How Java’s Floating-Point Hurts Everyone Everywhere

by
Prof. W. Kahan and Joseph D. Darcy
Elect. Eng. & Computer Science
Univ. of Calif. @ Berkeley

Originally presented 1 March 1998
at the invitation of the
ACM 1998 Workshop on Java for
High–Performance Network Computing
held at Stanford University

`http://www.cs.ucsb.edu/conferences/java98`

This document: `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf` or
`http://www.cs.berkeley.edu/~darcy/JAVAhurt.pdf`

Abstract:

Java’s floating-point arithmetic is blighted by **five** gratuitous mistakes:

1. Linguistically legislated exact reproducibility is at best mere wishful thinking.

2. Of two traditional policies for mixed precision evaluation, Java chose the worse.

3. Infinities and NaNs unleashed without the protection of floating-point traps and flags mandated by IEEE Standards 754/854 belie Java’s claim to robustness.

4. Every programmer’s prospects for success are diminished by Java’s refusal to grant access to capabilities built into over 95% of today's floating-point hardware.

5. Java has rejected even mildly disciplined infix operator overloading, without which extensions to arithmetic with everyday mathematical types like complex numbers, intervals, matrices, geometrical objects and arbitrarily high precision become extremely inconvenient.

To leave these mistakes uncorrected would be a tragic **sixth** mistake.

. . . .

The following pages expand upon material presented on Sunday morning 1 March 1998 partly to rebut Dr. James Gosling’s keynote address “Extensions to Java for Numerical Computation” the previous morning (Sat. 28 Feb.); see his `http://java.sun.com/people/jag/FP.html` .

For a better idea of what is in store for us in the future unless we can change it, see
`http://www.sun.com/smi/Press/sunflash/9803/sunflash.980324.17.html` and
`http://math.nist.gov/javanumerics/issues.html#LanguageFeatures` .

We agree with James Gosling about some things like …

- Some kind of infix operator overloading will have to be added to Java.
- Some kind of Complex class will have to be added to Java.
- Some changes to the JVM are unavoidable.
- " 95% of the folks out there are completely clueless about floating-point." (J.G., 28 Feb. 1998)
  (*Maybe more than* 95% ?)

… and disagree with him about other things like …

- " A proposal to enhance Java's numerics would split the Java community into three parts:
  1. Numerical Analysts, who would unanimously be enthusiastically FOR it,
  2. Others, who would be vehemently AGAINST it, and
  3. Others who wouldn't care." (J.G., 28 Feb. 1998)
  Actually, Numerical Analysts would be as confused as everyone else and even more divided.
- Complex arithmetic like Fortran's ? *That's not the best way. The* C9X *proposal is better.*
- "Loose Numerics" ? *Sloppy numerics***!** IEEE 754 Double-Extended *supported properly is better.*
- … and many more …

To cure Java’s numerical deficiencies, we too propose to modify it
but not the way Gosling would modify it.

We call our modified Java language “ Borneo.”

Borneo’s design was constrained to be *Upward Compatible* with Java :

- Compiling Java programs with Borneo semantics should leave integer arithmetic unchanged and should change floating-point arithmetic at most very slightly.
- Any old Java class already compiled to bytecode should be unable to tell whether other bytecode was compiled under Java’s semantics or Borneo’s.
- Borneo is designed to require the least possible change to the Java Virtual Machine (JVM) that can remedy Java’s floating-point deficiencies.
- Borneo adds to Java as little infix operator overloading, exception flag and trap handling, control over rounding directions and choice of precisions as is essential for good floating-point programming. If you wish not to know about them, don’t mention them in your program.

For more information about Borneo : `http://www.cs.berkeley.edu/~darcy/Borneo` .

For more information about Floating-Point : `http://www.cs.berkeley.edu/~wkahan` .

What follows is NOT about Borneo.

What follows explains why Java has to be changed. By Sun. Urgently.

```
+------------------------------------------------------------------------+
|                                                                        |
|                                            Anne  and  Pete  use the    |
|                                            same program.               |
|                                            But they do not use the     |
|                                            same platform.              |
| See  Pat.                                  How?  How can this be?      |
| Pat  wrote one program.                                                |
| It can run on all platforms.               They have  100% Pure Java.  |
|                                            It works with the platforms |
| Pat  used  100% Pure Java (TM)             they have.                  |
| to write the program.                                                  |
|                                            Anne  and  Pete  are happy. |
| Run  program,  run!                        They can work.              |
|                                            Work,  work,  work!         |
|                                                                        |
|                                                                        |
|                       mul–ti–plat–form lan–guage                       |
|                         no non  Java (TM)  code                        |
|                    write once,  run a–ny–where (TM)                    |
|                                                                        |
|                             100% Pure JAVA                             |
|                            Pure and Simple.                            |
|                                 ...                                    |
+------------------------------------------------------------------------+
```

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.

Whom does Sun expect to use Java ?

Everybody.

Everybody falls into one of two groups:

1. A roundup of the usual suspects

These numerical experts, engineers, scientists, statisticians, … are used to programming in C, Fortran, Ada, … or to using programs written in those languages. Among their programs are many that predate IEEE Standard 754 (1985) for Binary Floating-Point Arithmetic; these programs, many written to be “Portable” to the computers of the 1970s, demand no more from floating-point than Java provides, so their translation into Java is almost mechanical.

2. Everybody else

“ 95% of the folks out there are completely clueless about floating-point.” (J.G., 28 Feb. 1998)
Their numerical inexpertise will not deter clever folks from writing Java programs that depend upon floating-point arithmetic to perform parts of their computations:

- Materials lists and blueprints for roofing, carpentry, plumbing, wiring, painting.
- Numerically controlled machine tools and roboticized manufacturing, farming and recycling.
- Customizable designs for home-built furniture, sailboats, light aircraft, go-karts, irrigation.
- Navigation for sailboats, light aircraft and spaceships while their pilots doze at the wheel.
- Economic and financial forecasts, estimated yield on investments, and portfolio management.
- Predictions of supply and demand, predictive inventory management, just-in-time delivery.
- …

There is no end to this list.

Q & A about selling computing to Everyone Everywhere:

What would happen to the market for automobiles if transmissions and chokes were not automatic, and if brakes and steering were not not power-assisted? Would all drivers be dextrous and strong, or would there be fewer cars and more chauffeurs as in “the good old days” ? What if standards for vehicular body-strength, lights, brakes, tires, seat-belts, air-bags, safety-glass, … were relaxed? Would cheaper cars and trucks compensate us for the cost of caring for more cripples?

Are such questions irrelevant to our industry? What will happen to the market for our computer hard- and software if we who design them fail to make them as easy to use as we can and also robust in the face of misuse? Misuse is unavoidable. Our industry’s vigor depends upon a vast army of programmers to cope with innumerable messy details some of which, like floating-point, are also complicated; and …

In every army large enough, someone fails to get the message, or gets it wrong, or forgets it.

Most programmers never take a competent course in Numerical Analysis, or else forget it. Over “ 95% of the folks out there are completely clueless about floating-point.” (J.G., 28 Feb. 1998) Amidst an overabundance of Java Beans™ and Class Libraries, we programmers usually hasten to do our job without finding the information we need to cope well with floating-point’s complexities. Like Coleridge’s *Ancient Mariner* afloat in

“ Water, water every where, nor any drop to drink ”

we are awash in (mis- and dis-)information. To filter what we need from the world-wide web, we must know first that we need the information, then its name. No “ Open Sesame! ” reveals what we need to know and no more.

We trust *some* information: Experience tells us how programmers are likely to use floating-point. Modern error-analysis tells us how to enhance our prospects for success. It’s more than merely a way for experts to validate (we hope) the software we distribute through prestigious numerical libraries like LAPACK and `fdlibm`. Error-analysis tells us how to design floating-point arithmetic, like IEEE Standard 754, moderately tolerant of well-meaning ignorance among programmers though not yet among programming language designers and implementors.

Java has evolved …

… from a small language targeted towards TV-set-top boxes and networked toaster-ovens

… to a large language and operating system targeted towards Everybody
Everything
Everywhere

… to challenge Microsoft's hegemony.

Microsoft is vulnerable because its flaky Windows system is not one system but many. Would-be vendors of software for MS Windows™ have to cope with innumerable versions, a legacy of partially corrected bugs, unresolved incompatibilities, … . Software often fails to install or later malfunctions because diversity among Windows systems has become unmanageable by the smaller software developers who cannot afford to pretest their work upon every kind of Windows system.

Java's " Write Once, Run Anywhere™ " tantalizes software vendors with the prospect of substantially less debugging and testing than they have had to undertake in the past.

This prospect has been invoked spuriously to *rationalize* Java's adherence to bad floating-point design decisions that mattered little in Java's initial niche market but now can't be reconciled with Java's expanded scope. Later we shall see why Java's expanded market would be served better by actual conformity to the letter and spirit of IEEE Standard 754 for Binary Floating-Point Arithmetic.

Pure Java’s Two Cruel Delusions:

“ Write Once, Run Anywhere™ ” and
Linguistically Enforced Exact Reproducibility of all Floating-Point Results

These *do* figure among ideals that should influence our decisions. So does Universal Peace. But some ideals are better approached than reached, and best not approached too directly.

(How do you feel about Universal Death as a direct approach to Universal Peace ?)

Pure Java’s two cruel delusions are inconsistent with three facts of computing life:

- Rush-to-Market engenders mistakes, bugs, versions, incompatibilities, conflicts, … as in Java’s oft revised AWT (Window interface), disputes between Sun and Microsoft, … . Intentionally and unintentionally divergent implementations of the JVM will exist inevitably.
- Compliance with standards that reinforce commercial disparities can be enforced only by the kind of power to punish heretics for which emperors and popes used to yearn. JavaSoft lacks even the power to prevent heretic versions of Java from becoming preponderant in some markets.
- A healthy balance between Stability and Progress requires an approach to the Management of Change more thoughtful than can be expected from business entities battling for market share.

Perfect uniformity and stability, if taken literally, are promises beyond Java’s power to fulfill.

Suppose for argument’s sake that the two cruel delusions were not delusions. Suppose they became actuality at some moment in time. This situation couldn’t last long. To understand why consider …

Complex Arithmetic Classes.

Complex Arithmetic Classes.

Why More than One?

JavaSoft would promulgate its 100% Pure Java™ Complex Arithmetic Class Library, and the Free Software Foundation would promulgate another (you’d have to install it yourself), and the Regents of the University of California would offer Kahan’s Complex Arithmetic Class Library.

How would Kahan’s differ from JavaSoft’s ? In line with the C9X proposal before ANSI X3J11, he includes an Imaginary Class and allows complex variables to be written as $x + \mathbf{\imath}{*}y$ or $x + y{*}\mathbf{\imath}$ (where $\mathbf{\imath} := \sqrt{(-1)}$ is the declared imaginary unit) instead of sticking to Fortran-like (x, y) as James Gosling has proposed. Kahan’s imaginary class allows real and complex to mix without forcing coercions of real to complex. Thus his classes avoid a little wasteful arithmetic (with zero imaginary parts) that compilers can have trouble optimizing away. Other than that, with overloaded infix arithmetic operators, you can’t tell the difference between Kahan’s syntax and Gosling’s.

Imagine now that you are developing software intended to work upon your customer’s Complex functions, perhaps to compute their contour integrals numerically and to plot them in interesting ways. Can you assume that your market will use only JavaSoft’s Complex classes? Why should you have to test your software’s compatibility with *all* the competing Complex classes? Wouldn’t you rather write just once, debug just once, and then run anywhere that the official Pure JavaSoft Complex Classes are in use, and ignore potential customers who use those heretic alternatives?

But some heresies cannot be ignored.

Example: **Borda’s Mouthpiece**, a classical two–dimensional fluid flow

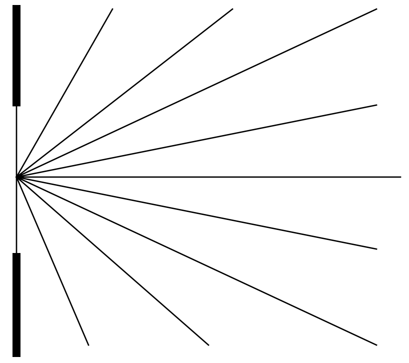
Define complex analytic functions

$$g(z) = z^2 + z \cdot \sqrt{z^2 + 1} \quad , \quad \text{and} \qquad F(z) = 1 + g(z) + \log(g(z)) \quad .$$

Plot the values taken by $F(z)$ as complex variable z runs along eleven rays

$$z = r \cdot \mathrm{i}\ , \quad z = r \cdot e^{4\mathrm{i}\cdot\pi/10}, \quad z = r \cdot e^{3\mathrm{i}\cdot\pi/10}, \quad z = r \cdot e^{2\mathrm{i}\cdot\pi/10}, \quad z = r \cdot e^{\mathrm{i}\cdot\pi/10}, \quad z = r$$

and their Complex Conjugates, taking positive r from near 0 to near $+\infty$.

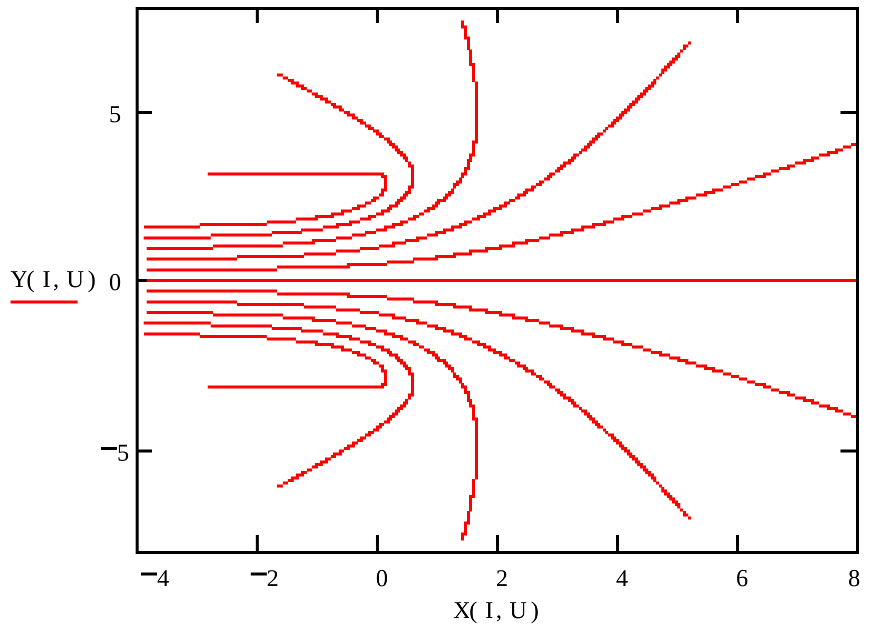


These rays are streamlines of an ideal fluid flowing in the right half-plane into a sink at the origin. The left half-plane is filled with air flowing into the sink. The vertical axis is a free boundary; its darker parts are walls inserted into the flow without changing it. The function $F(z)$ maps this flow *conformally* to a flow with the sink moved to $-\infty$ and the walls, pivoting around their innermost ends, turned into the left half-plane but kept straight to form the parallel walls of a long channel. (Perhaps the Physics is idealized excessively, but that doesn’t matter here.)

The expected picture, “ Borda’s Mouthpiece,” should show eleven streamlines of an ideal fluid flowing into a channel under pressure so high that the fluid’s surface tears free from the inside of the channel.

