Must Support Modern Rules of Thumb (Four are listed in an Appendix.)

Our vehicles, roads, appliances, … are all overdesigned a little to protect us from occasional inadvertence, our own and others'. If not for the sake of high professional standards, then for self-defence against software from the Internet that we use unwittingly we have to overdesign floating-point arithmetic, in both hardware and compilers, to diminish the incidence of avoidable malfunctions due to numerically inexpert but otherwise correct and clever programming.

Numerically expert software packages like LAPACK are so precious and costly to build that we dare not invalidate them by disabling some necessary hard-won arithmetic capability or property just because somebody thinks it not worth the bother. Newly marketed capabilities must *majorize* (match or surpass) the old.

Case study: 1970s Kernighan-Ritchie C *vs.* 1980s ANSI C & Java

Suppose `v`, `w`, `x`, `y` are four 4-byte `floats` and `Z` an 8-byte `double` related by the assignment

"`Z = v*w + y*z`".

What value does `Z` actually receive? It depends upon the dialect of C thus:

1980s ANSI C and Java:

`float` `p = v*w` rounded to 4-byte `float`;
`float` `q = y*z` rounded to 4-byte `float`;
`double` `Z = p + q` rounded to 8-byte `double`.

Rounding `p` and `q` to `float` seems in accord with an old rule of thumb that said the accuracy of an arithmetic operation need not much exceed the precision of its operands; and the precision of `Z` seems to exceed its accuracy. However, that old rule of thumb, inherited from the slide-rule era, was never quite right.

1970s K–R C :

`double` `P = v*w` exactly;
`double` `Q = y*z` exactly;
`double` `Z = P + Q` rounded to 8-byte `double`.

It seems like overkill to compute `Z` so accurately considering how much less precise the data `v`, `w`, `x`, `y` are, but actually this is almost always the better way.

Example: Given the equations $\mathbf{p}^T\cdot\mathbf{x} = \pi$ and $\mathbf{b}^T\cdot\mathbf{x} = ß$ of two planes that intersect in a line $£$, the point $\mathbf{z}$ in $£$ nearest a given point $\mathbf{y}$ turns out to be

$$\mathbf{z} = \mathbf{y} + \mathbf{v}\times(\,\mathbf{p}\cdot(ß-\mathbf{b}^T\cdot\mathbf{y}) - \mathbf{b}\cdot(\pi-\mathbf{p}^T\cdot\mathbf{y})\,)/\|\mathbf{v}\|^2 = (\,\mathbf{v}\cdot\mathbf{v}^T\cdot\mathbf{y} + \mathbf{v}\times(\mathbf{p}\cdot ß - \mathbf{b}\cdot\pi)\,)/\|\mathbf{v}\|^2$$

wherein $\mathbf{v} = \mathbf{p}\times\mathbf{b}$ (a cross-product in 3-space) and $\|\mathbf{v}\|^2 = \mathbf{v}^T\cdot\mathbf{v}$.

Try this data: $\mathbf{p}^T = [\ 38006,\ 23489,\ 14517\ ]$, $\pi = 8972$,
$\mathbf{b}^T = [\ 23489,\ 14517,\ 8972\ ]$, $ß = 5545$, and
$\mathbf{y}^T = [\ 1,\ –1,\ 1\ ]$, … all stored exactly as `floats`.

This data defines $£$ as the intersection of two nearly parallel planes, so tiny changes in data can alter $£$ and $\mathbf{z}$ drastically. More troublesome numerically are the many correlated appearances of the data $\mathbf{p}$ and $\mathbf{b}$ in the formulas for $\mathbf{z}$; though mathematically crucial, these correlations can be ruined by roundoff. (We could compute $\mathbf{z}$ in a way harmed less by roundoff but not a simple rational formula.)

Evaluating both formulas above for $\mathbf{z}$ naively in `float` arithmetic yields $\mathbf{z}_1^T = [\ 1,\ 1,\ –1\ ]$ and $\mathbf{z}_2^T = [\ 1,\ 1,\ –1.5\ ]$; but both points lie farther from both planes than about 0.65 . End-figure “errors” in data can’t account for that.

This naive arithmetic produces geometrically impossible results.

The correct point $\mathbf{z}^T = [\ 1/3,\ 2/3,\ –4/3\ ]$ is computed correctly rounded when all intermediate results (subexpressions and local variables) are evaluated in `double` before $\mathbf{z}$ is rounded back to `float`. This illustrates how …

Old Kernighan-Ritchie C works better than ANSI C or Java **!**

Old Kernighan-Ritchie C works better than ANSI C or Java !

Old K-R C works more nearly the way experienced practitioners worked before the 1960s when compilers, supplanting assembly-language, had to fit into tiny (by today's standards) memories and deliver executable code in one quick pass through a batch-oriented system. Experienced practitioners tolerated these crude compilers because we had nothing better, not because their arithmetic semantics conformed to widely taught rules of thumb we knew to be not quite right. In fact, we took perverse pride in how we got around perverse compilers and hardware.

Now C99 has appeared and offers implementors the opportunity to provide applications programmers the choice of floating-point semantics closer to their needs and better tuned to the capabilities of the most widely available hardware.

Almost all floating-point hardware nowadays conforms to IEEE Standard 754 for Binary Floating-Point Arithmetic. (There is also an underexploited IEEE Standard 854 for Decimal, but it will be mentioned no further here.) IEEE 754 was NOT designed exclusively for numerical experts (nor exclusively for my students) but was the result of a careful mathematical analysis of the needs of a mass market. It began in 1976 with Dr. John F. Palmer, then at Intel …

The Intel 8087 Numeric Coprocessor's Marketing Vicissitudes

It was a floating-point coprocessor to accompany Intel's 8086 and 8088 processors, which became ubiquitous later in IBM PCs. It was instigated by John Palmer, who foresaw a mass market (millions) despite disparagement by people who felt there was little demand for it. He offered to forego his salary if they would instead pay him $1 for every 8087 sold; they lacked the courage of their convictions, so his project proceeded. He hired me as a consultant.