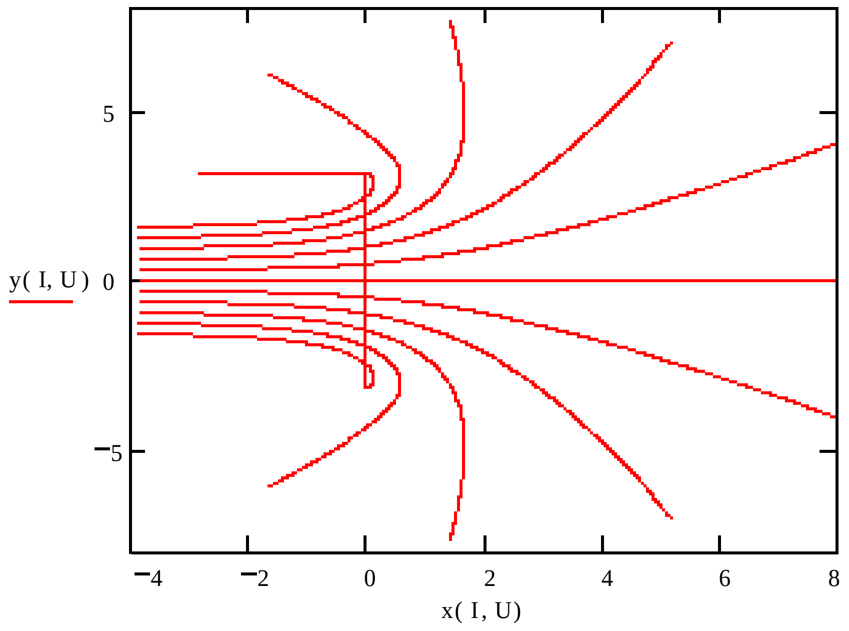
Borda’s Mouthpiece

Correctly plotted Streamlines



Plotted using C9X–like Complex and Imaginary

Streamlines should not cut across each other **!**

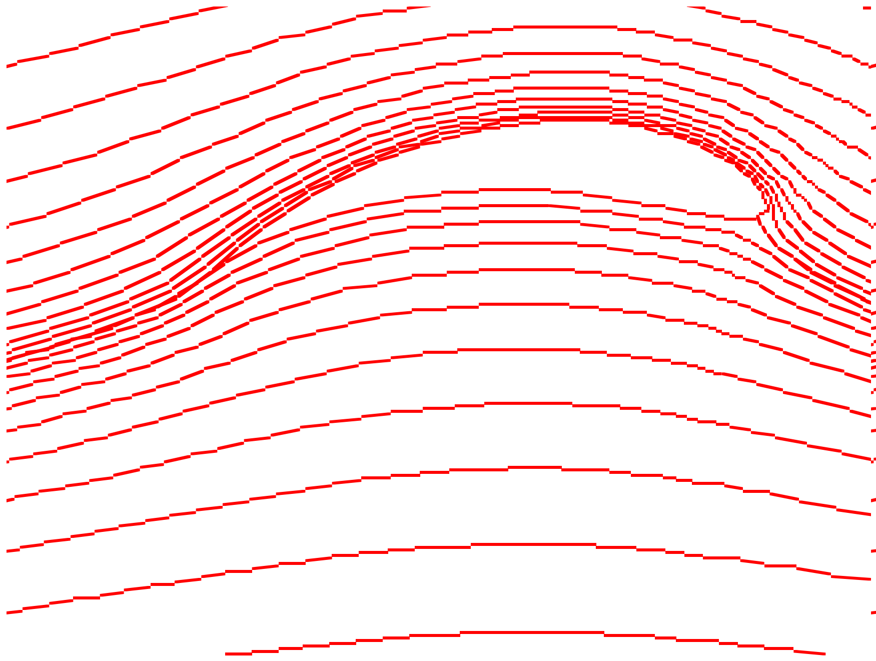


Misplotted using Fortran–like Complex

An *Ideal Fluid* under high pressure escapes to the left through a channel with straight horizontal sides. Inside the channel, the flow's boundary is *free*,— it does not touch the channel walls. But when –0 is mishandled, as Fortran-style Complex arithmetic must mishandle it, that streamline of the flow along and underneath the lower channel wall is misplotted across the inner mouth of the channel and, though it does not show above, also as a short segment in the upper wall at its inside end. Both plots come from the same program using different Complex Class libraries, first with and second without an Imaginary Class.

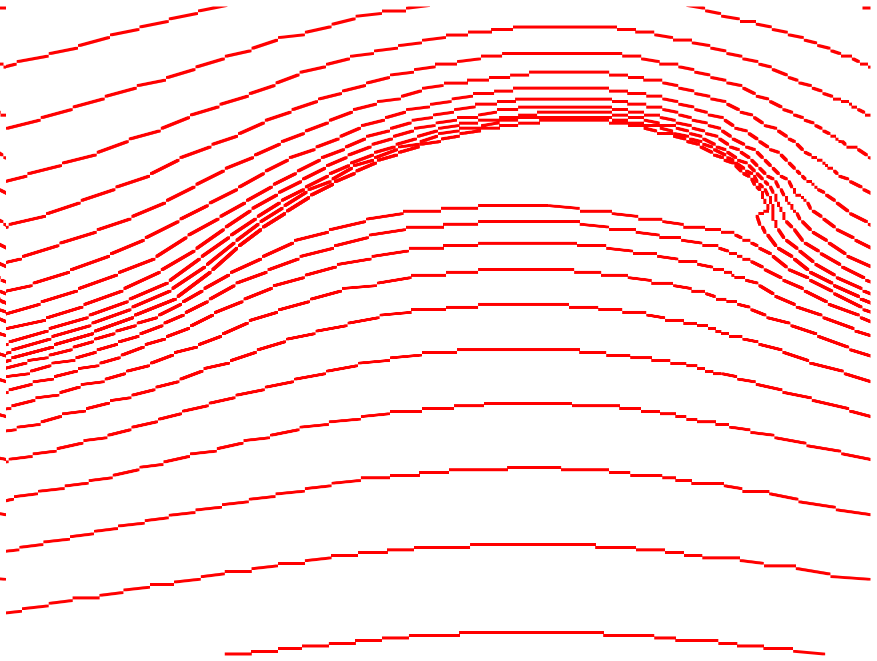
Lifting Flow past Joukowski's Aerofoil

Correctly Plotted Streamlines



Plotted using C9X–like Complex and Imaginary

Where is this wing's bottom ?



Misplotted using Fortran–like Complex

A circulating component, necessary to generate lift, speeds the flow of an idealized fluid above the wing and slows it below. One streamline splits at the wing's leading edge and recombines at the trailing edge. But when –0 is mishandled, as Fortran-style Complex arithmetic must mishandle it, that streamline goes only over the wing. The computation solves numerically nontrivial transcendental equations involving complex logarithms. Both plots come from the same program using different Complex Class libraries, first with and second without an Imaginary Class. Experienced practitioners programming in Fortran or C++ have learned to replace the split streamline by two streamlines, one above and one below, separated by as few rounding errors as produce a good-looking plot.

Why such plots malfunction, and a very simple way to correct them, were explained long ago in …

“ Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit ” by W. Kahan, ch. 7 in *The State of the Art in Numerical Analysis* (1987) ed. by M. Powell and A. Iserles for Oxford U.P.

A streamline goes astray when the complex functions SQRT and LOG are implemented, as is necessary in Fortran and in libraries currently distributed with **C**/**C**++ compilers, in a way that disregards the sign of ± 0.0 in IEEE 754 arithmetic and consequently **violates identities** like
SQRT(CONJ(Z)) = CONJ(SQRT(Z)) and LOG(CONJ(Z)) = CONJ(LOG(Z))
whenever the COMPLEX variable Z takes negative real values. Such anomalies are unavoidable if Complex Arithmetic operates on pairs (x, y) instead of notional sums $x + \imath \cdot y$ of real and imaginary variables. The language of pairs is *incorrect* for Complex Arithmetic; it needs the Imaginary type.

A controversial Complex Arithmetic Extension to the programming language **C** incorporating that correction, among other things, has been put before ANSI X3J11, custodian of the **C** language standard, as part of the **C9X** proposal. It is controversial because it purports to help programmers cope with certain physically important discontinuities by suspending thereat (and nowhere else) the logical proposition that “ `x == y` ” implies “ `f(x) == f(y)` ”. Many a programmer will prefer this anomaly to its alternatives.

. . . .

The moral of this story: There will always be good reasons (and bad) to call diverse versions of hard- and software, including mathematical software, by the same name.

Nobody can copyright “ Complex Class.”

Besides programs with the same name but designed for slightly different results,
there are programs with the same name designed to produce essentially the same results
as quickly as possible
which must therefore produce slightly different results on different computers.

Roundoff causes results to differ slightly not because different computers round arithmetic differently but because they manage memory, caches and register files differently.

Example: Matrix multiplication $C := A \cdot B$ … i.e. $c_{ij} := \sum_k a_{ik} \cdot b_{kj} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + a_{i3} \cdot b_{3j} + \dots$
To keep pipelines full and avoid unnecessary cache misses, different computer architectures have to perform multiplications $a_{ik} \cdot b_{kj}$ and their subsequent additions in different orders. In the absence of roundoff the order would not affect C because addition would be associative. Order affects accuracy only a little in the presence of roundoff because, for all suitable matrix norms $\|\dots\|$, $\|C - A \cdot B\|/(\|A\| \cdot \|B\|)$ cannot much exceed the roundoff threshold regardless of order, and this constraint upon C suffices for most applications even if
C varies very noticeably from one computer to another.

Ordering affects speed a lot. On most processors today, the most obvious matrix multiply program runs at least three times slower than a program with optimal blocking and loop-unrolling. Optimization depends delicately upon processor and cache details. For matrices of large dimensions, a code optimized for an UltraSPARC, about three times faster thereon than an unoptimized code, runs on a Pentium Pro (after recompilation) slower than a naive code and about six times slower than its optimal code. Speed degradation becomes worse on multi-processors.

Faster matrix multiplication is usually too valuable to forego for unneeded exact reproducibility.

Conclusion: Linguistically legislated exact reproducibility is unenforceable.

“ The merely Difficult we do immediately; the Impossible will take slightly longer.”
— Royal Navy maxim adopted during WW–II by American Seabees.

Ever-increasing diversity in hardware and software compounds the difficulty of testing new software intended for the widest possible market. Soon *“Difficult”* must become *“Impossible”* unless the computing industry collectively and programmers individually **share** a burden of …

Self-Discipline:

Modularize designs, so that diversity will add to your testing instead of multiplying it.

Know your market, or target only the markets you know;
exploit only capabilities you know to be available in all of your targeted markets.

Eliminate needless diversity wherever possible, though this is easier said than done; …
“ Things should be as simple as possible, but no simpler.” — Albert Einstein.

Java’s designers, by pursuing the elimination of diversity beyond the point of over-simplification, have turned a very desirable design goal into an expendable fetish.

They have mixed up two ideas:
Exact Reproducibility, needed by some floating-point programmers sometimes, and
Predictability within Controllable Limits, needed by all programmers all the time.
By pushing Exact Reproducibility of Floating-Point to an illogical extreme, the designers ensure it will be disparaged, disregarded and finally jettisoned, perhaps carrying Predictability away too in the course of a “ Business Decision ” that could all too easily achieve what the British call
“ Throwing Baby out with the bath water.”

The essence of programming is Control.

Control requires Predictability, which should be Java’s forte.
Java would impose “ Exact Reproducibility ” upon Floating-Point to make it Predictable.
But “ Exact Reproducibility ” is JavaSoft’s euphemism for “ Do as Sun’s SPARCs do.”
Thus it denies programmers the choice of better floating-point running on most other hardware.
Denied better choices, the programmer is not exercising Control but being controlled.

Throwing Baby out with the bath water:

When “Exact Reproducibility” of floating-point becomes too burdensome to implementors whose first priority is high speed, they will jettison Exact Reproducibility and, for lack of sound guidance, they will most likely abandon Predictability along with it. That’s happening now. That’s what Gosling’s “ Loose Numerics ” amounts to; a better name for it is “ Sloppy Numerics.”

To achieve Floating-Point Predictability:

Limit programmers’ choices to what is reasonable and necessary as well as parsimonious, and
Limit language implementors’ choices so as always to honor the programmer’s choices.

To do so, language designers must understand floating-point well enough to *validate*† their determination of “what is reasonable and necessary,” or else must entrust that determination to someone else with the necessary competency. But Java’s designers neglected timely engagement of Sun’s in-house numerical expertise, which would have prevented their floating-point blunders.

† **Footnote:** “*Validate* ” a programming language’s design? The thought appalls people who think such design is a *Black Art*. Many people still think Floating-Point is a *Black Art*. They are wrong too.

Java purports to fix what ain’t broken in Floating-point.

Floating-point arithmetic hardware conforming to IEEE Standard 754, as does practically all today’s commercially significant hardware on desktops, is already among the least diverse things, hard- or software, so ubiquitous in computers. Now Java, mistakenly advertised as conforming to IEEE 754 too, pretends to lessen its diversity by adding another one to the few extant varieties of floating-point.

How many significantly different floating-point hardware architectures matter today?

Four :

#0: Signal processors that may provide `float` and/or `float-extended` but not `double` .

#1: RISC-based computers that provide 4-byte `float` and 8-byte `double` but nothing wider.

#2: Power-PC; MIPS R-10000; H-P 8000 : same as #1 plus *fused* multiply-add operation.

#3: Intel x86, Pentium; clones by AMD and Cyrix; Intel 80960KB; new Intel/HP IA-64; and Motorola 680x0 and 88110 : the same as #1 plus a 10+-byte `long double` .

Over 95% of the computers on desktops have architecture #3 . Most of the rest have #2 . Both #3 and #2 can be and are used in restricted ways that match #1 as nearly as matters. All of #1, #2, #3 support Exception Flags and Directed Roundings, capabilities mandated by IEEE Standard 754 but generally omitted from architecture #0 because they have little value in its specialized market.

Java would add a fifth floating-point architecture #0.5 between #0 and #1 .

It omits from architecture #1 the Exception Flags and Directed Roundings IEEE 754 requires.

Java linguistically confuses the issues about floating-point Exceptions:

Java, like C++ , misuses the word “ Exception ” to mean what IEEE 754 calls a “ Trap.”
Java has no words for the five floating-point Events that IEEE 754 calls “Exceptions” :

Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result

These events are *not errors* unless they are handled badly.

They are called “Exceptions” because to any policy for handling them, imposed in advance upon all programmers by the computer system, some programmers will have good reasons to take exception.

IEEE 754 specifies a *default* policy for each exception, and allows system implementors the option of offering programmers an alternative policy, which is to *Trap* (jump) with specified information about the exception to a programmer-selected trap-handler. We shall not go into traps here; they would complicate every language issue without adding much more than speed, and little of that, to what flags add to floating-point programming. (Borneo would provide some support for traps.)

IEEE 754 specifies five *flags*, one named for each exception:

Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result

A flag is a type of global variable raised as a side-effect of exceptional floating-point operations. Also it can be sensed, saved, restored and lowered by a program. When raised it may, in some systems, serve an extra-linguistic diagnostic function by pointing to the first or last operation that raised it.

Java lacks these flags and cannot conform to IEEE 754 without them.

Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result

IEEE 754 specifies a *default* policy for each of these kinds of floating-point exception:

I Signal the event by raising an appropriate one of the five flags, if it has not already been raised.

II (Pre)substitute a default value for what would have been the result of the exceptional operation:

Name of Flag and Exception	(Pre)substituted Default Value
Invalid Operation	Not-a-Number (NaN), which arithmetic propagates; or a huge integer on overflowed flt.pt. —› integer conversion
Overflow	±∞ approximately, depending on Rounding Direction
Division-by-Zero	±∞ … Infinity exactly from finite operands.
Underflow	Gradual Underflow to a Subnormal (very tiny) value
Inexact Result	Rounded or Over/Underflowed result as usual

III Resume execution of the program as if nothing exceptional had occurred.

With these default values, IEEE 754’s floating-point becomes an *Algebraically Completed* system; this means the computer’s every algebraic operation produces a well-defined result for all operands.

Why should computer arithmetic be Algebraically Completed ?

What’s wrong with the Defaults specified for these Exceptions by IEEE 754 ?

Why does IEEE 754 specify a flag for each of these kinds of exception?

The next three pages answer these three questions and a fourth: What should Java do ?.

Why should computer arithmetic be Algebraically Completed ?

Otherwise some exceptions would have to trap. Then robust programs could avert loss of control only by precluding those exceptions (at the cost of time wasted pretesting operands to detect rare hazards) or else by anticipating them *all* and providing handlers for their traps. Either way is tedious and, because of a plethora of visible or invisible branches, prone to programming mistakes that lose control after all. For example, …

A Cautionary Tale of the Ariane 5 `( http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html )`

In June1996 a satellite-lifting rocket named *Ariane 5* turned cartwheels shortly after launch and scattered itself, a payload worth over half a billion dollars, and the hopes of European scientists over a marsh in French Guiana. A commission of inquiry with perfect hindsight blamed the disaster upon inadequate testing of the rocket’s software.

What software failure could not be blamed upon inadequate testing ?

The disaster can be blamed just as well upon a programming language (Ada) that disregarded the default exception-handling specifications in IEEE Standard 754 for Binary Floating-Point Arithmetic. Here is why:

Upon launch, sensors reported acceleration so strong that it caused Conversion-to-Integer Overflow in software intended for recalibration of the rocket’s inertial guidance while on the launching pad. This software could have been disabled upon rocket ignition but leaving it enabled had mistakenly been deemed harmless. Lacking a handler for its unanticipated overflow trap, this software trapped to a system diagnostic that dumped its debugging data into an area of memory in use at the time by the programs guiding the rocket’s motors. At the same time control was switched to a backup computer, but it had the same data. This was misinterpreted as necessitating strong corrective action: the rocket’s motors swivelled to the limits of their mountings. Disaster ensued.

Had overflow merely obeyed the IEEE 754 default policy, the recalibration software would have raised a flag and delivered an invalid result both to be ignored by the motor guidance programs, and the Ariane 5 would have pursued its intended trajectory.

The moral of this story: A trap too often catches creatures it was not set to catch.

Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result

What’s wrong with the Default values specified for these Exceptions by IEEE 754 ?

Its is not the only useful way to Algebraically Complete the real and complex number systems.
(Were there just one we’d all learn it in school and Over/Undeflow would be the only floating-point exceptions.)

Other ways? For instance, instead of two infinities with $1/(-0) = -\infty <$ (every finite real number) $< +\infty = 1/(+0)$, a completion with just one $\infty = -\infty = 1/0$ has its uses. Another completion has no ∞ , just NaN . There are illegitimate completions too, like APL’s $0/0 = 1$. Every legitimate completion must have this property:

> In the absence of roundoff and over/underflow, evaluations of an algebraic expression that differ because the customary commutative, distributive, associative and cancellation laws have been applied can yield at most two values and, if two, one must be NaN . For instance, $2/(1+1/x) = 2$ at $x = \infty$ but $(2\cdot x)/(x+1)$ is NaN .

By majority vote a committee chose the particular completion specified by IEEE 754 because it was deemed less strange than others and more likely to render exceptions ignorable. It ensures that, although Invalid Operations and Overflows can rarely be ignored for long, in their absence Underflows can usually be ignored, and Division-by-Zero and Inexact can almost always be ignored. Java too has adopted the IEEE 754 completion as if there were nothing exceptional about it.

But a programmer can have good reasons to take exception to that completion and to every other since they jeopardize cancellation laws or other relationships usually taken for granted. For example, $x/x \neq 1$ if x is 0 or not finite; $x-x \neq 0 \neq 0\cdot x$ if x is not finite. After non-finite values have been created they may invalidate the logic underlying subsequent computation and then disappear: (finite/Overflow) becomes 0 , $(\text{NaN} < 7)$ becomes false , … . Perhaps no traces will be left to arouse suspicions that plausible final results are actually quite wrong.

Therefore a program *must* be able to detect that non-finite values have been created in case it has to take steps necessary to compensate for them.

Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result

Why does IEEE 754 specify a flag for each of these kinds of exception?

Without flags, detecting rare creations of ∞ and NaN before they disappear requires programmed tests and branches that, besides duplicating tests already performed by the hardware, slow down the program and impel a programmer to make decisions prematurely in many cases. Worse, a plethora of tests and branches undermines a program's modularity, clarity and concurrency.

With flags, fewer tests and branches are necessary because they can be postponed to propitious points in the program. They almost never have to appear in lowest-level `methods` nor innermost loops.

Default values and flags were included in IEEE 754 because they had been *proved necessary* for most floating-point programmers even though a few numerical experts could often find complicated ways to get around the lack of them. And, in the past, if an expert bungled the avoidance of floating-point exceptions his program's *trap* would reveal the bungle to the program's user.

Without Traps nor Flags, Java's floating-point is *Dangerous* .

What should Java do instead?

Java could incorporate a standardized package of native-code flag-handling `methods`. The Standard Apple Numeric Environment (SANE) did that (*Apple Numerics Manual* 2d ed. 1988, Addison-Wesley). But leaving flags out of the language predisposes compile-time optimization to thwart the purpose of flags while rearranging floating-point operations and flag-references. Borneo would make flags part of the language and let programmers specify in a `method`'s signature conventions for copying, saving, restoring and merging flags. Java should do the same. Of course, a programmer can disregard all that stuff, in which case users of his `methods` may be grateful for the insights into his oversights that flags reveal afterwards.

By now 95% of readers should be aware that there is more to floating-point than is taught in school.

Moreover, much of what is taught in school about floating-point error-analysis is wrong.

Because they are enshrined in textbooks, ancient rules of thumb dating from the era of slide-rules and mechanical desk-top calculators continue to be taught in an era when numbers reside in computers for a billionth as long as it would take for a human mind to notice that those ancient rules don’t always work. They *never* worked reliably.

13 Prevalent Misconceptions about Floating-Point Arithmetic :

1• Floating–point numbers are all at least slightly uncertain.
2• In floating–point arithmetic, every number is a “ Stand–In ” for all numbers that differ from it in digits beyond the last digit stored, so “ 3 ” and “ 3.0 E0 ” and “ 3.0 D0 ” are all slightly different.
3• Arithmetic much more precise than the data it operates upon is needless, and wasteful.
4• In floating–point arithmetic nothing is ever exactly 0 ; but if it is, no useful purpose is served by distinguishing +0 from -0 . (We have already seen on pp. 13 - 15 why this might be wrong.)
5• Subtractive cancellation always causes numerical inaccuracy, or is the only cause of it.
6• A singularity always degrades accuracy when data approach it, so “ Ill–Conditioned ” data or problems deserve inaccurate results.
7• Classical formulas taught in school and found in handbooks and software must have passed the Test of Time, not merely withstood it.
8• Progress is inevitable: When better formulas are found, they supplant the worse.
9• Modern “ Backward Error-Analysis ” explains all error, or excuses it.
10• Algorithms known to be “ Numerically Unstable ” should never be used.
11• Bad results are the fault of bad data or bad programmers, never bad programming language design.
12• Most features of IEEE Floating-Point Standard 754 are too arcane to matter to most programmers.
13• “ ‘ Beauty is truth, truth beauty.’ — that is all ye know on earth, and all ye need to know.” ... from Keats’ *Ode on a Grecian Urn* . (In other words, you needn’t sweat over ugly details.)

“ The trouble with people is not that they don’t know
but that they know so much that ain’t so.”
… *Josh Billings’ Encyclopedia of Wit and Wisdom* (1874)

The foregoing misconceptions about floating-point are quite wrong, but this is no place to correct them all. Several are addressed in `http://http.cs.berkeley.edu/~wkahan/Triangle.pdf` .

Here we try first to upset beliefs in a few of those misconceptions, and than show how they combine with historical accidents to mislead designers of modern programming languages into perpetuating the floating-point mistakes built into so many old programming languages. To succeed we must undermine faith in much of the floating-point doctrine taught to language designers.

Consider “ Catastrophic Cancellation,” a phrase found in several texts. Many people believe that …
- Catastrophically bad numerical results are <u>always</u> due to massive cancellation in subtraction.
- Massive cancellation in subtraction <u>always</u> results in catastrophically bad numerical results.

Both are utterly mistaken beliefs.

So firmly were they believed in the early 1960s that IBM’s /360 and its descendants could trap on a “Significance Exception” whenever 0.0 was generated by subtracting a number from itself; the SIGMA 7 clone could trap whenever more than a programmer-chosen number of digits cancelled. For lack of a good application those traps were never enabled. Besides, the fastest way to assign X = 0.0 was to compute X = X-X in a register.

The next example is designed to disassociate “Catastrophic” from “Cancellation” in a reader’s mind. Since, to most minds, money matters more than geometry, the example is distilled from a program that computes the rate of return on investment, though the connection is not obvious.

We attempt to program the function $A(x) := (x-1)/(\exp(x-1) - 1)$ as follows:

```
Real Function Å( Real X ) ;
   Real Y, Z ;
     Y := X – 1.0 ;
     Z := EXP(Y) ;
     If  Z ≠ 1.0  then  Z := Y/(Z – 1.0) ;
     Return  Å := Z ;
  End  Å .
```

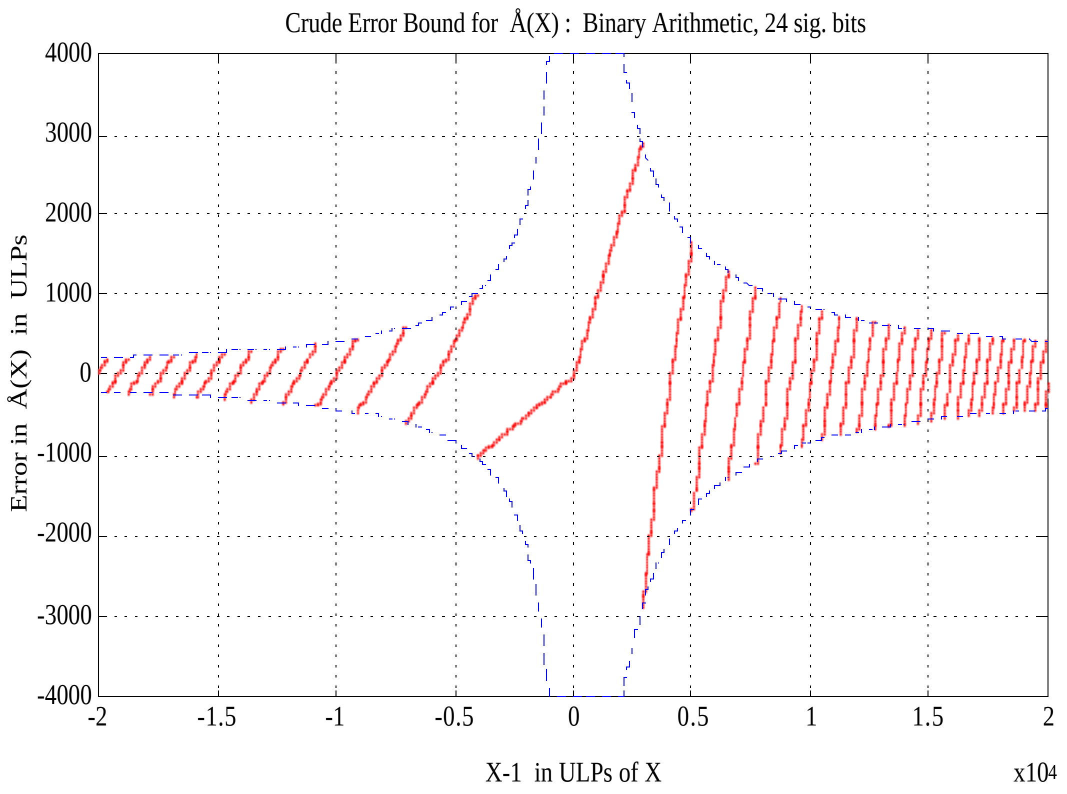
Cancellation appears to turn Å(X) into (roundoff)/(more roundoff) when X is very near 1.0 , very much as the *expression* $(x-1)/(\exp(x-1) - 1)$ for $A(x)$ approaches 0/0 as x approaches 1 . The conventional estimate of the relative error in Å is (roundoff)/$(\exp(x-1) - 1)$. Does this imply that the *function* $A(x)$ cannot be computed accurately if x is too near 1 ? No. In fact, $A(x)$ has a Taylor Series

$$A(x) = 1 - (x-1)/2 + (x-1)^2/12 - (x-1)^4/720 + (x-1)^6/30240 - (x-1)^8/1209600 + \dots \quad \text{for } |x-1| < \pi$$

that shows how well the *function* $A(x)$ behaves for x near 1 regardless of the behavior of its original *expression*. For arguments x close enough to 1 we can compute $A(x)$ as accurately as needed by using enough terms of this series. When we do so and compare this computation with the program Å(X) above, we discover that the conventional error estimate is too crude:

> Despite suggestions above that cancellation might render Å(X) = (roundoff)/(more roundoff) worthless, it never loses all accuracy. Å retains at least half the sig. digits arithmetic carries. If the arithmetic carries, say, eight sig. dec., Å(X) is always accurate to at least four. How come?

Compute Å(X) and plot its error and the conventional crude error bound in ULPs:



An ULP is one Unit in the Last Place stored.

Error: ——

Bound: - - - -

The graph above shows how nearly unimprovable conventional error bounds can be; but they still tend to ∞ as X approaches 1 , so they still suggest wrongly that Å(X) can lose all the digits carried. To dispel that suggestion we must take explicit account of the discrete character of floating-point numbers: The graph shows the worst error in Å(X) to be about $\pm 2900 \approx \pm 2^{11.5}$ ULPs at which point less than half the 24 sig. bits carried got lost, not all bits. This is no fluke; in general Å(X) is provably accurate to at least half the sig. bits carried by the arithmetic.

At first sight an obvious way to repair the inaccuracy of program Å(…) is to put the series *A*(X) into it like this:

```
Real Function Á( Real X ) ;
   Real Y, Z ;
     Y := X–1.0 ;
     If |Y| < Threshold then  Z := 1.0 – Y·(1/2 – Y·(1/12 – Y·(1/720 – Y·(1/30240 – …))))
                         else  Z := Y/( EXP(Y) – 1.0 ) ;
     Return Á := Z ;
  End Á .
```

Before this program Á(X) can be used, three messy questions need tidy answers:

What value should be assigned to " Threshold " in this program?

How many terms " …–Y·(1/30240 – …)… " of the series *A*(X) should this program retain?

How accurate is this program Á(X) ?

The answers are complicated by a speed/accuracy trade-off that varies with the arithmetic's precision.

Rather than tackle this complication, let's consider a simpler but subtle alternative:

```
Real Function Â( Real X ) ;
   Real Y, Z ;
     Y := X – 1.0 ;
     Z := EXP(Y) ;
     If Z ≠ 1.0 then Z := LN(Z)/(Z – 1.0) ;
     Return Â := Z ;
  End Â .
```

This third program Â(X) differs from the first Å(X) *only* by the introduction of a logarithm into the assignment Z := LN(Z)/(Z – 1.0) instead of Z := Y/(Z – 1.0) . This logarithm recovers the worst error, committed when EXP(Y) was rounded off, well enough to cancel almost all of it out. Â(X) runs somewhat slower than Á(X) .

This subtle program Â(X) is provably always accurate within a few ULPs unless Overflow occurs.

What general conclusions do the foregoing examples (A, Å, *A*, Á, Â) support? These three:

1. Cancellation is not a reliable indication of (lost) accuracy. Quite often a drastic departure of intermediate results (like LN(Z) above) from what would have been computed in the absence of roundoff is no harbinger of disaster to follow. Such is the case for matrix computations like inversion and eigensystems too; they can be perfectly accurate even though, at some point in the computation, no intermediate results resemble closely what would have been computed without roundoff. What matters instead is how closely a web of mathematical relationships can be maintained in the face of roundoff, and whether that web connects the program's output strongly enough to its input no matter how far the web sags in between. **Error-analysis can be very unobvious.**

2. Error-analysts do not spend most our time estimating how big some error isn't. Instead we spend time concocting devious programs, like the third Â(X) above, that cancel error or suppress it to the point where nobody cares any more. **Competent error-analysts are extremely rare.**

3. " 95% of the folks out there are completely clueless about floating-point." (J.G., 28 Feb. 1998) They certainly aren't error-analysts. They are unlikely to perceive the vulnerability to roundoff of a formula or program like the first Å(X) above until after something bad has happened, which is more likely to happen first to you who use the program than to him who wrote it. What can protect you from well-meaning but numerically inexpert programmers? **Use Double Precision**. When the naive program Å(X) is run in arithmetic twice as precise as the data X and the desired result, it cannot be harmed by roundoff. Except in extremely uncommon situations, extra-precise arithmetic generally attenuates risks due to roundoff at far less cost than the price of a competent error-analyst.

Uh-oh. The advice “ **Use Double Precision** ” contradicts an ancient Rule of Thumb, namely

“ Arithmetic should be barely more precise than the data and the desired result.”

. . .

This Rule of Thumb is wrong.

It was *never* quite right, but it’s still being built into programming languages and taught in school.

. . .

Why do so many people still believe in this wrong Rule of Thumb ?

What’s wrong with this Rule of Thumb?

How, when and why did this wrong Rule of Thumb get put into so many programming languages?

So it’s wrong. What should we be doing instead?

The next twelve pages address these questions.

“ Arithmetic should be barely more precise than the data and the desired result.”
Why do so many people still believe in this wrong Rule of Thumb ?

It is propagated with a plausible argument whose misuse of language obscures its fallacy.

The argument goes thus: “ When we try to compute c := a¤b for some arithmetic operation ¤ drawn from { +, –, ·, / }, we actually operate upon inaccurate data a+∆a and b+∆b , and therefore must compute instead c+∆c = (a+∆a)¤(b+∆b) . To store more ‘significant digits’ of c+∆c than are accurate seems surely wasteful and possibly misleading, so c+∆c might as well be rounded off to no more digits than are ‘significant’ in whichever is the bigger (for { +, – }) or less precise (for { ·, / }) of a+∆a and b+∆b . In both cases, the larger of the precisions of a+∆a and b+∆b turns out to be at least adequate for c+∆c .”

To expose the fallacy in this argument we must first cleanse some of the words in it of mud that has accreted after decades of careless use. In the same way as a valuable distinction between “disinterested” (≈ impartial) and “uninterested” (≈ indifferent) is being destroyed, misuse is destroying the distinction between “precision” and “accuracy”. For instance, Stephen Wolfram’s *Mathematica*® misuses “*Precision*” and “*Accuracy*” to mean *relative* and *absolute* accuracy or precision.. Let’s digress to refresh these words’ meanings:

“Precision” concerns the tightness of a specification; “Accuracy” concerns its correctness. An utterly inaccurate statement like “You are a louse” can be uttered quite precisely. The Hubble space-telescope’s mirror was ground extremely precisely to an inaccurate specification; that precision allowed a corrective lens, installed later by a space-walking astronaut, to compensate for the error. 3**.**177777777777777 is a rather precise (16 sig. dec) but inaccurate (2 sig. dec.) approximation to π = 3**.**141592653589793… . Although “ exp(-10) = 0.0000454 ” has 3 sig. dec. of precision it is accurate to almost 6 . Precision is to accuracy as intent is to accomplishment; a natural disinclination to distinguish them invites first shoddy science and ultimately the kinds of cynical abuses brought to mind by “ People’s Democracy,” “ Correctional Facility ” and “ Free Enterprise.”

Strictly speaking, a number can possess neither precision nor accuracy.

A number possesses only its value.

Precision attaches to the format into which the number is written or stored or rounded. Better (higher or wider) precision implies finer resolution or higher density among the numbers representable in that format. All three of

3 3.0 E0 3.0 D0

have exactly the same value though the first is written like a 2-byte `INTEGER` in Fortran or `int` in C, the second is written like a 4-byte `REAL` in Fortran or 8-byte `double` in C, and the third is written for 8-byte `DOUBLE PRECISION` in Fortran. To some eyes these numbers are written in order of increasing precision. To other eyes the integer " 3 " is exact and therefore more precise than any floating-point " 3.0 " can be. Precision (usually *Relative* precision) is commonly gauged in " significant digits " regardless of a number's significance.