John planned to put a complete floating-point arithmetic facility, including an extensive math. library of binary-decimal conversions and elementary functions like SQRT, LOG, COS, ARCTAN, …, all on one chip that a compiler writer could use instead of cobbling together his own facility of unpredictable quality. John and I squeezed as much of his plan as we could into 40,000 transistors.
(Simultaneously the same mathematical design went into Intel's 432, but this chip fell by the market's wayside.)

In 1980 we went to Microsoft to solicit language support for the 8087, for which a socket was built into the then imminent IBM PC. Bill Gates attended our meeting for a while and then prophesied that almost none of those sockets would ever be filled**!** He departed, leaving a dark cloud over the discussions.
Microsoft's languages still lack proper support for Intel's floating-point.

Floating-Point Arithmetic for a Mass Market

Intel's floating-point, now in Pentiums, AMD clones, and the new HP/Intel IA-64 Itanium, comes in three formats, all conforming to IEEE 754:

Format's Name	Wordsize in bytes	Sign bit	Exponent bits	Leading sig. bit	Sig. bits of Precision
Single Precision	4	1	8	Implicit	24
Double Precision	8	1	11	Implicit	53
Double Extended	at least 10	1	at least 15	unspecified	at least 64

The Extended format is intended not for data exchange but to enhance accuracy in *all* intermediate computations with little loss of speed, and to widen in response to future market demand. Intel's, AMD's and Motorola's Extended formats can fit in a word 10 bytes wide but avoid alignment delays if stored in 12- or 16-byte words. The IBM S/390 G5 processor's 16-byte Extended is a quadruple-precision format, with 113 sig. bits of precision, somewhat faster than some others' software implementations but still too slow.

(Some others' hardware, conforming to IEEE 754, lack any Extended format; this is adequate for the market formerly served by mainframes and minis that ran floating-point software like LINPACK written by numerical experts who avoided slow Extendeds.)

What Floating-Point was Best for a Mass Market?

It was Apple's *Standard Apple Numeric Environment* (SANE) implemented as early as 1983 on Apple III and then on 680x0-based Macintoshes. Besides supporting all three IEEE 754 floating-point formats and its directed roundings and exception-handling (Invalid Operation, Overflow, Underflow, Divide-by-Zero, Inexact, and their flags), SANE had an extensive library of elementary and financial functions, and decimal-binary conversions. It included advice for compiler writers, leading to numerically superb compilers for Pascal, Fortran and C provided by other vendors. When Motorola's 68881 floating-point coprocessor became available on the 68020-based Mac II, SANE ran fast as well as accurately, and applications programmers became enthusiastic boosters. The last machines to support SANE fully and well were the 68040-based Mac Quadras and Performas. Then marketing madness befell Apple.

John Sculley decided to ally Apple with IBM and adopt the latter's RS/6000 processor into "Power Macs". This RISC processor lacks a third Extended format, so that was the end of SANE. It was also almost the end of Apple, though for other reasons besides the abandonment of SANE.

Why does a mass market need the Extended format?

How can floating-point results be determined to be adequately accurate?

Error analysis can be too time-consuming even if carried out very competently.

Testing has proved too unreliable at revealing unsuspected numerical instability.

Computers have become so cheap that now most computations have little value, usually far less than the cost of establishing rigorously their (in)correctness.

Because arithmetic is cheap, but worthless if too slow, we have to try to design computer hardware and languages to ensure that vast amounts of approximate computations performed inexpertly, but otherwise about as fast as possible, will not misbehave intolerably often. Higher precision not too slow is the simplest single step towards that end. Extra precision beyond the data's has to run fast enough most the time to be used for all anonymous variables (subexpressions and literal constants) by default, and for almost all declared local variables.

The foregoing inference arises out of subtle mathematical analyses. …

How do Errors get Amplified? By Singularities Near the Data. …

First let $f(x)$ be a function of a (possibly vector) argument x ; varying x a little by Δx alters $f(x)$ by $f(x+\Delta x) - f(x) \approx f'(x)\cdot\Delta x$ roughly, where $f'(x)$ is the first derivative (or Jacobian array of first partial derivatives) of f . A little variation Δx turns into a big alteration $f'(x)\cdot\Delta x$ only if $\|f'(x)\|$ is huge, which can happen only if x is near a singularity where $\|f'(x)\|$ becomes infinite. In the space of all arguments x the locus of points where $\|f'(x)\| = \infty$ is some kind of (hyper)surface (or curve or manifold or variety) that we shall call "Pejorative". If big, $\|f'(x)\|$ is typically roughly proportional to some negative integer power of the distance from x to the nearest pejorative surface; usually

$$\|f'(x)\| = O(\ 1/(\text{distance from } x \text{ to the nearest pejorative surface})\ ) .$$