Many a textbook asserts that a floating-point number represents the set of all numbers that differ from it by no more than a fraction of the difference between it and its neighbors with the same floating-point format. This figment of the author's imagination may influence programmers who read it but cannot otherwise affect computers that do not read minds. A number can represent only itself, and does that perfectly.

Accuracy connects a number to the context in which it is used. Without its context, accuracy makes no more sense than the sentence " Rosco is very tall." does before we know whether Rosco is an edifice, an elephant, a sailboat, a pygmy, a basketball player, or a boy being fitted with a new suit for his confirmation. In context, better (higher) accuracy implies smaller error. Error (usually *Absolute* error) is the difference between the number you got and the number you desired. *Relative* error is the absolute error in ln(what you got) and is often approximated by (absolute error)/(what you got) and gauged in " significant digits."

To distinguish between Precision and Accuracy is important. " The difference between the right word and the almost right word is … the difference between lightning and the lightning bug." — Mark Twain

Precision and **Accuracy** are related, indirectly, through a speed – accuracy trade-off.

Before the mid 1980s, floating-point arithmetic’s accuracy fell short of its precision on several commercially significant computers. Today only the Cray X-MP/Y-MP/…/J90 family fails to round every arithmetic operation within a fraction of an ULP, and only the IBM /360/370/390 family and its clones have non-binary floating-point not rounded within half an ULP. All other commercially significant floating-point hardware now on and under desktops rounds binary within half an ULP as required by IEEE Standard 754 unless directed otherwise. That is why we rarely have to distinguish an arithmetic operation’s accuracy from its precision nowadays. But …

Accuracy < Precision for *most* floating-point computations, not *all*.

The loss of accuracy can be severe if a problem or its data are *Ill-conditioned*, which means that the correct result is hypersensitive to tiny perturbations in its data. The term “ Ill-conditioned ” suggests that the data does not deserve an accurate result; often that sentiment is really “ sour grapes.” Data that deserve accurate results can be served badly by a naive programmer’s choice of an algorithm *numerically unstable* for that data although the program may have delivered satisfactory results for all other data upon which it was tested. Without a competent error-analysis to distinguish this numerical instability from ill-condition, inaccuracy is better blamed upon “ bad luck.” Surprisingly many numerically unstable programs, like Å(X) above, lose up to half the sig. digits carried by the arithmetic; some lose all, as if the program harbored a grudge against certain otherwise innocuous data.

Despite how most programs behave, no law limits every program’s output to less accuracy than its arithmetic’s precision. On the contrary, a program can simulate arithmetic of arbitrarily high precision and thus compute its output to arbitrarily high accuracy limited only by over/underflow thresholds, memory capacity, cleverness and time. (Learn how from papers by David Bailey, by Douglas Priest, and by Jonathan Shewchuk.) Since very high precision is slow, a programmer may substitute devious tricks to reach the same goal sooner without ever calling high-precision arithmetic subroutines. His program may become hard to read but, written in Fortran with no EQUIVALENCE statements or in Pascal with no variant records or in **C** with no `union` types or in Java with no bit-twiddling, and using integer-typed variables only to index into arrays and count repetitions, it can be written in every language to run efficiently enough on all computers commercially significant today except Crays.

It would seem then that today’s common programming languages pose no insurmountable obstacles to satisfactory floating-point accuracy; it is limited mainly by a programmer’s cleverness and time. Ay, there’s the rub. Clever programmers are rare and costly; programmers too clever by half are the bane of our industry. An unnecessary obstacle, albeit surmountable by numerical cleverness, levies unnecessary costs and risks against programs written by numerically inexpert but otherwise clever programmers. If programming languages are to evolve to curb the cost of programming (not just the cost of compilers) then, as we shall see, they should support arbitrarily high precision floating-point explicitly, and they should evaluate floating-point expressions differently than they do now. But they don’t.

Current programming languages flourish despite their numerical defects, as if the ability of a numerical expert to circumvent the defects proved that they didn’t matter. When a programmer learns one of these languages he learns also the floating-point misconceptions and faulty rules of thumb implicit in that language without ever learning much else about numerical analysis. Thus does belief persist in the misconceptions and faulty rules of thumb despite their contradiction by abundantly many counter-examples about which programmers do not learn. Å(X) above was one simple counter-example; here is another:

Let $f(x) := \mathbf{(}\ \tan(\sin(x)) - \sin(\tan(x))\ \mathbf{)}/x^7$. If $x = 0.0200000$ is accurate to 6 sig. dec., how accurately does it determine $f(x)$ and how much precision must arithmetic carry to obtain that accuracy from the given expression? This x determines $f(x) = 0.0333486813$ to about 9 sig. dec. but at least 19 must be carried to get that 9 .

The precision declared for storing a floating-point variable,
the accuracy with which its value approximates some ideal,
the precision of arithmetic performed subsequently upon it,
and the accuracy of a final result computed from that value
cannot be correlated reliably using only the rules of a programming language without error-analysis.

“ Arithmetic should be barely more precise than the data and the desired result.”

What’s wrong with this Rule of Thumb?

By themselves, numbers possess neither precision nor accuracy. In context, a number can be less accurate or (like integers) more accurate than the precision of the format in which it is stored. Anyway, to achieve results at least about as accurate as data deserve, arithmetic precision well beyond the precision of data and of many intermediate results is often the most efficient choice albeit not the choice made automatically by programming languages like Java. Ideally, arithmetic precision should be determined not *bottom-up* (solely from the operand’s precisions) but rather *top-down* from the provenance of the operands and the purposes to which the operation’s result, an operand for subsequent operations, will be put. Besides, in isolation that intermediate result’s “accuracy” is often irrelevant no matter how much less than its precision.

What matters in floating-point computation is how closely a web of mathematical relationships can be maintained in the face of roundoff, and whether that web connects the program’s output strongly enough to its input no matter how far the web sags in between. A web of relationships just adequate for reliable numerical output is no more visible to the untrained eye than is a spider’s web to a fly.

Under these circumstances, we must expect most programmers to leave the choice of every floating-point operation’s precision to a programming language rather than infer a satisfactory choice from a web invisible without an error-analysis unlikely to be attempted by most programmers.

Error-analysis is always tedious, often fruitless; without it programmers who despair of choosing precision well, but have to choose it somehow, are tempted to opt for speed because they know benchmarks offer no reward for accuracy. The speed-accuracy trade-off is so tricky we would all be better off if the choice of precision could be automated, but that would require error-analysis to be automated, which is provably impossible in general.

Why hasn't error-analysis been automated? Not for lack of trying.

The closest we can come to automated error-analysis is *Interval Arithmetic*. It is a scheme, used more in Europe than in America, that approximates every real variable not by a single floating-point number but by a pair computed to surely straddle the variable's true value. By exploiting IEEE 754's *directed roundings*, we can implement Interval Arithmetic to run no more than a few times slower than ordinary arithmetic; speed is rarely at issue. More important is that our numerical algorithms must be recast to make use of Interval Arithmetic in just the right places lest it produce awfully pessimistic error bounds. Besides, nobody wants error bounds; we desire final results known to be reliable because their errors have been proved inconsequential.

Therefore we cannot get full value from Interval Arithmetic unless it is integrated into our programming language along with arithmetic of arbitrarily high precision variable at run-time. Moreover, to help recast algorithms into forms suitable for Interval Arithmetic, we need automated algebra systems, akin to Macsyma®, Maple® or Mathematica®, capable of generating derivatives and divided differences of a program from its text.

It is a daunting investment.

Recurring attempts to invent cheaper substitutes for Interval Arithmetic have all failed in the end after enough local limited success initially to tantalize their inventors with dreams of glory.

Among these attempts are …

- Significance Arithmetic,
- Probabilistic Error-Estimates, and
- Repeated Recomputation with Ever Increasing Precision.

The next two pages describe these attempts.

Significance Arithmetic is one of those recurring attempts. It was advocated for floating-point hardware first by N. Metropolis and R. Ashenhurst in the late 1950s. The idea is to store for each number only those significant digits believed to be correct and discard the rest. For instance, " 3.140 " might be interpreted as the interval of numbers between 3.1395 and 3.1405 in the same way as some texts would have us treat all floating-point numbers. Something like that is built into *Mathematica*®. Most implementations provide a special way to store those floating-point numbers intended to represent only themselves exactly. Every implementor has to choose for each kind of arithmetic operation a rule whereby the result's number of significant digits retained is determined from the operands' numbers of significant digits stored. Some choices tend to be pessimistic; in the course of many arithmetic operations, retained sig. digits tend to dwindle faster than correct digits would for ordinary floating-point operations. Other choices tend to be optimistic; retained sig. digits tend to accrete faster than correct digits would. Some choices are pessimistic for one computations, optimistic for another. Computations can always be contrived for which digits accrete and/or dwindle at the rate of at least half a digit too much per operation. Blind faith in Significance Arithmetic is faith misplaced.

Probabilistic error-estimates have an long history of failures. The hope was that the results of a few repeated recomputations, with random roundoff-like perturbations augmenting roundoff in every arithmetic operation, would scatter to an extent indicative of their errors. Hardware to do this was first built into the IBM 7030 *Stretch* in the late 1950s. Alas, scatter far tinier than error has a surprisingly high probability when the error is gross. See "The Improbability of Probabilistic Error Analyses for Numerical Computations" in `http://http.cs.berkeley.edu/~wkahan/improber.ps` for a disparaging critique.

The futility of all such simple-minded attempts to automate error-analysis is exposed by an example contrived by Jean-Michel Muller around 1980 and modified slightly here. Given $G(y, z) := 108 - (815 - 1500/z)/y$ and initial values $x_0 := 4$ and $x_1 := 4.25$, define $x_{n+1} := G(x_n, x_{n-1})$ for $n = 1, 2, 3, \ldots$ in turn. We seek the limit L to which the sequence $\{x_n\}$ tends; $x_n \longrightarrow L$ as $n \longrightarrow +\infty$. In the absence of an analysis that finds L exactly let us compute the sequence $\{x_n\}$ until x_{N-1} differs negligibly from x_N or else until $N = 1000$, say, and then stop with x_N as our estimate of L . All fast floating-point hardware and every implementation of Significance Arithmetic or randomized arithmetic will allege $L = 100$ very convincingly. Try it! The correct limit is $L = 5$. Interval Arithmetic delivers a narrow interval around $L \approx 5$ instead of a worthless wide interval only if it carries enormous precision, rather more than $5N$ sig. bits. However, changing either $x_0 := 4$ or $x_1 := 4.25$ ever so slightly changes the true L from 5 to 100 . which may then be miscomputed if N is not huge enough.

Repeated Recomputation with Ever Increasing Precision is your best bet for removing the obscuration of roundoff from a floating-point computation. The idea is to rerun a program repeatedly, each time with the same input data but with all local and intermediate variables and all constant literals redeclared to higher precision, until successive outputs converge closely enough to overwhelm skepticism. Each repetition should ideally increase precision by a factor near $\sqrt{2}$; go from, say, 8 sig. dec. to 12 to 16 to 24 to 32 … , so after a while each repetition will cost roughly as much time as have all previous repetitions. This prescription is easier to follow in languages like Axiom®, Derive®, Macsyma®, Maple® and Mathematica®, whose mathematical libraries were designed for this purpose, than to follow in languages like Lisp, C++ and Fortran 9X that were not designed with this prescription in mind. ("Easier" does not mean "easy;" the aforementioned languages manage literal constants and mixed-precision expressions in inconvenient ways that invite mistakes.)

This prescription is impractical in Java primarily because it lacks operator overloading.

Ever increasing precision usually works, but it can be slow. And it is certainly not foolproof.
For example, for real variables x and z define three continuous real functions E, Q and H thus:
$E(z) :=$ if $z = 0$ then 1 else $(\exp(z) - 1)/z$; $Q(x) := |\,x - \sqrt{(x^2+1)}\,| - 1/(\,x + \sqrt{(x^2+1)}\,)$; $H(x) := E(\,Q(x)^2\,)$.
Then letting $x = 15.0, 16.0, 17.0, \ldots, 9999.0$ in turn compute $H(x)$ in floating-point arithmetic rounded to the same precision in all expressions. No matter how high the precision, the computation almost always delivers the same wrong $H(x) = 0$. Try it**!** In perfect arithmetic $Q(x) = 0$ instead of roundoff, so the correct $H(x) = 1$.

(This "numerical instability" can be cured by changing E(z) the way Å(X) was changed into Â(X) above.)

Conclusion: In general there is no way to automate error-analyses without which we cannot choose arithmetic precision aptly nor guarantee the correctness of floating-point results. For programmers who will not perform error-analyses we must build into programming languages the rules of thumb that choose precisions in ways that usually work and aren't too slow. But Java hasn't done that.

“ Arithmetic should be barely more precise than the data and the desired result.”

How, when and why did this wrong Rule of Thumb get put into so many programming languages?

It started in 1963. Before then IBM’s 709/7090/7094 mainframes had been delivering sums and products of SINGLE PRECISION variables into a DOUBLE PRECISION floating-point accumulator that mimicked old electro-mechanical calculators like the *Friden* designed decades earlier for statisticians and actuaries. IBM’s Fortan compilers routinely truncated this DOUBLE sum or product to SINGLE when combining it arithmetically with a SINGLE operand, but retained the registers’ DOUBLE value when combining it with a DOUBLE variable, as in scalar product accumulation DSUM = DSUM + SA(I)*SB(I) . This matched what experienced programmers had been doing in assembly language but was unobvious to other programmers. In 1963 the Fortran IV compiler released with IBSYS 13 adopted a strict bottom-up semantics that truncated sums and products of SINGLEs from DOUBLE to SINGLE immediately, thus replacing the interpretation dble(DSUM + SA(I)*SB(J)) rounded once by a twice-rounded dble(DSUM + sngl(SA(I)*SB(I))) . To obtain the older semantics now programmers had to write DSUM = DSUM + DPROD(SA(I),SB(I)) but few knew that and fewer knew why it had changed.

IBM wished to wean programmers from old 7094 habits in anticipation of its System/360’s utterly different multi-register floating-point architecture revealed in 1964. The new semantics appealed also to CDC because their CDC 6600, designed by Seymour Cray with eight SINGLE PRECISION floating-point registers almost as wide as IBM’s DOUBLE PRECISION, ran faster that way. Compiler writers liked the new simpler semantics; it helped fit fast one-pass compilers entirely into the core memories of that era, and its determination of arithmetic precision bottom-up complied with a “ context-free ” paradigm adopted by computer linguists. Although earlier computers and their languages had been designed by people who expected to use them daily, by 1963 design had fallen to computer- and language- “architects” who did not have to use their handiwork to earn their daily bread.

What is an Architect ? He designs a house for another to build and someone else to inhabit.

In 1966 delegates from IBM’s user-group SHARE heard Gene Amdahl,
architect of System/360, admit about its floating-point that …

“ If we had known then what we know now, we wouldn’t have done it that way.”

Error-analysts like Hirondo Kuki who warned about the new architectures’ impact upon floating-point were not heeded until too late. Besides, we were fully occupied developing portable numerical software to run on a now madly proliferating diversity of competing computer arithmetics; bottom-up arithmetic semantics was the least of our concerns. In 1967 some of System/360’s floating-point hardware defects were repaired, but not Fortran’s.

In 1988 ANSI **C** copied Fortran’s mistake. Before then Kernighan-Ritchie **C** had evaluated all floating-point expressions in `double` regardless of whether operands were `float`s or `double`s. This was the right thing to do albeit for the wrong reason: the old DEC PDP-11 on which **C** had first been developed a decade earlier ran faster that way. But CDC’s descendants from Cray’s 6600 and the newer CRAY machines ran much slower that way because their `double` arithmetic had to be simulated in software. Besides, their `float` was almost as wide as everyone else’s `double`, so Cray’s `double` was a luxury rarely needed. And compiler writers taught to revere “ context free ” felt more comfortable with Fortran-like bottom-up semantics. Consequently when ANSI X3J11 allowed (but did not oblige) **C** compilers to use Fortran-style bottom-up semantics instead of Kernighan-Rirchie all-double, CRAY’s **C** was not the only compiler to switch. This switch degraded some programs’ accuracy sometimes severely on some machines. Usually, severe degradation occurred only for rare seemingly random data. The cure was the insertion of `(double)` casts in a few places in a few programs, but hardly any programs were corrected that way. Vendors prefer that software users accept aberrations due to roundoff as Acts of God instead of errors induced by historically accidental language defects.

Example: Should removal of algebraically redundant parentheses correct a “ programmer’s error ” ?
(Such parentheses are usually best left in place, but here is a floating-point exception of an entirely different kind.)

A Java programmer wrote “ `C=(F-32)*(5/9)` ” instead of “ `C=(F-32)*5/9` ” to convert Fahrenheit `F` to Celsius `C` ; see comp.lang.java.help for 1997/07/02 . It could have been **C** or Fortran. Is the joke on the programmer? Or on us for perpetuating ancient blunders blindly?

“ Arithmetic should be barely more precise than the data and the desired result.”

So it’s **wrong**. What should we be doing instead?

We must institute better rules for the determination of arithmetic precision than the bottom-up rule inferred naively from a “ context free ” principle that is at best a linguistic idealization.