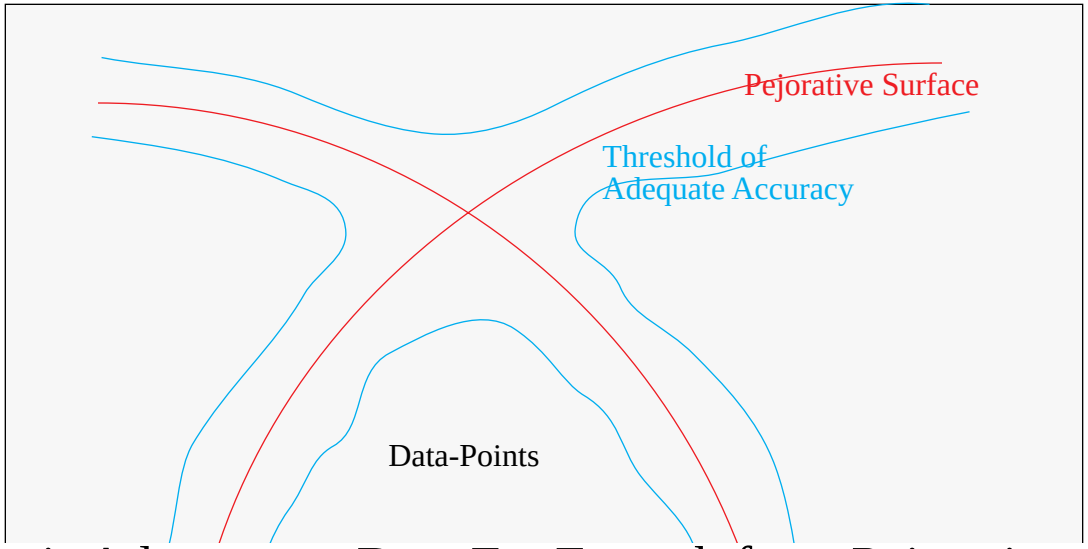
Next let $F(x; y)$ denote an idealized program intended to compute $f(x)$; here y stands for all the rounding errors in intermediate local variables and subexpressions subject to roundoff without which $y = 0$. For example, if $f(x) := ((3\cdot x–2)\cdot x–1)/(x–1)$ for $x \neq 1$, but $f(1) := 4$, then

$$F(x; y) = ((3\cdot x+y_1 – 2 + y_2)\cdot x+y_3 – 1 + y_4)/(x–1+y_5) + y_6 \quad \text{for } x \neq 1 .$$

In general $F(x; 0) = f(x)$ and, if y is tiny enough, $F(x; y) – f(x) \approx H(x)\cdot y$ where H is an array of first partial derivatives of $F(x; y)$ with repect to y evaluated at $y = 0$. If $\|H(x)\|$ is huge then, as before, it is roughly proportional to a negative integer power of the distance from x to the nearest pejorative surface associated with F , not with f . In fact, F may have some pejorative surfaces that f lacks, in which case we call them "Undeserved" or "Disliked" pejorative surfaces because the program F misbehaves there though the desired function f does not. In the example above, F introduces an undeserved pejorative point $x = 1$ because $f(x) = 3\cdot x + 1$ is well-behaved there; this artificial program F computes f in a numerically unstable way that dislikes data x too near 1 .

In short, intolerable losses of accuracy occur only at data x too near a pejorative surface, which may be undeserved if the program's algorithm is a numerically unstable way to compute the desired result.

All Accuracy is Lost if Data lie on a "Pejorative" Surface



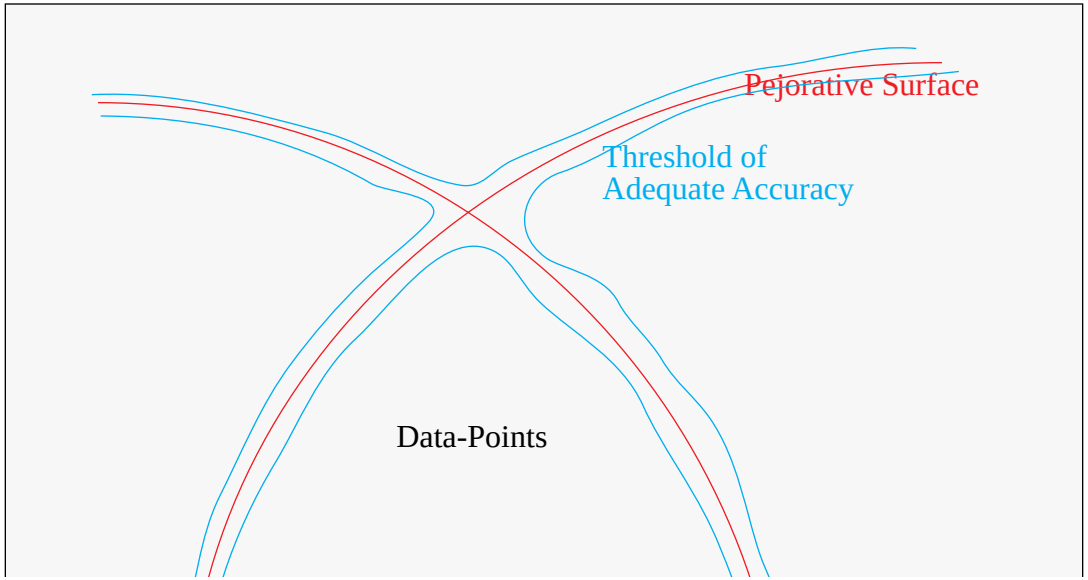
Accuracy is Adequate at Data Far Enough from Pejorative Surfaces

e.g.:

Data Points	Computed Result	Pejorative Data	Threshold Data
Matrices	Inverse	Singular Matrices	Not too ill-conditioned
Matrices	Eigensystem	Degenerate Eigensystems	Not too near degenerate
Polynomials	Zeros	Repeated Zeros	Not too near repeated
4 Vertices	Tetrahedron's Volume	Collapsed Tetrahedra	Not too near collapse
Diff'l Equ'n	Trajectory	Boundary-Layer Singularity	Not too "Stiff"

Numerically Unstable Algorithms introduce additional Undeserved Pejorative Surfaces often so "narrow" that they can be almost impossible to find by uninformed testing.

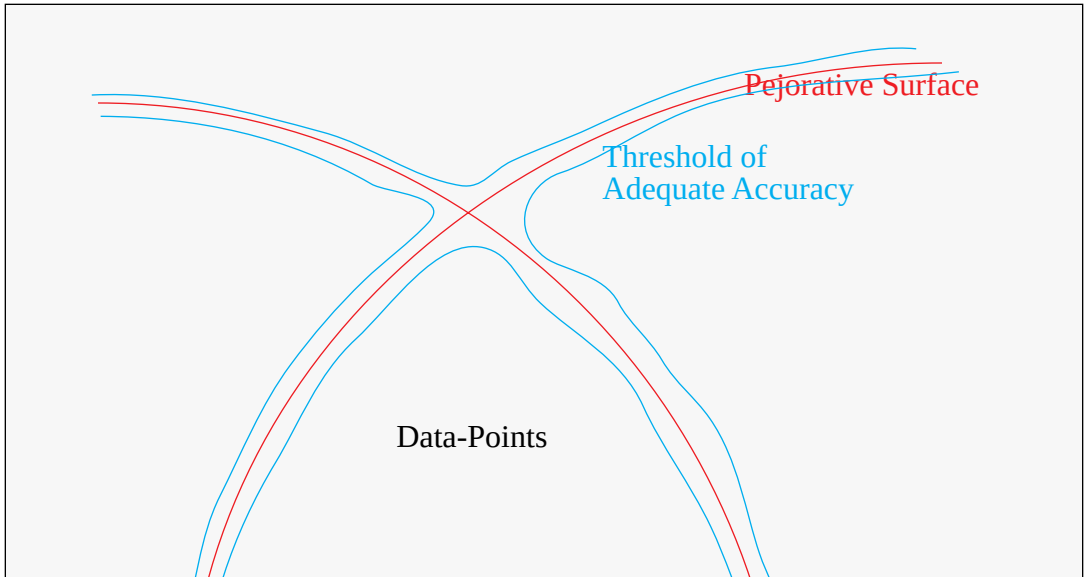
Carrying Extra Precision Squeezes Thresholds of Adequate Accuracy
Towards Pejorative Surfaces.



11 Bits of Extra Precision (beyond the data's) for *All* Intermediate Calculations
Diminishes the Incidence of Embarrassment due to Roundoff
by a Factor Typically Smaller than 1/2000 .

A Good Example: 3 dec. digits of extra precision in all HP calculators since 1976 make every keystroke function more reliable, simpler to implement and faster than if no more precision were available to the microprogrammer than is provided for all the calculator user's data and results. Roy Martin's financial calculator would have been unfeasible without those three extra internal digits.

Testing Software to Detect Unsuspected Numerical Instability Requires Locating Data Closer to a Pejorative Surface than the Threshold of Adequate Accuracy.



Whether due to an intrinsically ill-conditioned problem or to the choice of an unstable algorithm that malfunctions for otherwise innocuous data, mistreated data nearer a pejorative surface than the threshold of adequate accuracy can be ubiquitous and yet so sparse as hardly likely to be found by random testing**!** A recent instance is the 1994 Pentium FDIV bug; lots of stories about it are on the web, my web page included.

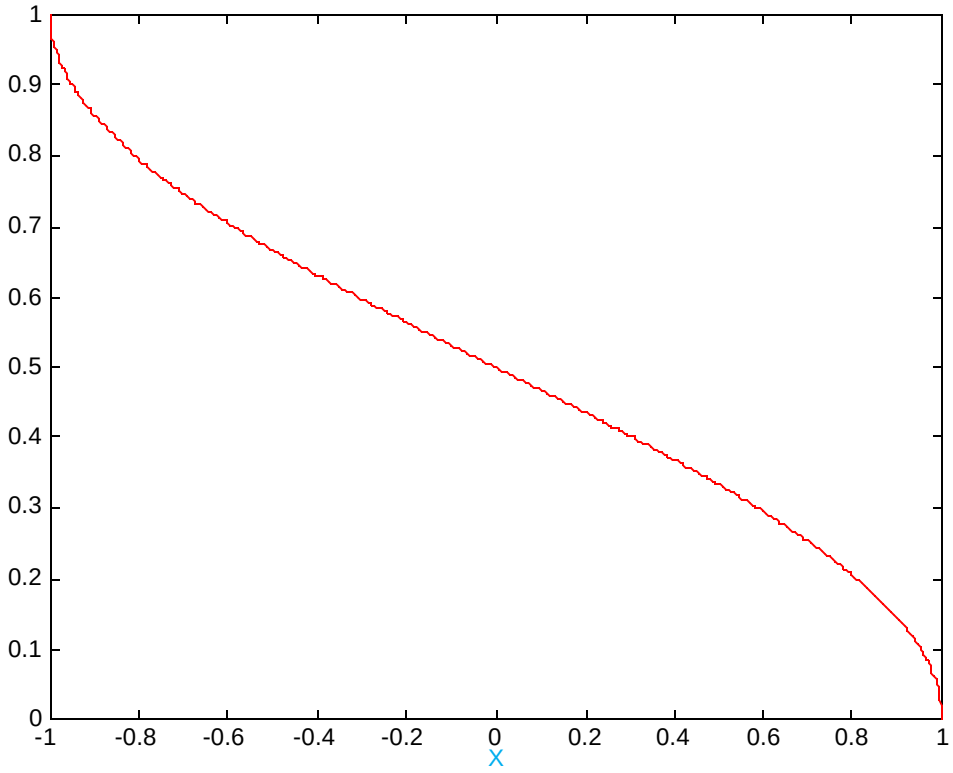
A Frightening Example: EDSAC's original arccos

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".

EDSAC's original arccos had errors 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.

From 1949 until A. van Wijngaarden exposed its "treachery" in 1951, the program computed $B(x) := \arccos(x)/\pi$ for users of EDSAC at Cambridge, England.



Here is EDSAC's simple program for $B(x) := B$, somewhat edited:

Set $x_1 := x = \cos(B\pi)$; $ß_0 := 0$; $B_0 := 0$; $t_0 := 1$; … Note $-1 \le x \le 1$.

While $(B_{j-1} + t_{j-1} > B_{j-1})$ do (for $j := 1, 2, 3, \ldots$ in turn)

{ $t_j := t_{j-1}/2$; … $= 1/2^j$.