Ideally, floating-point precision should be determined by the programmer from an error-analysis that takes account of operands’ provenances and the purpose that each operation’s result will serve. Sometimes this ideal is achievable. Then the programmer must be able to use type-declarations, similar to those that determine the meanings of expressions involving integers, characters, arrays and other classes, to express his intent succintly without superfluous locutions (like casts) that obscure mathematical formulas. And the compiler must honor his stated intent scrupulously, taking only those liberties the programmer has licensed explicitly. Such liberties (optimizations) will be described later; they exclude “ loose numerics ” that would undermine a programmer’s control.

“ 95% of the folks out there are completely clueless about floating-point.” (J.G., 28 Feb. 1998) Error-analysis is no option for them. For them, programming languages must determine floating-point precision by default from rules of thumb that, taking both accuracy and speed into account, optimize prospects for successful use of their programs. Such rules of thumb are on the next page.

What about taking account of cost? It matters for embedded systems sold in millions, for PDAs, for clever credit cards, … that simulate floating-point in firmware to reduce hardware costs. If they perform little floating-point, its speed doesn’t matter. Otherwise they use floating-point hardware enough to justify the space it occupies on chip; signal processing is like that. We assume full hardware support for Java’s or Borneo’s floating-point.

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

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, except possibly for gargantuan arrays, declare temporary (local) variables *all* with the widest finite precision that is not too slow nor too narrow. Here “too narrow” applies only when a declared variable in a floating-point expression or assignment is 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. This is also the precision to which to round infinitely precise literal constants and integer-typed variables. Otherwise 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 explicit casts and assignments to a narrower precision must round superfluous digits away as the programmer directs.

3. Objects represented by numbers should ideally have one parsimonious representation, called “fiducial” and rounded according to rule 1, from which all other representations and attributes are computed according to rule 2. 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 from the other in `float` may render them inconsistent if the triangle is too obtuse. In general, a good fiducial representation can be hard to determine. Moreover, an object in motion may require two representations, a `float` fiducial snapshot and a moving `double`.

Example: Given the angles of declination and right ascension of two very distant stars, what angle do they subtend at the eye of an astronaut floating slowly in space not too far from the solar system?

Let D be the given declination and A the right ascension, in degrees, for one star; they satisfy $-180^\circ \le A \le 180^\circ$ and $-90^\circ \le D \le 90^\circ$. Similarly let D+d and A+a be given for the other star. Then a well-known formula for the angle V the stars subtend at the eye is

$$V = 2\cdot\arcsin\sqrt{\left(\sin^2(d/2) + \cos(D+d)\cdot\cos(D)\cdot\sin^2(a/2) \right)} \ .$$

This formula is easy to derive and serves earth-bound astronomers well because their V is usually a small angle. However, an astronaut might be interested in angles V very near 180° for which this formula loses about half the sig. digits arithmetic carries. Try it**!** The loss can occur despite that every term inside $\sqrt{(\dots)}$ is positive, so …
Don't blame cancellation; there isn't any. This formula just doesn't like V too near 180° .

Given `float` data A, a, D, d, Java's (and ANSI **C**'s) Fortran-like semantics will let `float` V be computed far less accurately than the data deserve at some future time when trigonometric functions for `float` precision are added to the `java.lang.Math` library. (Currently it has only `double`.) At that time the formula above for V (written V = (`float`)(2*`asin`…) in Java) may malfunction in subtle ways that could not show up when its program was first written and tested. How likely will this malfunction, if it occurs, be diagnosed correctly?

Once written and tested, can the program serve safely Everywhere, including outer space?

<u>It would be safe enough if Java used old-fashioned Kernighan-Ritchie **C** floating-point semantics.</u>

(A better way to compute V is from a formula fully accurate for all data, if such can be found. It does exist:
$V = 2\cdot\arctan\sqrt{\left(((TD+ta+1)\cdot td + ta)/(((td+1)\cdot ta + 1)\cdot TD + 1)\right)}$ wherein $TD = \tan^2(D+d/2)$, $td = \tan^2(d/2)$ and $ta = \tan^2(a/2)$. It's fast too. Would you have found it? Can you prove it? How much time will you need?)

What can’t be proved right
about floating-point
is very likely wrong.

Java’s treatment of Floating-Point is provably wrong-headed.

The mistakes must be corrected by Sun
lest Java’s claim to leadership be undermined
and its mission jeopardized.

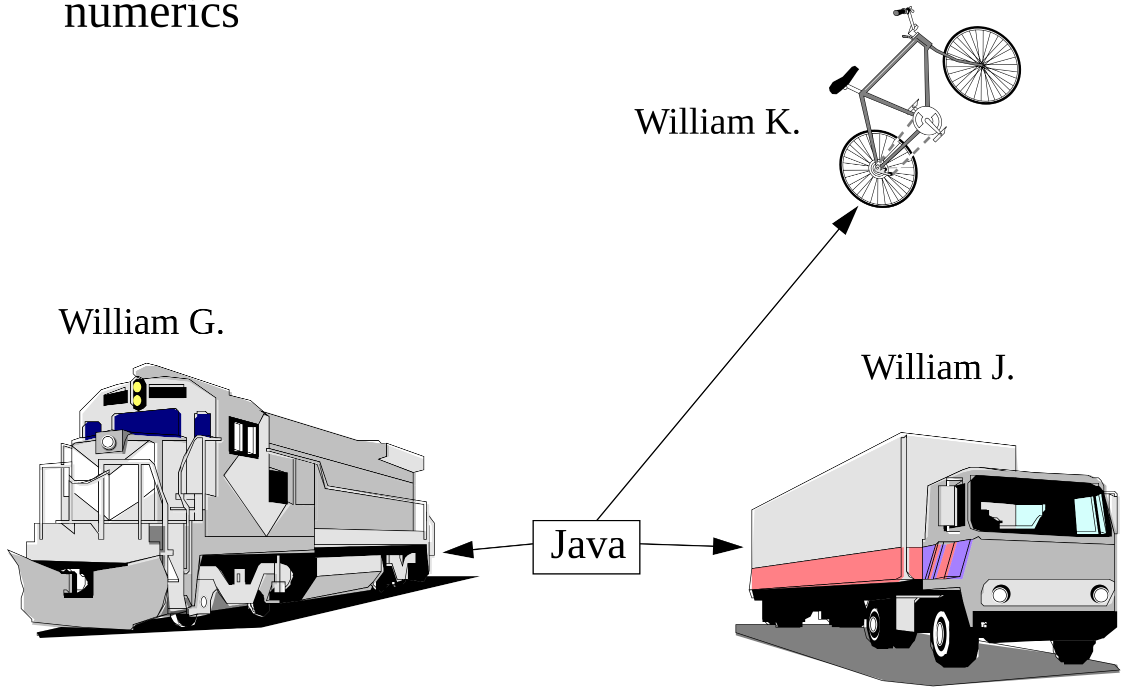
Only if 100% Pure Java™ is acknowledged to be better Java
can it compete against Microsoft’s J++ .

The first step with the least cost and biggest payoff is to

abandon Fortran-like bottom-up floating-point semantics, and

adopt Kernighan-Ritchie **C** floating-point semantics.

Three Williams contend for Java’s numerics



It is bizarre that a programming language, promulgated to Everyman to program Everything to run Everywhere, has floating-point syntax and semantics that is so disadvantageous to the overwhelming majority of programmers and users of the overwhelming majority of computers on desktops. Java’s floating-point semantics can’t be blamed upon unawareness of old-fashioned Kernighan-Ritchie **C** .

A Java Technicality

Overloaded `methods` selected according to the types, `float` or `double`, of their arguments:

Currently Java widens a `float` argument to `double` if this is the type that a `method` expects, according to its `signature`. If the selection of the `method` depends upon whether its argument is `float` or `double`, then a way to inhibit that widening must be available to a programmer who intends to select that `method`'s `float` version. Borneo has introduced a convention that Java too can adopt for the purpose: inhibit widening of an argument with an explicit `(float)` cast. For example, `tan(x)` should delivers a `double` result no matter whether x is `float` or `double`, but `tan((float)x)` should deliver a `float` result provided a suitable `tan method` is available. Thus, a programmer who gives no thought to the question gets the safer default.

The adoption of old-fashioned Kernighan-Ritchie **C** semantics for floating-point entails no change to the JVM, very little change to the Java language, and some changes in the behavior of pre-existing Java programs after they are recompiled. These last changes will almost never be significantly disadvantageous. Accuracy will almost always improve. Speed may drop 20%, most likely on Sun SPARCs. On DEC Alphas and Intel processors and their clones speed will change imperceptably, and it may increase on older Power PCs, because their register architectures favor `double`.

There is no substantial downside risk associated with Java's adoption of old-fashioned Kernighan-Ritchie **C** semantics for floating-point, and it could improve the reliability of Java's floating-point computation awesomely. Here follows an elaborate eight-page example:

Three-dimensional rectilinear geometry.

Matrix Notation for 3-Dimensional Euclidean Geometry
Lines, Planes and Cross–Products:

Let bold-faced lower-case letters $\mathbf{p}, \mathbf{q}, \mathbf{r}, \ldots, \mathbf{x}, \mathbf{y}, \mathbf{z}$ stand for real 3-dimensional column-vectors. Then row vector $\mathbf{p}^T = [p_1, p_2, p_3]$ is the transpose of column vector $\mathbf{p}$, and $\mathbf{p}^T\cdot\mathbf{q}$ is the scalar product $\mathbf{p}\bullet\mathbf{q}$ of row $\mathbf{p}^T$ and column $\mathbf{q}$. Euclidean length $\|\mathbf{p}\| = \sqrt{}(\mathbf{p}^T\cdot\mathbf{p})$.

Do not confuse the scalar $\mathbf{p}^T\cdot\mathbf{q} = \mathbf{q}^T\cdot\mathbf{p}$ with the 3–by–3 matrices ("dyads") $\mathbf{p}\cdot\mathbf{q}^T \neq \mathbf{q}\cdot\mathbf{p}^T$ nor with the vector cross-product $\mathbf{p}\times\mathbf{q} = -\mathbf{q}\times\mathbf{p}$.

As we shall see, cross-products are so important as to justify introducing a notation $\mathbf{p}^¢$, pronounced " p–cross," for a 3–by–3 skew-symmetric ($\mathbf{p}^{¢T} = -\mathbf{p}^¢$) matrix defined by the vector cross-product thus: $\mathbf{p}\times\mathbf{q} = \mathbf{p}^¢\cdot\mathbf{q}$. Explicitly the matrix $\mathbf{p}^¢$ is

$$\begin{bmatrix} 0 & -p_3 & p_2 \\ p_3 & 0 & -p_1 \\ -p_2 & p_1 & 0 \end{bmatrix}$$

The main advantage of a matrix notation for these geometrical entities is that matrix multiplication is *associative*: $\mathbf{p}^T\cdot\mathbf{q}^¢\cdot\mathbf{r} = (\mathbf{p}^T\cdot\mathbf{q}^¢)\cdot\mathbf{r} = \mathbf{p}^T\cdot(\mathbf{q}^¢\cdot\mathbf{r}) = \mathbf{p}\bullet(\mathbf{q}\times\mathbf{r})$ and $\mathbf{p}^¢\cdot\mathbf{q}^¢\cdot\mathbf{r} = (\mathbf{p}^¢\cdot\mathbf{q}^¢)\cdot\mathbf{r} = \mathbf{p}^¢\cdot(\mathbf{q}^¢\cdot\mathbf{r}) = \mathbf{p}\times(\mathbf{q}\times\mathbf{r})$ unlike scalar and cross-products; $(\mathbf{p}\bullet\mathbf{q})\cdot\mathbf{r} \neq \mathbf{p}\cdot(\mathbf{q}\bullet\mathbf{r})$ and $(\mathbf{p}\times\mathbf{q})\times\mathbf{r} \neq \mathbf{p}\times(\mathbf{q}\times\mathbf{r})$. Besides legibility, this matrix notation promotes simpler expressions, shorter proofs, and easier operator overloading in programming languages.

For Readers Reluctant to Abandon • and × Products

(Other readers can skip this page.)

We’re not abandoning familiar locutions; we’re just writing most of them shorter. Compare the Triple Product formula $(\mathbf{p}\times\mathbf{q})\times\mathbf{r} = \mathbf{q}\cdot\mathbf{p}\bullet \mathrm{r} - \mathbf{p}\cdot\mathbf{q}\bullet \mathrm{r}$ with its matrix equivalent $(\mathbf{p}^{¢}\cdot\mathbf{q})^{¢} = \mathbf{q}\cdot\mathbf{p}^{T} - \mathbf{p}\cdot\mathbf{q}^{T}$, or Jacobi’s Identity $\mathbf{p}\times(\mathbf{q}\times\mathbf{r}) + \mathbf{q}\times(\mathbf{r}\times\mathbf{p}) = -\mathbf{r}\times(\mathbf{p}\times\mathbf{q})$ with its equivalent $\mathbf{p}^{¢}\cdot\mathbf{q}^{¢} - \mathbf{q}^{¢}\cdot\mathbf{p}^{¢} = (\mathbf{p}^{¢}\cdot\mathbf{q})^{¢}$, or Lagrange’s Identity $(\mathbf{t}\times\mathbf{u})\bullet(\mathbf{v}\times\mathbf{w}) = \mathbf{t}\bullet\mathbf{v}\cdot\mathbf{u}\bullet\mathbf{w} - \mathbf{u}\bullet\mathbf{v}\cdot\mathbf{t}\bullet\mathbf{w}$ with $(\mathbf{t}^{¢}\cdot\mathbf{u})^{T}\cdot(\mathbf{v}^{¢}\cdot\mathbf{w}) = \det([\mathbf{t}, \mathbf{u}]^{T}\cdot[\mathbf{v}, \mathbf{w}])$, for succintness and ease of proof. Some things don’t change much; $\mathbf{p}\times\mathbf{q} = -\mathbf{q}\times\mathbf{p}$ becomes $\mathbf{p}^{¢}\cdot\mathbf{q} = -\mathbf{q}^{¢}\cdot\mathbf{p}$, so $\mathbf{p}^{¢}\cdot\mathbf{p} = \mathbf{o}$ (the zero vector), and $\mathbf{p}\bullet(\mathbf{q}\times\mathbf{r}) = \mathbf{p}^{T}\cdot\mathbf{q}^{¢}\cdot\mathbf{r} = \det([\mathbf{p}, \mathbf{q}, \mathbf{r}])$.

The notations’ difference becomes pronounced as problems become more complicated. For instance, given a unit vector $\mathbf{p}$ (with $\|\mathbf{p}\| = 1$) and a scalar ψ , what orthogonal matrix $\mathbf{R} = (\mathbf{R}^{T})^{-1}$ rotates Euclidean 3–space through an angle ψ radians around the axis $\mathbf{p}$? In other words, $\mathbf{R}\cdot\mathbf{x}$ is to transform a vector $\mathbf{x}$ by rotating it through an angle ψ about an axis $\mathbf{p}$ fixed through the origin $\mathbf{o}$.

An ostensibly simple formula $\mathbf{R} := \exp(\psi\cdot\mathbf{p}^{¢})$ uses the skew-symmetric cross-product matrix $\mathbf{p}^{¢}$ defined before. Here exp(…) is *not* the *array* exponential that is applied elementwise, but is the *matrix* exponential; think of $\mathbf{R} = \mathbf{R}(\psi)$ as a matrix-valued function of ψ that solves the differential equation $d\mathbf{R}/d\psi = \mathbf{p}^{¢}\cdot\mathbf{R} = \mathbf{R}\cdot\mathbf{p}^{¢}$ starting from $\mathbf{R}(0) = \mathbf{I}$, the identity matrix. Computed from $\mathbf{p}$ and ψ directly, $\mathbf{R} = \mathbf{I} + 2\cdot\big(\cos(\psi/2)\cdot\mathbf{I} + (\sin(\psi/2)\cdot\mathbf{p}^{¢})\big)\cdot(\sin(\psi/2)\cdot\mathbf{p}^{¢})$. Rewriting this expression with solely • and × products doesn’t improve it. Try it! Surely $\mathbf{R} = \exp(\psi\cdot\mathbf{p}^{¢})$ must be preferred.

Geometric Operations as Overloaded Operators in Java/Borneo

We contemplate defining classes of 3-dimensional real (`float`, `double`, and `long double` too when available) rows, columns and matrices interpreted as vectors and geometrical mappings. Java's infix operators +, –, *, / are to be overloaded to combine these geometrical objects with each other and with scalars, subject to restrictions of the kind taught in sound courses on Linear Algebra.

In Java-like Borneo programs we can write `l`(**x**) for $\|\mathbf{x}\|$ for both rows and columns **x** , and write `Trp`(**p**) for $\mathbf{p}^T$ and `Crs`(**p**) for $\mathbf{p}^{¢}$. However, to allow `Trp` and `Crs` and the like to be called without prepending a class or package name, Java/Borneo classes that use them would have to include a host of "wrapper" `static` methods like

```
    static colvector Trp(rowvector rT)  { return rT.Trp() ; } .
```

Alternatively, we can use postfix locutions like **x.**`l`( ) and **p.**`Trp`( ) and **p.**`Crs`( ) and suffer the annoyance of Java's redundant ( ) in silence as the price paid for freedom from wrappers.

We must *not* write **p*q** for the scalar product **p•q** nor the cross-product **p**×**q** lest they become non-associative invitations to blunder. Instead we write the scalar product **p•q** as `Trp`(**p**)*__q__ for $\mathbf{p}^T\cdot\mathbf{q}$, and the cross-product **p**×**q** as `Crs`(**p**)*__q__ for $\mathbf{p}^{¢}\cdot\mathbf{q}$. If you like `Dot`(**p**, **q**) and `Cross`(**p**, **q**) respectively, or **p.**`Dot`(**q**) and **p.**`Cross`(**q**) , use them instead; but we avoid them because our mathematical matrix notation from the previous two pages transliterates so immediately to our Java-like notation with overloaded operators described on this page, and *vice-versa*.

In what follows, computational solutions to several common geometrical problems are presented in our mathematical matrix notation because it is slightly easier to read than Java could ever be.

Applications of Cross-Products to Nearest-Point Problems

Cross-products $\mathbf{p}\times\mathbf{q}$, or $\mathbf{p}^{¢}\cdot\mathbf{q}$ in our matrix notation, figure prominently instead of determinants to provide neat textbook solutions of many commonplace geometrical problems. For example, given the equations $\mathbf{p}^T\cdot\mathbf{x} = \pi$, $\mathbf{b}^T\cdot\mathbf{x} = ß$, $\mathbf{w}^T\cdot\mathbf{x} = \Omega$ of three planes, their point of intersection is

$$\mathbf{z} = (\pi\cdot\mathbf{b}^{¢}\cdot\mathbf{w} + ß\cdot\mathbf{w}^{¢}\cdot\mathbf{p} + \Omega\cdot\mathbf{p}^{¢}\cdot\mathbf{b})/(\mathbf{p}^T\cdot\mathbf{b}^{¢}\cdot\mathbf{w}) \ .$$

Neat formulas are more memorable and therefore more likely to be used by programmers than are ugly numerical algorithms like Gaussian Elimination even if the latter are numerically more stable. Gaussian Elimination is also faster than the foregoing formula, but a programmer can easily fix that by rewriting $\mathbf{z} = ((\mathbf{b}^{¢}\cdot\mathbf{w})\cdot\pi + \mathbf{p}^{¢}\cdot(\mathbf{b}\cdot\Omega - \mathbf{w}\cdot ß))/(\mathbf{p}^T\cdot(\mathbf{b}^{¢}\cdot\mathbf{w}))$ and reusing a common subexpression. Still, this not so stable numerically as Gaussian Elimination with pivotal exchanges.

Like Beauty, the neatness of a formula and often its speed lie more easily in the eye of the beholding programmer than does numerical stability. Textbook formulas don't show off roundoff. The reader will not easily determine which are numerically unstable among the next page's neat solutions for seven commonplace geometrical problems each of the following Nearest-Point kind:

Given a point **y** and specifications for a geometrical object **G** , we seek a point **z** in **G** nearest **y** .

We expect the line segment joining **y** and **z** to stick out of **G** perpendicularly. If two formulas for **z** are offered below they suffer differently from rounding errors; the first formula suffers less than the second whenever $\|\mathbf{z}-\mathbf{y}\| \ll \|\mathbf{y}\|$ and the second less than the first whenever $\|\mathbf{z}\| \ll \|\mathbf{y}\|$. Unless parentheses indicate otherwise, associative products $\mathbf{A}\cdot\mathbf{B}\cdot\mathbf{C}$ should be evaluated in whichever order, $(\mathbf{A}\cdot\mathbf{B})\cdot\mathbf{C}$ or $\mathbf{A}\cdot(\mathbf{B}\cdot\mathbf{C})$, requires fewer arithmetic operations; doing so below diminishes roundoff too.

1. Given the equation $\mathbf{p}^{T}\cdot\mathbf{x} = \pi$ of a plane Π, the point $\mathbf{z}$ in Π nearest $\mathbf{y}$ is

$$\mathbf{z} = \mathbf{y} - \mathbf{p}\cdot(\mathbf{p}^{T}\cdot\mathbf{y} - \pi)/\|\mathbf{p}\|^{2} \;=\; (\,\mathbf{p}\cdot\pi - \mathbf{p}^{¢}\cdot\mathbf{p}^{¢}\cdot\mathbf{y}\,)/\|\mathbf{p}\|^{2} \;.$$

2. Given three points $\mathbf{u}$, $\mathbf{u}+\mathbf{v}$ and $\mathbf{u}+\mathbf{w}$ through which one plane Π passes, the point $\mathbf{z}$ in Π nearest $\mathbf{y}$ is $\mathbf{z} = \mathbf{y} - \mathbf{p}\cdot\mathbf{p}^{T}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2} = \mathbf{u} - \mathbf{p}^{¢}\cdot\mathbf{p}^{¢}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2}$ wherein $\mathbf{p} = \mathbf{v}^{¢}\cdot\mathbf{w}$.

3. Given three points $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ through which one plane Π passes, the point $\mathbf{z}$ in Π nearest $\mathbf{y}$ is $\mathbf{z} = \mathbf{y} - \mathbf{p}\cdot\mathbf{p}^{T}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2} = \mathbf{u} - \mathbf{p}^{¢}\cdot\mathbf{p}^{¢}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2}$ wherein $\mathbf{p} = (\mathbf{v} - \mathbf{u})^{¢}\cdot(\mathbf{w} - \mathbf{u})$. The order of $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ is permutable in each formula separately. To diminish roundoff in $\mathbf{p}$ choose $\mathbf{u}$ to maximize $\|\mathbf{v} - \mathbf{w}\|$; in $\mathbf{z}$ choose $\mathbf{u}$ to minimize $\|\mathbf{y} - \mathbf{u}\|$ in the first formula, $\|\mathbf{u}\|$ in the second.

4. Given two points $\mathbf{u}$ and $\mathbf{u}+\mathbf{v}$ through which one line **£** passes, the point $\mathbf{z}$ in **£** nearest $\mathbf{y}$ is

$$\mathbf{z} = \mathbf{y} + \mathbf{v}^{¢}\cdot\mathbf{v}^{¢}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{v}\|^{2} \;=\; (\,\mathbf{v}\cdot\mathbf{v}^{T}\cdot\mathbf{y} - \mathbf{v}^{¢}\cdot\mathbf{v}^{¢}\cdot\mathbf{u}\,)/\|\mathbf{v}\|^{2} \;=\; \mathbf{u} + \mathbf{v}\cdot\mathbf{v}^{T}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{v}\|^{2} \;.$$

5. Given two points $\mathbf{u}$ and $\mathbf{u}+\mathbf{v}$ through which one line **£** passes, and two points $\mathbf{y}$ and $\mathbf{y}+\mathbf{w}$ through which another line **¥** passes, the point nearest **£** in **¥** is $\mathbf{x} = \mathbf{y} - \mathbf{w}\cdot\mathbf{p}^{T}\cdot\mathbf{v}^{¢}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2}$ wherein $\mathbf{p} = \mathbf{v}^{¢}\cdot\mathbf{w}$. Nearest **¥** in **£** is $\mathbf{z} = \mathbf{x} - \mathbf{p}\cdot\mathbf{p}^{T}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2} = \mathbf{u} - \mathbf{v}\cdot\mathbf{p}^{T}\cdot\mathbf{w}^{¢}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{p}\|^{2}$.

6. Given two points $\mathbf{u}$ and $\mathbf{w}$ through which one line **£** passes, the point $\mathbf{z}$ in **£** nearest $\mathbf{y}$ is $\mathbf{z} = \mathbf{y} + \mathbf{v}^{¢}\cdot\mathbf{v}^{¢}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{v}\|^{2} = (\,\mathbf{v}\cdot\mathbf{v}^{T}\cdot\mathbf{y} - \mathbf{v}^{¢}\cdot\mathbf{v}^{¢}\cdot\mathbf{u}\,)/\|\mathbf{v}\|^{2} = \mathbf{u} + \mathbf{v}\cdot\mathbf{v}^{T}\cdot(\mathbf{y} - \mathbf{u})/\|\mathbf{v}\|^{2}$ wherein $\mathbf{v} = \mathbf{w} - \mathbf{u}$. Since $\mathbf{u}$ and $\mathbf{w}$ are permutable, choose $\mathbf{u}$ to minimize $\|\mathbf{y} - \mathbf{u}\|$ in the first and last formulas, and to minimize $\|\mathbf{u}\|$ in the middle formula, which is best if $\|\mathbf{z}\| << \|\mathbf{u}\|$ too.

7. Given the two equations $\mathbf{p}^{T}\cdot\mathbf{x} = \pi$ and $\mathbf{b}^{T}\cdot\mathbf{x} = ß$ of a line **£** , the point $\mathbf{z}$ in **£** nearest $\mathbf{y}$ is

$$\mathbf{z} = \mathbf{y} + \mathbf{v}^{¢}\cdot(\,\mathbf{p}\cdot(ß - \mathbf{b}^{T}\cdot\mathbf{y}) - \mathbf{b}\cdot(\pi - \mathbf{p}^{T}\cdot\mathbf{y})\,)/\|\mathbf{v}\|^{2} \;=\; (\,\mathbf{v}\cdot\mathbf{v}^{T}\cdot\mathbf{y} + \mathbf{v}^{¢}\cdot(\mathbf{p}\cdot ß - \mathbf{b}\cdot\pi)\,)/\|\mathbf{v}\|^{2} \quad \text{wherein} \quad \mathbf{v} = \mathbf{p}^{¢}\cdot\mathbf{b} \;.$$

We have just seen seven neat solutions for commonplace geometrical problems that

Java’s floating-point expression-evaluation turns into

Numerical Junk.

HOW ? WHY ?

Java gets us into trouble that old-fashioned Kernighan-Ritchie **C** avoided by rounding everything by default to `double` unless an explicit cast specified otherwise.

Java gets us into trouble because it rounds all subexpressions involving exclusively `float` operands to `float` precision.

Let’s see how it happens: …

HOW ? An example shows how Java-like floating-point malfunctions:

7. Given the two equations $\mathbf{p}^T \cdot \mathbf{x} = \pi$ and $\mathbf{b}^T \cdot \mathbf{x} = ß$ of a line £ , the point $\mathbf{z}$ in £ nearest $\mathbf{y}$ is
$\mathbf{z} = \mathbf{y} + \mathbf{v}^{¢} \cdot (\mathbf{p} \cdot (ß - \mathbf{b}^T \cdot \mathbf{y}) - \mathbf{b} \cdot (\pi - \mathbf{p}^T \cdot \mathbf{y}))/\|\mathbf{v}\|^2 = (\mathbf{v} \cdot \mathbf{v}^T \cdot \mathbf{y} + \mathbf{v}^{¢} \cdot (\mathbf{p} \cdot ß - \mathbf{b} \cdot \pi))/\|\mathbf{v}\|^2$ wherein $\mathbf{v} = \mathbf{p}^{¢} \cdot \mathbf{b}$.

Try 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 will cause trouble because it defines £ as the intersection of two nearly parallel planes, so tiny changes in data can change $\mathbf{z}$ drastically.

When all arithmetic is performed naively in `float` the two formulas above for $\mathbf{z}$ yield respectively $\mathbf{z}_1^T = [\ 1,\ 1,\ -1\ ]$ and $\mathbf{z}_2^T = [\ 1.000000054,\ 1.000000054,\ -1.500000148\ ]$ instead of the correct $\mathbf{z}^T = [\ 1/3,\ 2/3,\ -4/3\ ]$ which is computed correctly rounded when all intermediate results (subexpressions) are evaluated in `double` before $\mathbf{z}$ is rounded back to `float` .

Naively computed $\mathbf{z}_1$ and $\mathbf{z}_2$ are not so far from $\mathbf{z}$ as to be obviously wrong if $\mathbf{z}$ were unknown, and yet too far away to be acceptable for most purposes. Worst of all, the distances from both planes that intersect in £ to $\mathbf{z}_1$ is about 0.81 , to $\mathbf{z}_2$ about 0.65 , so neither $\mathbf{z}_1$ nor $\mathbf{z}_2$ can be correct solutions for problem **7** with slightly different data. *The naive results are geometrically impossible*.

Computed entirely in `float` arithmetic upon `float` data, every neat solution to problems **1 - 7** is numerically unstable. Skilled numerical analysts can reformulate them as constrained least-squares problems and solve them to acceptable accuracy using only `float` arithmetic, but not so quickly nor so accurately as `double` works above. The neat solutions are fine if computed extra-precisely.

WHY ? Bilinear forms vulnerable to roundoff followed by cancellation occur frequently:

Scalar products: $\mathbf{p}\bullet\mathbf{b} = \mathbf{p}^{\mathrm{T}}\cdot\mathbf{b} = p_1 \cdot b_1 + p_2 \cdot b_2 + p_3 \cdot b_3$.

Linear combinations: $$\mathbf{p}\cdot ß - \mathbf{b}\cdot\pi = \begin{bmatrix} p_1 \cdot \beta - b_1 \cdot \pi \\ p_2 \cdot \beta - b_2 \cdot \pi \\ p_3 \cdot \beta - b_3 \cdot \pi \end{bmatrix} .$$

Cross products: $$\mathbf{p}\times\mathbf{b} = \mathbf{p}^{¢}\cdot\mathbf{b} = \begin{bmatrix} p_2 \cdot b_3 - p_3 \cdot b_2 \\ p_3 \cdot b_1 - p_1 \cdot b_3 \\ p_1 \cdot b_2 - p_2 \cdot b_1 \end{bmatrix} .$$

These entities are *geometrically redundant*; they are so correlated that $(\mathbf{p}\cdot ß - \mathbf{b}\cdot\pi)\bullet(\mathbf{p}\times\mathbf{b}) = 0$ for *all* data $\{\mathbf{p}, \pi, \mathbf{b}, ß\}$. Even if data are “accurate” to few sig. digits and computed entities to fewer, their geometrical redundancy must be conserved as accurately as possible. We can tolerate slightly inaccurate results interpretable as realizable geometrical objects slightly different from our original intent, but not geometrically impossible objects like a $\mathbf{p}\times\mathbf{b}$ too far from orthogonal to $\mathbf{p}$ and $\mathbf{b}$.

Therefore these bilinear forms must be computed carrying somewhat more precision than in the data, thereby preserving geometrical redundancy despite “losses” of several digits to cancellation. At any precision, prolonged chains of computation risk losing geometrical redundancy. The wider is the precision, the longer is that loss postponed and the more often prevented, provided that extra-precise arithmetic does not run intolerably slowly. And extra precision usually costs less than error-analysis.

Dynamic Directed Rounding Modes

All hardware conforming to IEEE Standard 754 for Binary Floating-Point Arithmetic must (and all do) afford the programmer a way to specify *Dynamically* one of four *Rounding Modes*:

Round to Nearest (the default), and three *Directed* Rounding Modes —

Round towards 0 (truncate), Round Up (towards +∞), Round Down (towards –∞) .

That these modes are Dynamic means they are selected by setting two bits in a control word in the floating-point hardware. Programmers should regard a rounding mode as a global variable implicitly influencing every floating-point operation not protected from that influence by a *Static* assignment of its rounding mode. For these static assignments the DEC Alpha provides two bits in the op-code of every floating-point operation to achieve the same effect as other machines accomplish by saving, setting and some time later restoring the control word’s bits.

Alpha’s and Java’s designers seem to have had none but arcane uses, like Interval Arithmetic and error-bounding, in mind for the directed rounding modes. That may be why Java forbade them. That mind-set is almost right. Most programmers, and all programmers most the time, have no use for directed rounding modes. Consequently almost all programs include no mention of them and should be compiled to get the default rounding mode from the control word’s two bits. However …

“Almost all true is entirely a lie.” — a Yiddish folk-saying.

Many a programmer will encounter a compelling reason to run the same subprogram four times, each time choosing a different rounding mode to govern the way all but the statically rounded arithmetic operations in the subprogram are rounded, and then compare the subprogram’s four outputs. Why?

Numerical Instability.

It may be suspected as the cause of dubious output from a program comprising several subprograms of diverse provenances, pedigrees and perspicuities. How is numerical instability to be debugged?

Debugging Numerical Instability:

It is difficult even for experts. We try first to blame the instability upon one subprogram. If this can be done, and if the subprogram is our own, we hope we can fix it; and if the subprogram came from someone else we hope *he* can fix it. Either way, let’s start by trying to determine whom to blame.

Into our program comprising several subprograms let us insert two kinds of modifications:

- Display, file or print intermediate values put out from some subprograms into others.
- Rerun some or all subprograms in all four rounding modes and compare intermediate values.

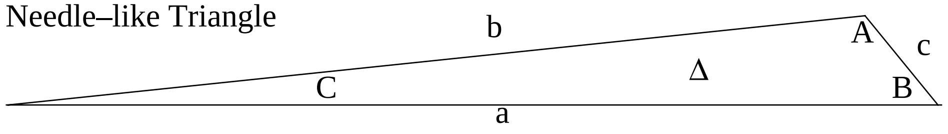
Doubts fall first upon the first subprogram(s), if any, whose outputs vary much more than expected.