$\mu_j := \text{SignBit}(x_j)$; … = 0 or 1 according as $x_j \ge 0$ or not.

$ß_j := |\mu_j - ß_{j-1}|$; … = 0 or 1 according as $\mu_j = ß_{j-1}$ or not.

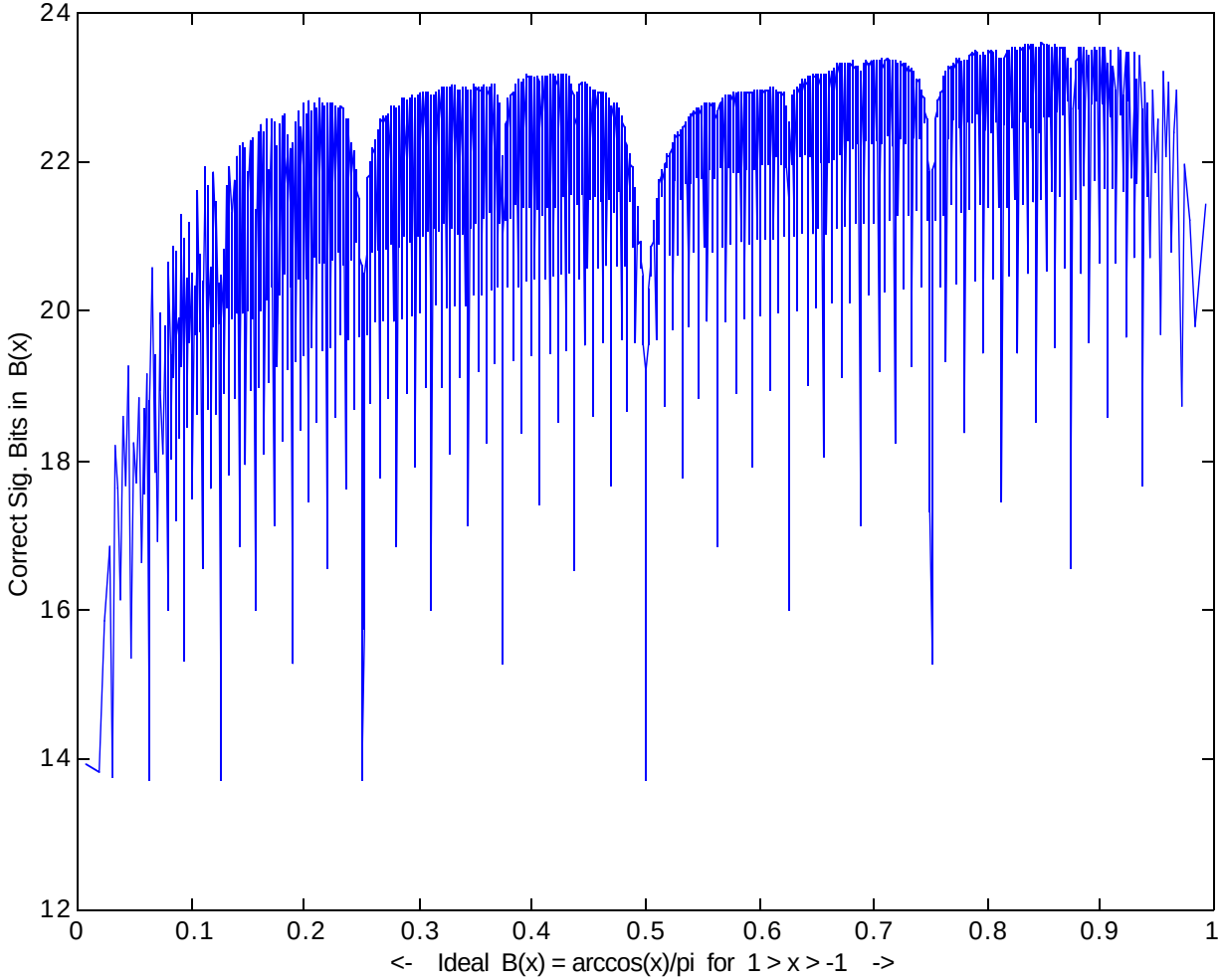
$B_j := B_{j-1} + ß_j \cdot t_j$; … $= \sum_{1 \le k \le j} ß_k/2^k < 1$, a binary expansion

$x_{j+1} := 2 \cdot x_j^2 - 1$ } . … $= \cos(2^j \cdot \arccos(x)) = \cos(2^j \cdot B\pi)$.

No subscript j appears in the actual program. The last equation $x_{j+1} = \cos(2^j \cdot B\pi)$ follows by induction from the identity $\cos(2\Theta) = 2 \cdot \cos^2(\Theta) - 1$. With each pass around the While-loop, the program commits at most one rounding error in the last statement " $x := 2 \cdot x^2 - 1$ ". Each pass amplifies this error by a factor near $4 \cdot x$ but its contribution in $ß \cdot t$ to B gets attenuated by a factor 1/2 . Let's see how bad it gets.

This program's B(x) was computed in `float` arithmetic for all two billion 4-byte `float` arguments x between –1 and +1 . (EDSAC ran its program in fixed-point.) For each of 2048 batches of nearly 2^{20} consecutive arguments, the worst error in B was recorded and plotted in the following figure. …

Of 24 Sig. Bits Carried, How Many are Correct in EDSAC's B(x) ?



Accuracy spikes downward wherever B(x) comes very near (but not exactly) a small odd integer multiple of a power of 1/2 . The smaller that integer, the wider and deeper the spike, down to half the sig. bits carried. Such arguments x are common in practice.

How could losing half the bits carried go unnoticed during tests?

The first explanation, presented in 1951 by Adrian van Wijngaarden (1953), included an estimate under 1% for the probability that one random test might expose the loss of more than 3 or 4 sig. bits. M.V. Wilkes (1971) said testers had laboriously compared EDSAC's B(x) with published tables at about 100 more-or-less random arguments x . Their probability of finding no bad error exceeded 1/3 . They were slightly unlucky.

How much worse is error in program B(x) than is inherited from its argument x ?

It has been argued that if x is not worth distinguishing from $x + \Delta x$ then neither is B(x) worth distinguishing from $B(x+\Delta x)$ nor from a computed value nearly as close to B(x) . The argument continues (wrongly*) thus: Since x too is a computed value unlikely to be accurate down to its last bit, we must tolerate errors in B(x) slightly bigger than variations in $B(x + \Delta x)$ arising from end-figure changes Δx in x . For instance B(x) is nearly 1 when x is very nearly -1 ; then tiny end-figure changes of order ε in x can alter B(x) as much as $\sqrt{(2\varepsilon)/\pi}$, thus altering almost half the sig. bits carried. Reasoning like this may explain why Morrison (1956, p. 206) seems to deem the program for B(x) about as accurate as the function B(x) deserves. It's all quite mistaken:

When x is near $1/\sqrt{2} = 0.7071\ldots$ and B(x) is near 0.25 , then tiny changes in x induce roughly equally tiny changes in B(x) , but EDSAC's program can lose almost half the sig. bits carried. So big a loss must be blamed not upon the function B(x) but upon a treacherous program for it that dislikes certain innocuous arguments x near 0, $\pm 1/\sqrt{2}$, $\pm\sqrt{(\ 1/2 \pm 1/\sqrt{8}\ )}$, … .

(* Backward error-analysis does not necessarily excuse errors by explaining them.)

Annotated Citations concerning EDSAC's arccos:

D.R. Morrison (1956) "A Method for Computing Certain Inverse Functions" pp. 202-8 of *MTAC* (*Math. Tables and Aids to Computation*) vol. **X**, corrected in 1957 on **XI**'s p. 314. On **XI**'s p. 204 is a short deprecatory "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 did not apologize for the loss of half the fixed-point bits 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 a computer program, and one of the first to mention floating-point if only in passing. At that time, floating-point error-analysis was widely deemed intractable. Now we know it isn't; it's just tiresome.

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. His contributions to Algol 68 are still remembered.

.....

How often do proficient practitioners' tests uncover errors like B's too late or never? Hard to say. Nobody's been keeping score. Some error-analysis must figure in competent tests of numerical software to obtain an indication of its error and to know how much error is tolerable. If these things are known to the testers for too few data sets, undeservedly pejorative data may easily remain hidden until customers stumble upon them.

No wonder experienced engineers still distrust predictions by numerical simulations not corroborated by different numerical methods and by experiments with actual devices.

"… the wages of sin is death; …" (*Romans* 6:23) but payment may be long delayed.

What else should we remember about the foregoing examples?

- Invisible yet ubiquitous, roundoff imperils all floating-point computation. However …
- The need for a rounding error-analysis can easily go unnoticed or unheeded.
- Rounding error-analysis is often too subtle or too costly for a programmer to perform.
- Without error-analysis, tests cannot reliably expose unsuspected numerical instability.

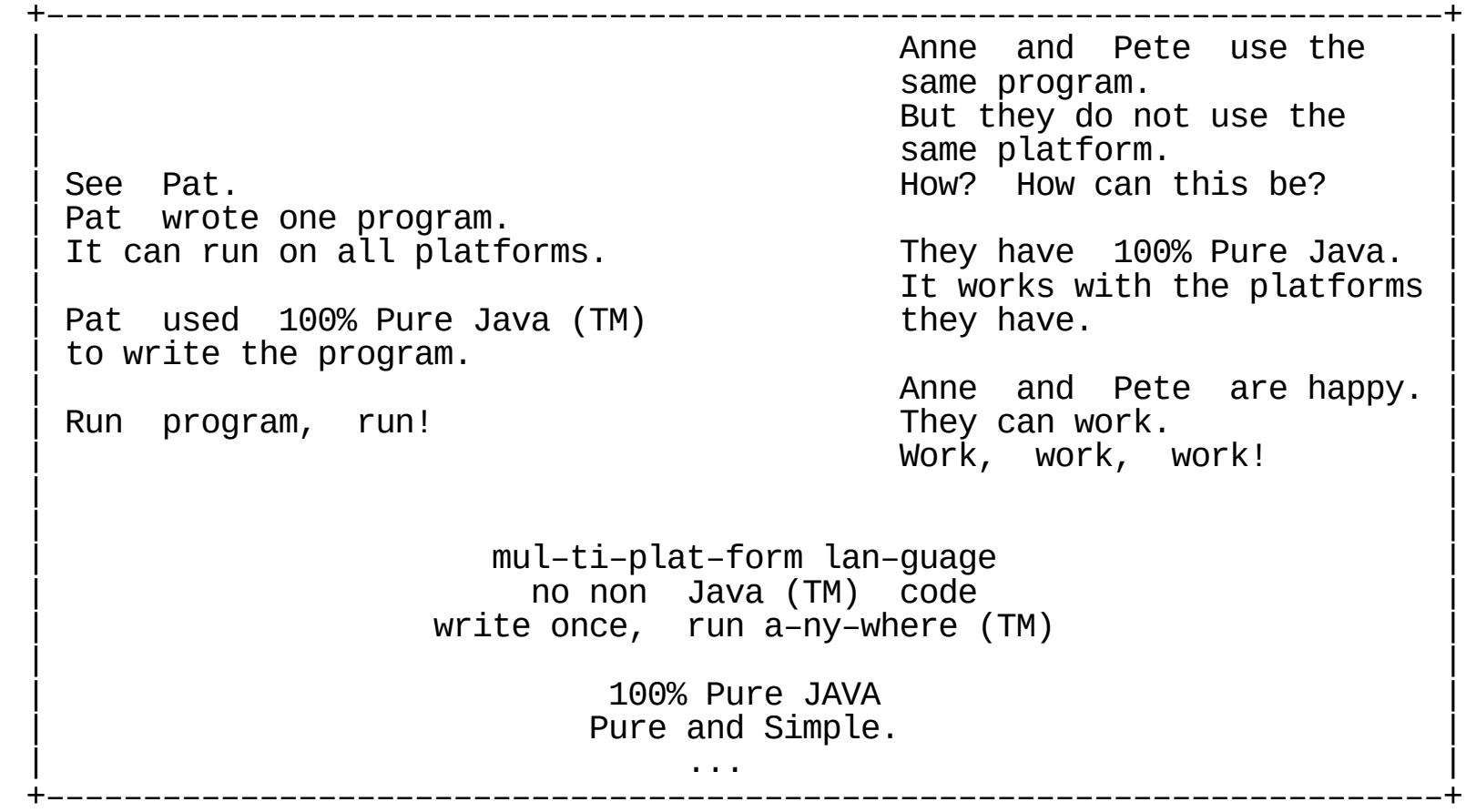
Therefore, provided it is not too slow, carrying extra arithmetic precision beyond the data's offers numerically inexpert programmers the simplest way to dininish the incidence of embarrassment due to roundoff. Designers of hardware and programming languages for a mass market can enhance their reliability by making this "simplest way" the default.