Of course this scheme can’t be foolproof since error-analysis can’t be automated in general; see pp. 38-39 and 41 for three examples and p. 35 for perhaps another that defy this scheme. And after the scheme casts suspicion upon a subprogram we must analyze it and rule out other causes before we condemn it as unstable. Among other reasons for violent roundoff-induced fluctuations in a subprogram’s output, and ways to cope with them, are …

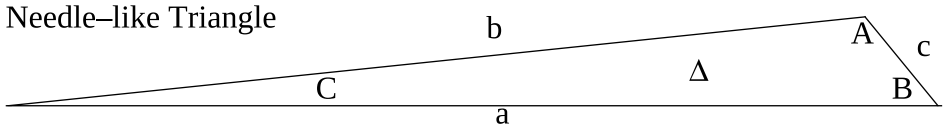
- The function computed accurately by the subprogram has a singularity so near the data that it amplifies tiny changes in data into violent fluctuations of the function; therefore don’t let its data change.
- The fluctuations don’t matter so long as they conform to some constraint; try to find it and determine some measures of departure from that constraint to display/file/print instead. It’s easier said than done.
- By design, the subprogram malfunctions under any but the default mode, so don’t change that. (Rare.)

Despite these *caveat*s, reruns with directed roundings focus attention where it belongs far more often than not. This scheme works for examples on pp. 27, 44, 51 - 55, and the next several examples:

Needle–like Triangle



Example: 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.

In the 1950s and 1960s computer programmers rearranged his formula to stabilize it as follows: First sort a, b, c so that $a \ge b \ge c$; 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

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

Don't remove parentheses from this formula! It can't give rise to $\sqrt{(<0)}$. It works on all but Cray's computers. Nowadays only error-analysts and a few programmers know this stable formula though it is explained on p. 153 in *Floating-Point Computation* by P. Sterbenz (1973) Prentice-Hall, and in " Miscalculating Area and Angles of a Needle-like Triangle " `http://http.cs.berkeley.edu/~wkahan/Triangle.ps` , and elsewhere.

Let's compare both formulas on two nearby needle-like triangles, and compare also the effects of the different Directed Rounding Modes mandated by IEEE 754 but forbidden to us by Java. Since all data are `float`s, we also compare the effect of Java's all-`float` arithmetic semantics with that of Kernighan-Ritchie **C** all-`double` arithmetic upon evaluations of Heron's unstable formula.

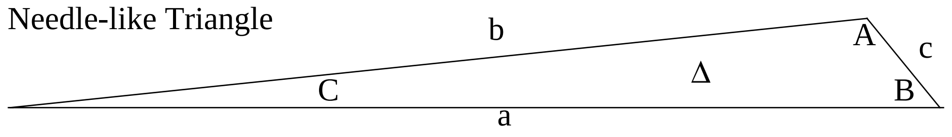
The 1st triangle's $a = 12345679.$, $b = 12345678.$, $c = 1.01233995$, Condition no. $\approx$ 500000000**.**
The 2nd triangle's $a = 12345679.$, $b = 1234567\mathbf{9}.$, $c = 1.01233995$, Condition no. = 2 .
Infinitesimal relative perturbations in the data get amplified by the Condition number when they are transmitted to Δ . The 1st triangle is ill-conditioned; the 2nd is well-conditioned and deserves an accurately computed Δ .

Table: Sensitivity to Rounding of two different formulas to calculate the Area Δ of a Triangle from the Lengths of its Sides
(calculations performed upon 4-byte `float` data).

Rounding mode	Heron’s Formula $s = ((a+b)+c)/2$ $\sqrt{s\cdot(s-a)\cdot(s-b)\cdot(s-c)}$ (unstable in `float`)	Better Formula $\frac{\sqrt{(a+(b+c))\cdot(c-(a-b))\cdot(c+(a-b))\cdot(a+(b-c))}}{4}$ (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=1234567**9** > 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

Note that only incorrect results change drastically when the rounding mode changes, and that old-fashioned Kernighan-Ritchie **C** gets fine results from an “unstable” formula.

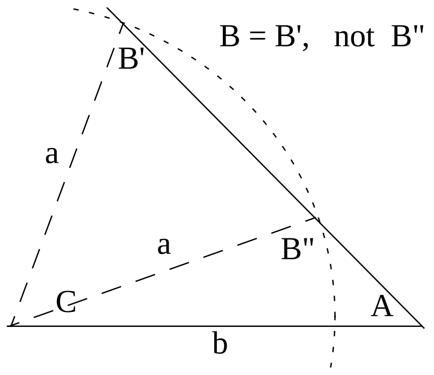
Heron’s formula is one of many schoolbook trigonometric formulas that dislike certain triangles:



$C = \arccos((a^2 + b^2 - c^2)/(2 \cdot a \cdot b)) = 2 \cdot \arctan(\sqrt{((s-a) \cdot (s-b)/(s \cdot (s-c)))})$

$c = \sqrt{(a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos C)}$

$B = \arcsin((b/a) \cdot \sin A)$ … This formula may deliver the smaller of two angles B’ and B” , and dislikes triangles with B too near 90˚ .



For better formulas than these classical formulas see
“ Miscalculating Area and Angles of a Needle-like Triangle “
`http://http.cs.berkeley.edu/~wkahan/Triangle.ps`

These classical formulas have *withstood* the Test of Time, not *passed* it.

Their unnecessary inaccuracies could be detected with the aid of directed roundings., but …

What is a Java programmer to do? With `float` data, he runs some risk that Java’s floating-point will get wretched results from a program that delivered fine results under old-fashioned Kernighan-Ritchie **C** . His hardware includes the tools he most needs to debug wretched results but Java denies him their use. Maybe better formulas lurk in places like “ Miscalculating … Triangle ” cited above, but what are his chances of finding them? What are our chances if he doesn’t, and we use his code?

Java's designers blundered if they deemed features of IEEE Standard 754 for Binary Floating-Point Arithmetic that they did not appreciate to be features usable by none but numerical experts.

The facts are quite the opposite.

In 1977 those features were designed into the Intel 8087 to serve the widest possible market, Java's market — Everybody Everywhere. A few years later similar features and more were built into the Motorola 68881/2 to go with the 68020 and 68030, and live on in the 68040 and 88110, but they are fading from the marketplace. Today Intel's floating-point architecture, now borne by Pentiums and their clones, is the most nearly ubiquitous of all architectures., And yet one of its numerically most valuable features continues to be under-utilized for lack of linguistic support. That feature is …

IEEE 754 double-extended precision, also known as `long double`.

This format occupies 10 bytes, carries 64 sig. bits of precision and 15 bits of exponent range. The Motorola chips stored it in 12 bytes, allocating two for future expansion explicitly foretold by IEEE 754. But the programming language community appears not to understand how nor why this format is intended to be used.

How: `long double` is intended to support `double` and `float` the way `double` supports `float` in Kernighan-Ritchie **C** .

Why: Extra-precise arithmetic attenuates the risk of chagrin due to roundoff. This risk is impossible to estimate well enough to determine insurance premiums; it is usually too small for most of us to notice, too big for all of us to ignore.

How much does extra precision attenuate the risk of chagrin due to roundoff?

Consider some algorithm that has been programmed to solve a problem for all input data, except perhaps a set of measure zero in data-space, and that would achieve this goal if the program were executed with infinite precision at infinite speed. For example, Gaussian Elimination with pivotal exchanges would solve all square systems of linear equations except those whose determinant vanishes, which happens on a surface in the space of all square matrices of any particular dimension.

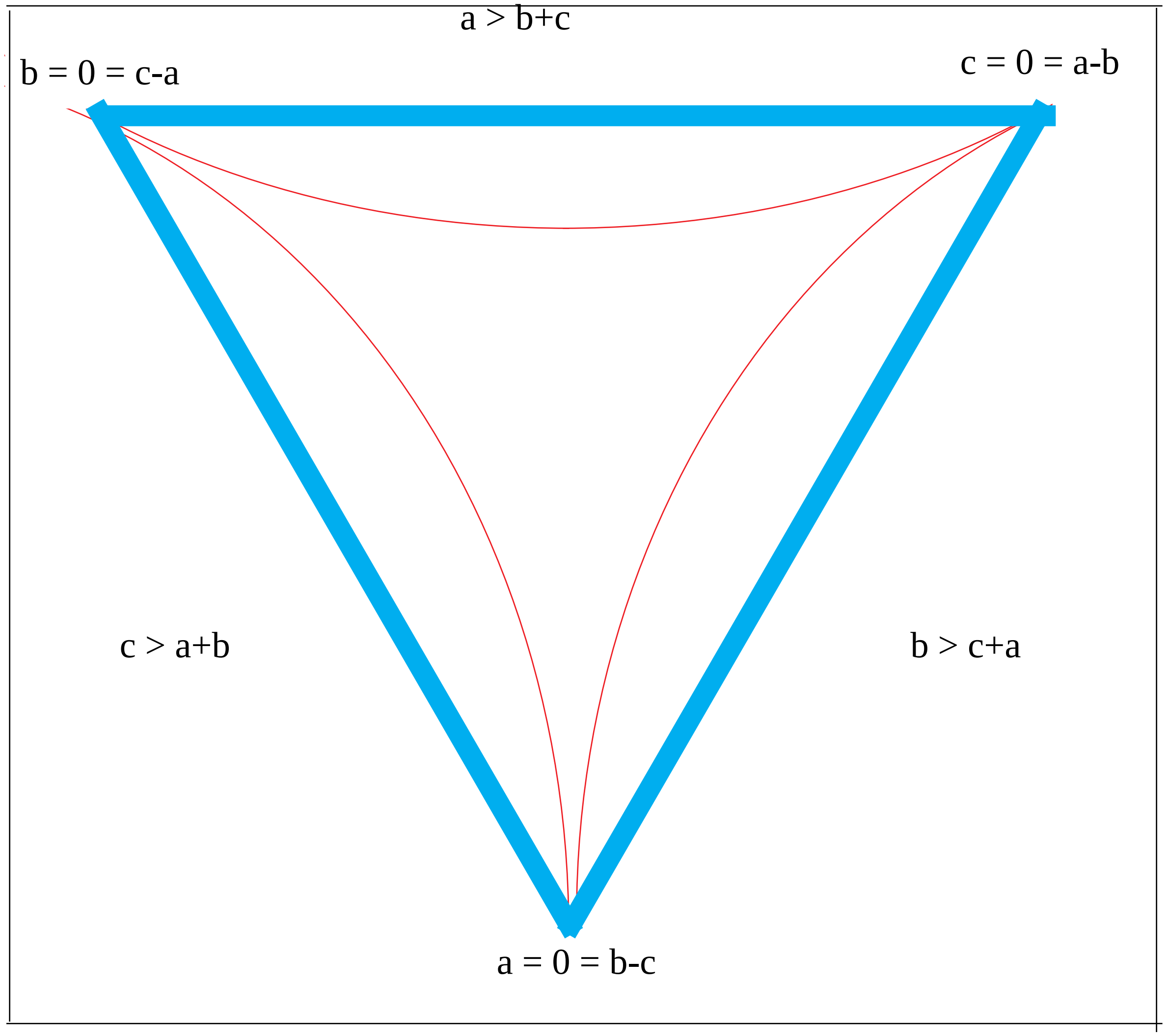
Because we compute with finite precision, there is a population of data sets for which the problem has a solution but the program computes it too inaccurately, whence arises chagrin due to roundoff.

For data of any precision fixed in advance, increasing the precision of the program’s arithmetic shrinks the population of data whence arises chagrin. The rate of shrinkage depends upon the algorithm under consideration. Typically, that population shrinks by about 1/2 for every extra bit of arithmetic precision carried until a Law of Diminishing Returns set in. Typically, carrying 11 extra bits of arithmetic precision shrinks the risk of chagrin by a factor smaller than 0.0005 , enough to change a program’s or a computer’s perceived reliability from Bad to Good.

(Atypical algorithms exist for which the rate of shrinkage is different, better like 1/4 per extra bit for some, worse like $1/\sqrt{2}$ per extra bit for those that lose half the bits carried, …, no shrinkage at all for a contrived few.)

For instance, 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, these shapes whence comes chagrin lie in a narrow ribbon:

Map Triangles to Points in the Plane by taking side–lengths (a, b, c) as Barycentric Coordinates:



Every point (a,b,c) in this triangle represents a family of Similar triangles.

Every triangle Similar to a given generic triangle maps to six of these points.

Points near the boundary (thickened here) represent Needle-like triangles.

Points between the center and the hyperbolic arcs represent triangles with all angles acute and utterly well-conditioned areas $\Delta(a,b,c)$.

But $\Delta(a,b,c)$ is computed by Heron’s formula inaccurately at points in the thickened boundary layer regardless of how well- or ill-conditioned Δ may be at that point.

When Heron’s formula is computed, every extra sig. bit of arithmetic precision carried halves the width of the boundary layer thus halving the population of triangles whose areas are computed too inaccurately.

The revised formula for Δ with sorted a, b, c is accurate at all triangles. Everybody should use it.

But they won’t.

The better formula has been published at least four times but not where most programmers who might need it are likely to look it up. Heron’s formula is what they will almost surely find insteadin their school books.

In general, programmers who use a little (or a lot of) floating-point arithmetic may be very clever at things they care about, but not at error-analysis of floating-point. (Not even the great John von Neumann got it quite right.) And all of us shall occasionally run their programs unwittingly, and be thus exposed to risks of which they were unaware. Extra-precise arithmetic, if not too slow, is the easiest way to attenuate those risks in practically *all* computations, not just the examples presented in this document, and to solve numerous other problems too. …

Extra Precision as a Way to Conserve Interest Rates’ Monotonicity

A little-known requirement for certain financial computations of Rates of Return is *Monotonicity.* This means that if a small change in data causes a computed result to change, its change should not go in the wrong direction. For instance, if the return on an investment is increased, its computed rate of return must not decrease; if the repayments on a loan are diminished, its computed interest rate must not increase. The conservation of monotonicity becomes more challenging as it becomes more important during computations designed to optimize rates of return. These rates satisfy equations that can have more than one root, and then the choice of the right root can be spoiled if monotonicity is lost to roundoff. Roundoff affects an equation’s solution both during the equation’s computation and in the accuracy criterion that stops the equation-solving iteration. To prevent changes in results from becoming artifacts of roundoff instead of consequences of changed data, equations must be solved more accurately than might naively have been thought adequate. Experience indicates 11 extra bits suffice here.

By far the easiest way to conserve monotonicity is to compute extra-precisely, carrying enough extra precision (say 11 bits) to keep roundoff’s effect utterly negligible compared with the effect of end-figure perturbations in data, provided of course that extra-precise computation does not run too slow.

The floating–point arithmetics on AMD/Cyrix/Intel chips in PCs, and on Motorola chips in old[†] Macintoshes and Sun IIIs, were designed to attenuate the risks you face and to help you diagnose them. They were designed to evaluate at full speed every subexpression to 10+-byte Extended Precision thereby attenuating the incidence of dangerously inaccurate results by orders of magnitude. DEC’s Alpha chip, and PowerPC chips used on current Power Macs and IBM RS/6000s, were designed to evaluate at full speed every subexpression to 8-byte Double-Precision, like old-fashioned K-R **C** , to somewhat attenuate the risks you face. All these chips were designed to help you debug inaccuracy by rerunning subprograms, whose source-code you can’t or won’t change, unchanged but in different rounding modes upon data that produce suspicious results. Attenuating risks does not eliminate them nor does the foregoing diagnostic technique work every time. Still, …

these hardware designs do improve your chances. But not with Java programs.

Speed Above All Else

Forced to choose between speed and safety, most people choose speed. This is the only conclusion consistent with what happens on our highways. Even people who distrust Our Government (for no apparent reason) trust the accuracy of computer arithmetic, so they too choose speed above all else. Knowing what most programmers will do, those of us who design computer systems have to design them in ways that enhance rather than detract from the programmer’s prospects for success lest his failure turn into our failure. Therefore prudence, if not due diligence, obliges programming language implementors to evaluate all floating-point expressions by default in the widest precision that does not run too slow, unless the programmer has gone to some trouble to demand otherwise.

[†]Footnote: *Post hoc, ergo propter hoc.* (What occurred must have been caused by whatever just preceded it.) The decline of Apple Computers dates from their abandonment of the superior floating-point architecture of the Motorola 680x0 processor in favor of the faster but numerically inferior Power PC. Intel’s, the only floating-point left that offers that superiority, is ubiquitous. Coincidence?

How many floating-point formats run fast on most desktop hardware today?

Three :

IEEE 754 Single	IEEE 754 Double	IEEE 754 Double-Extended
4-byte `float`	8-byte `double`	10+-byte `long double` .
24 sig. bits	53 sig. bits	64+ sig. bits
7-bit exponent	11-bit exponent	15+-bit exponent

Over 95% of the hardware on desktops support all three as recommended by IEEE 754, though the 10+-byte format may be stored in 10, 12 or 16 bytes in memory to avert word-alignment penalties.

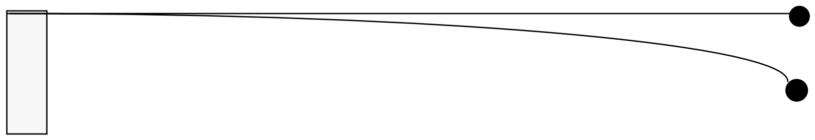
Java, like Microsoft, forbids the majority of us that have the 10+-byte format from using it.
We paid for it but we can’t benefit from it.