I am an error-analyst. I do not condone neglecting error-analysis when I advocate defending against that negligence by making fast higher precision the default. I do not condone reckless driving by advocating seat-belts and air-bags, nor condone loveless "recreational" sex by advocating realistic sex education and contraceptive advice for teen-agers. When we neglect precautions against disasters ensuing from predictable human folly, we allow harm to befall innocents too, maybe ourselves.

Not all designers appreciate the foregoing reasoning.

"No one … ever lost money by underestimating the intelligence of the great masses …."
H.L. Mencken (1880-1956) in the *Chicago Tribune*.

A Bad Example of a Programming Language Ostensibly for a Mass Market



This parody of puffery promoting 100% Pure Java™ for everyone everywhere filled page C6 in the **San Franisco Chronicle** Business Section of Tues. May 6, 1997.

It was paid for and copyrighted by **Sun Microsystems**.
Behind Sun's corporate facade must have twinkled a wicked sense of humor.

There are three reasons why Java's floating-point is ill suited to a mass market:

• Java supports `float` and `double`, but not in the preferable Kernighan-Ritchie C style, and lacks support for the wider Extended floating-point format built into the overwhelming majority of desktop machines running Java. This exacerbates the unreliability of numerically inexpert programmers' software.

• Java forbids use of the Directed Roundings mandated by IEEE 754 and built into practically all hardware upon which Java runs. Thus, roundoff variation techniques that usually locate numerically unstable subprograms are unavailable to Java programmers and their clients.

• Java forbids use of the exception-signaling Flags mandated by IEEE 754 and built into practically all hardware upon which Java runs. Thus, floating-point exceptions can ambush programs unless programmers introduce hordes of precautionary tests and branches that inflate a program's capture cross-section for error. For instance, try to protect division of complex numbers from underflow while guaranteeing that a quotient computable exactly will be computed exactly.

In short, despite Java's initial claims to conformity with IEEE 754, 'taint so.

(See "How Java's Floating-Point Hurts Everybody Everywhere" at
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`)

Appendix: How Directed Roundings *May* Help Locate Numerical Instability.

When roundoff is suspected of causing occasionally incorrect results, Directed Roundings mandated by IEEE 754 can help locate the offending subprogram by changing roundoff without changing disliked data that can be so hard to find. Investigate first any subprogram whose results change a lot when it is rerun under different directed roundings. Recall for example the nearest point's formulas $\mathbf{z} = \mathbf{y} + \mathbf{v}\times(\mathbf{p}\cdot(ß–\mathbf{b}^T\cdot\mathbf{y}) – \mathbf{b}\cdot(\pi–\mathbf{p}^T\cdot\mathbf{y}))/\|\mathbf{v}\|^2 = (\mathbf{v}\cdot\mathbf{v}^T\cdot\mathbf{y} + \mathbf{v}\times(\mathbf{p}\cdot ß–\mathbf{b}\cdot\pi))/\|\mathbf{v}\|^2$, wherein $\mathbf{v} = \mathbf{p}\times\mathbf{b}$ and $\|\mathbf{v}\|^2 = \mathbf{v}^T\cdot\mathbf{v}$. Recall too the evidently disliked `float` data $\mathbf{y}^T = [\ 1,\ –1,\ 1\ ]$ and

$\mathbf{p}^T = [\ 38006,\ 23489,\ 14517\ ]$, $\pi = 8972$, $\mathbf{b}^T = [\ 23489,\ 14517,\ 8972\ ]$, $ß = 5545$.

Let us compute $\mathbf{z}$ in sixteen ways: two from both formulas, times two from `float` and `double` arithmetics, times four for the rounding modes "to Nearest", "Down", "Up" and "to Zero".

Arithmetics	$\mathbf{z}_1$			$\mathbf{z}_2$		
`float` Nearest	1.0000000	1.0000000	-1.0000000	1.0000000	1.0000000	-1.5000000
`float` Down	0.5000000	-0.2000000	0.6000000	-0.0750000	0.5000000	0.2500000
`float` Up	1.0000000	-1.0000000	0.0	1.0000000	0.5000000	-0.2500000
`float` Zero	0.3333334	0.1904762	0.2380953	-0.0952381	0.5476190	-0.0476190
`double` (all)	0.33333333	0.66666666	-1.33333333	0.33333333	0.66666666	-1.33333333

Gross disparities with just `float` arithmetic corroborate a diagnosis of instability due to roundoff.

Caution: Don't always presume results to be correct just because they stayed almost unaltered as rounding modes changed. For an assessment of risks see "The Improbability of Probabilistic Error Analyses", `http://www.cs.berkeley.edu/~wkahan/improber.pdf` . Interpreting alterations in results caused by perturbations in input data is risky too without an error-analysis.

Appendix: Over/Underflow Undermines Complex Number Division in Java.

Given finite complex floating-point numbers $\mathbf{w} = u + \mathbf{i}v$ and $\mathbf{z} = x + \mathbf{i}y \neq \mathbf{0}$, naive formulas for the quotient $\mathbf{q} := \mathbf{w}/\mathbf{z} = r + \mathbf{i}s$, namely $d := |\mathbf{z}|^2 := x^2 + y^2$, $r := (x\cdot u + y\cdot v)/d$ and $s := (x\cdot v - y\cdot u)/d$, can be used safely only if d, $(x\cdot u + y\cdot v)$ and $(x\cdot v - y\cdot u)$ are not spoiled by over/underflow before r and s can be computed. This hazard, which afflicts half the arithmetic's exponent range, would be mitigated by the over/underflow and other warning flags mandated by IEEE 754 if Java did not deny them to programmers. Instead, since the naive formulas can deliver dangerously misleading results without warning, conscientious Java programmers must try something else. What?

Robert Smith's algorithm, devised for this situation, very nearly averts the hazard by computing …

if $|x| > |y|$ then { $t := y/x$; $p := x + t\cdot y$; $r := (u + t\cdot v)/p$; $s := (v - t\cdot u)/p$ }
else { $t := x/y$; $p := t\cdot x + y$; $r := (t\cdot u + v)/p$; $s := (t\cdot v - u)/p$ } .

But it runs slower because of a third division; it still generates spurious over/underflows at extreme arguments; and it spoils users' test cases when $\mathbf{q}$ should ideally be real or pure imaginary, or when $\mathbf{q}$, $\mathbf{z}$ and $\mathbf{w} = \mathbf{q}\cdot\mathbf{z}$ are all small complex integers, but the computed $\mathbf{q}$ isn't. Try these examples:

$(27 - 21\mathbf{i})/(9 - 7\mathbf{i})$, $(3 - 19\cdot\mathbf{i})/(1 - 3\cdot\mathbf{i})$ and $(31 - 5\cdot\mathbf{i})/(3 + 5\cdot\mathbf{i})$, all in `double`.

For C99, which does support IEEE 754's flags, Jim Thomas and Fred Tydeman have coded fast and accurate complex division by saving and clearing the flags, computing $\mathbf{q}$ the naive way, and then checking the flags for possibly spurious exceptions. Only rarely, just when necessary, do they recompute $\mathbf{q}$ after applying multiplicative scale factors, each a power of 2 to avert extra rounding errors, so that no over/underflow occurs unless $\mathbf{q}$ deserves it. To achieve a similar result in Java, a programmer has to compute an appropriate scale factor L in advance, set $\mathbf{Z} := L\cdot\overline{\mathbf{z}} = L\cdot x - \mathbf{i}L\cdot y$, and compute $\mathbf{q} := (\mathbf{Z}\cdot\mathbf{w})/(\mathbf{Z}\cdot\mathbf{z})$ with just two real divisions by $\mathbf{Z}\cdot\mathbf{z} = L\cdot|\mathbf{z}|^2$. To determine quickly an appropriate L (it must be a power of 2) is a challenge left to the diligent reader.

Four Rules of Thumb for Best Use of Modern Floating-point Hardware

all condensed to one page, to be supported by programming languages for a mass market

0. All Rules of Thumb but this one are fallible. Good reasons to break rules arise occasionally.

1. Store large volumes of data and results no more precisely than you need and trust.
   Storing superfluous digits wastes memory holding them and time copying them.

2. Evaluate arithmetic expressions and, excepting too huge arrays, declare temporary (local) variables *all* with the widest finite precision neither too slow nor too narrow.
   Here "too narrow" applies only when a variable in a floating-point expression or assignment is declared more precise than the hardware can support at full speed, and then arithmetic throughout the expression has to be at least as precise as that variable even if slowed by the simulation of its wider precision in software. Then also round infinitely precise literal constants and integer-typed variables to this wider precision. Elsewhere, all expressions containing only `float` variables should be evaluated, in the style of Kernighan-Ritchie **C**, in `double` or, better, `long double` if the hardware supports it at full speed. Of course every explicit cast and assignment to a narrower precision must round superfluous digits away as the programmer directs.

3. Objects represented by numbers should ideally have a parsimonious representation, called "fiducial" and rounded as rule 1 says, from which all other representations and attributes are computed using wider precision as rule 2 says.
   For instance, a triangle can be represented fiducially by `float` vertices from which edges are computed in `double`, or by `float` edges from which vertices are computed in `double`. Computing either in `float` from the other may render them inconsistent if the triangle is too obtuse. In general, a satisfactory fiducial representation can be hard to determine. Moreover, an object in motion may require *two* representations, both a moving `double` and, obtained from it by a cast (rounding down), a `float` fiducial snapshot.

Conclusion:

Slipshod marketing cannot identify its market well enough to guide technological development to a successful product, nor defend customers against slipshod engineering. To follow through properly, high-tech marketing departments need

- continuing close association with technicians developing products, and
- occasionally, access to independent mathematical competency.

These diverse talents must learn how to cooperate if they don't already know.

(Support graduate students at your local university's Mathematics department**!**)

Acknowledgement:

Almost half the foregoing pages respond to questions raised after the original shorter presentation on 15 Aug. 2000. I thank all questioners for assistance.

Many of my students, friends and colleagues helped do the work mentioned in these pages, especially IEEE Standards 754/854 for Floating-Point Arithmetic. In fact, they did most the work. The work is theirs; they are mine.