Some computers have set aside room in their instruction-sets for currently unimplemented 16-byte `Quadruple Precision` floating-point. Too slow to use much now, it will run practically as fast as `float` and `double` some day when it is implemented on-chip like them. It will invade a lot of chip area, so we are trying to postpone its arrival by devising adequately fast and accurate numerical algorithms that use tricks instead of `Quadruple`. Its day will come anyway. And then programs that use the 10+-byte `long double` format properly will, after recompilation to use `Quadruple` instead, continue to work at least as well as they ever did. Meanwhile, a few compilers support slow software-simulated `Quadruple`; and a few support a variety of not-so-slow 16-byte Doubled-`double` formats that are rounded in ways too perverse to qualify as IEEE 754 Double-Extended formats.

Linguistic support for three floating-point types, the third somewhat variable, instead of just two will impose a substantial burden upon compiler writers. Do any applications benefit enough from extra precision to pay for it? Yes; elastic deformations of thin sheets. Here is an oversimplified example.

Cantilever Calculation

A uniform steel spar is clamped horizontal at one end and loaded with a mass at the other. How far does the spar bend under load?



The calculation is discretized: For some integer N large enough (typically in the thousands) we compute approximate deflections { $x_0 = 0$, x_1, x_2, x_3, ..., x_{N-1}, $x_N \approx$ deflection at tip} at uniformly spaced stations along the spar. Discretization errors, the differences between these approximations and true deflections, tend to 0 like $1/N^2$. These x_j 's are the components of a column vector **x** that satisfies a system **A**·**x** = **b** of linear equations in which column vector **b** represents the load (the mass at the end plus the spar’s own weight) and the matrix **A** looks like this for N = 10 :

$$A = \begin{bmatrix} 9 & -4 & 1 & o & o & o & o & o & o & o \\ -4 & 6 & -4 & 1 & o & o & o & o & o & o \\ 1 & -4 & 6 & -4 & 1 & o & o & o & o & o \\ o & 1 & -4 & 6 & -4 & 1 & o & o & o & o \\ o & o & 1 & -4 & 6 & -4 & 1 & o & o & o \\ o & o & o & 1 & -4 & 6 & -4 & 1 & o & o \\ o & o & o & o & 1 & -4 & 6 & -4 & 1 & o \\ o & o & o & o & o & 1 & -4 & 6 & -4 & 1 \\ o & o & o & o & o & o & 1 & -4 & 5 & -2 \\ o & o & o & o & o & o & o & 1 & -2 & 1 \end{bmatrix}$$

The loss of accuracy to roundoff during Gaussian elimination (triangular factorization) poses a **Dilemma**:

Discretization error —> 0 like $1/N^2$, so for realistic results we want N big.

Roundoff is amplified by $O(N^4)$, so for accurate results we want N small.

Accuracy loses very roughly $4 \log_2 N$ sig. bits to roundoff. For realistic problems (crash-testing car bodies, aircraft wings, ...), typically $N > 10000$. With `double` arithmetic carrying the usual 53 sig. bits (about 16 sig. dec.) we must expect to lose almost all accuracy to roundoff occasionally.

Iterative Refinement mollifies the dilemma:

Compute the *residual* $\mathbf{r} := \mathbf{A}\cdot\mathbf{x} - \mathbf{b}$ for $\mathbf{x}$.

This residual tells how much the alleged solution dissatisfies the equation we wish to solve.

Solve $\mathbf{A}\cdot\Delta\mathbf{x} = \mathbf{r}$ for a *correction* $\Delta\mathbf{x}$.

By reusing the same triangular factors as were used to “solve” $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ for
a solution $\mathbf{x}$ contaminated by roundoff, we compute $\Delta\mathbf{x}$ very quickly.

Update $\mathbf{x}$ to $\mathbf{x} - \Delta\mathbf{x}$ in the hope of reducing its *error* $\mathbf{x} - \mathbf{A}^{-1}\mathbf{b}$, or its residual $\mathbf{r}$, or both.

When N is big, the error can be enormous even though the residual looks negligible.

Repeat as often as necessary.

How often? That’s a good question.

For details see “ Roundoff Degrades an Idealized Cantilever ” by W. Kahan and Melody Y. Ivory, `http://http.cs.berkeley.edu/~wkahan/Cantilever.ps` , from which the following results are extracted.

The following results were obtained from two iterative refinement programs for MATLAB v. 4.2 to run on several brands of computers. The computers group naturally into two families, namely

1: HP PA-RISC, IBM RS/6000, (≈ Apple Power-Mac, Sun SPARC, SGI MIPS, DEC Alpha)
2: Intel-based PCs and clones, 680x0-based Apple Macs, (≈ 68020-based Sun III).

Both programs run on both families of machines though each program was designed for optimal results on its respective family:

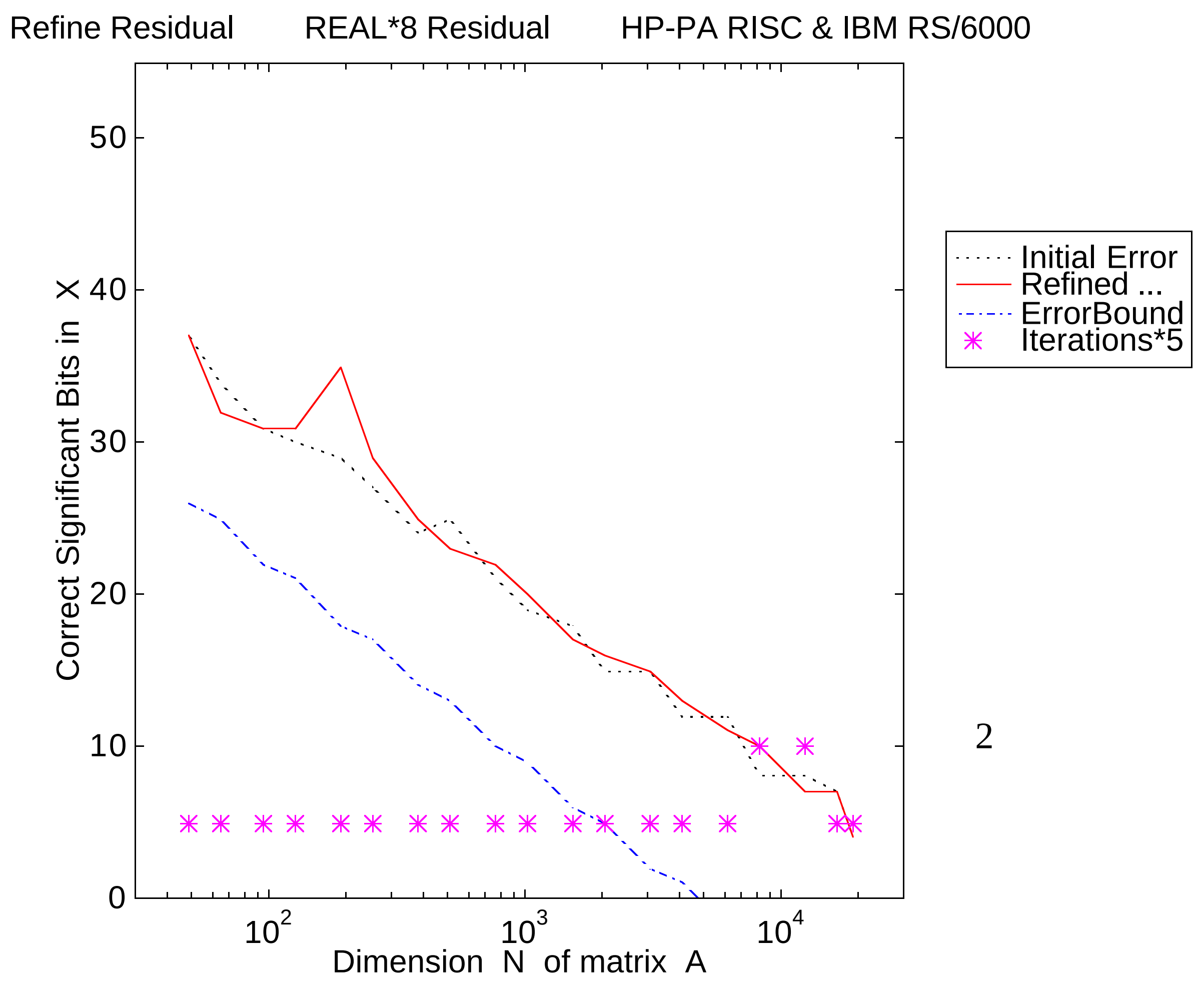
1: Refine Residual program repeats iterative refinement until the residual **r** becomes negligible, and then uses **r** to estimate an upper bound for the error in **x** .
2: Refine Error program repeats iterative refinement until the decrement Δ**x** stops diminishing, and then uses Δ**x** to estimate an upper bound for the error in **x** .

Like Java, MATLAB 4.2 was intended to get the same results on all machines *except* for steps taken to multiply matrices as fast as possible on each machine. Ultimately Java too will have to let fast matrix multiply programs exploit concurrency in pipelines, register files and caches lest performance be degraded by factors worse than 3 . Therefore different machines will deliver different results.

From which family of computers would you expect to get the more accurate results?

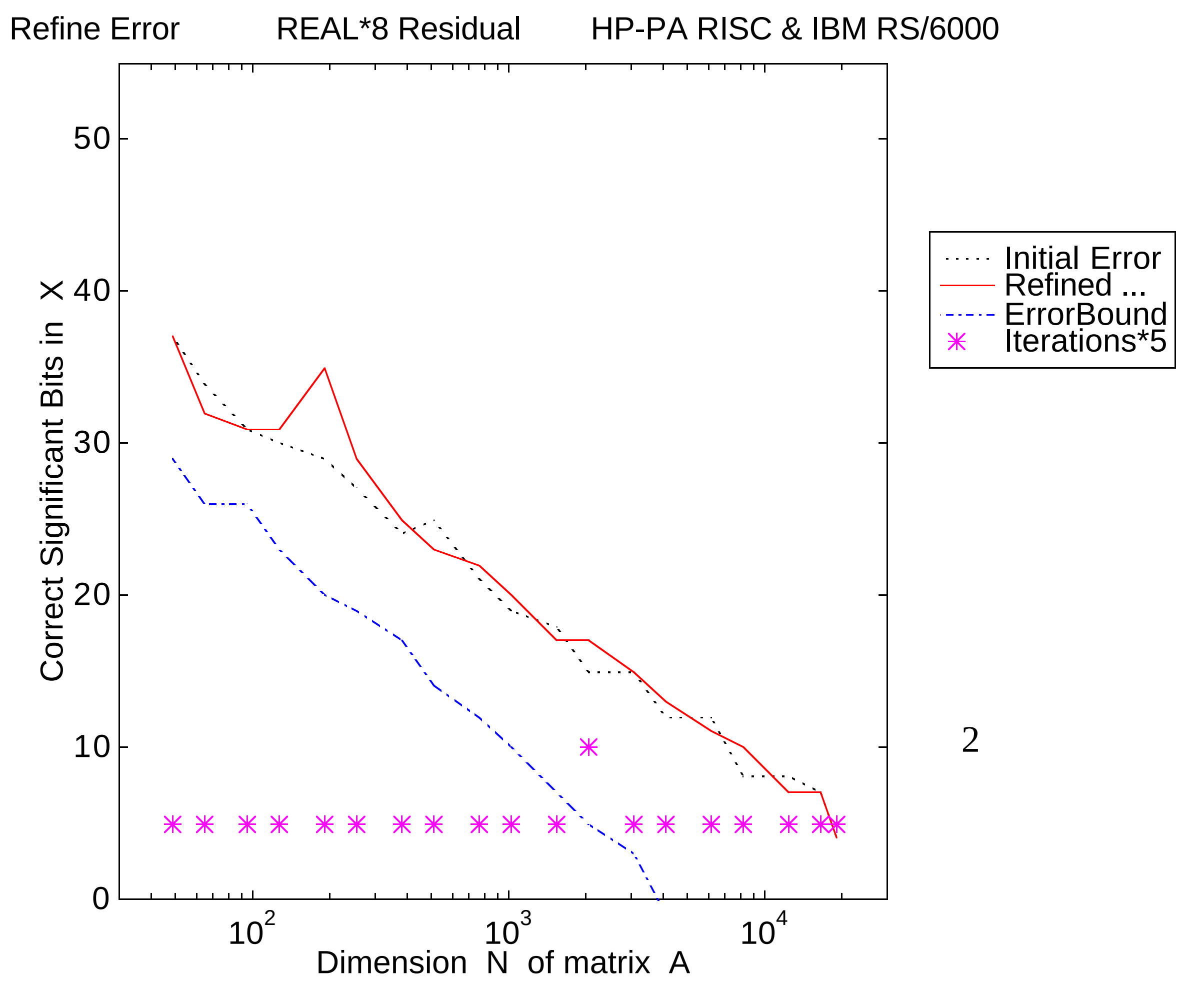
Legend:	············	No. of correct sig. bits in initial **x** delivered by Gaussian Elimination.
	———	No. of correct sig. bits in final **x** delivered by Iterative Refinement.
	-·-·-·-·-	No. of sig. bits computed error bound says are correct in final **x** .
	* * * *	No. of steps of Iterative Refinement required to get final **x** .
		Numbers are plotted against the dimension N of matrix **A** .

The graphs look better printed by a Laser-Printer than displayed by Adobe Acrobat Reader.



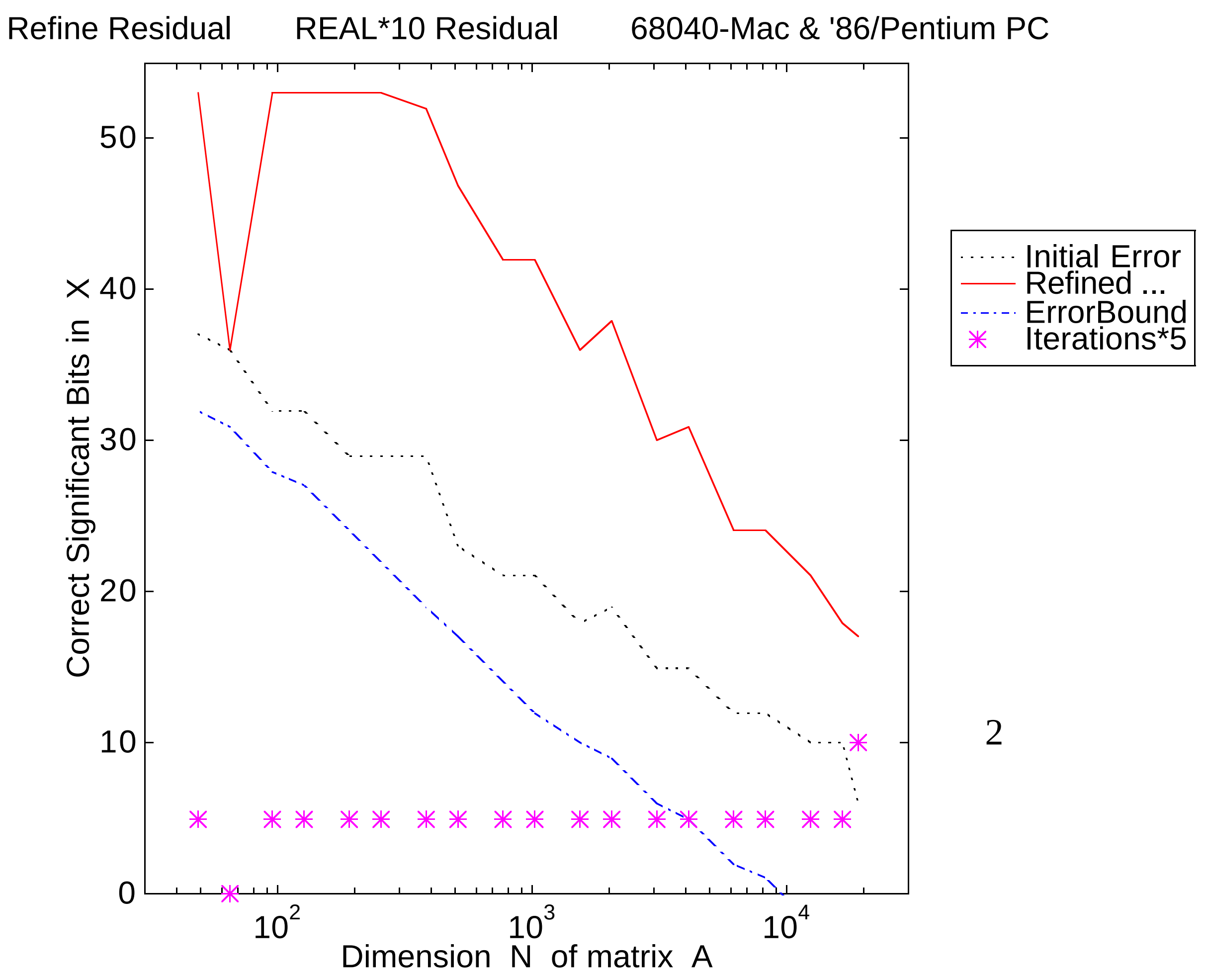
2

Although iterative refinement on RISC-based workstations soon renders the residual negligible, the error isn't improved much (it may be worsened), and the error-bound is about 1000 times too pessimistic.

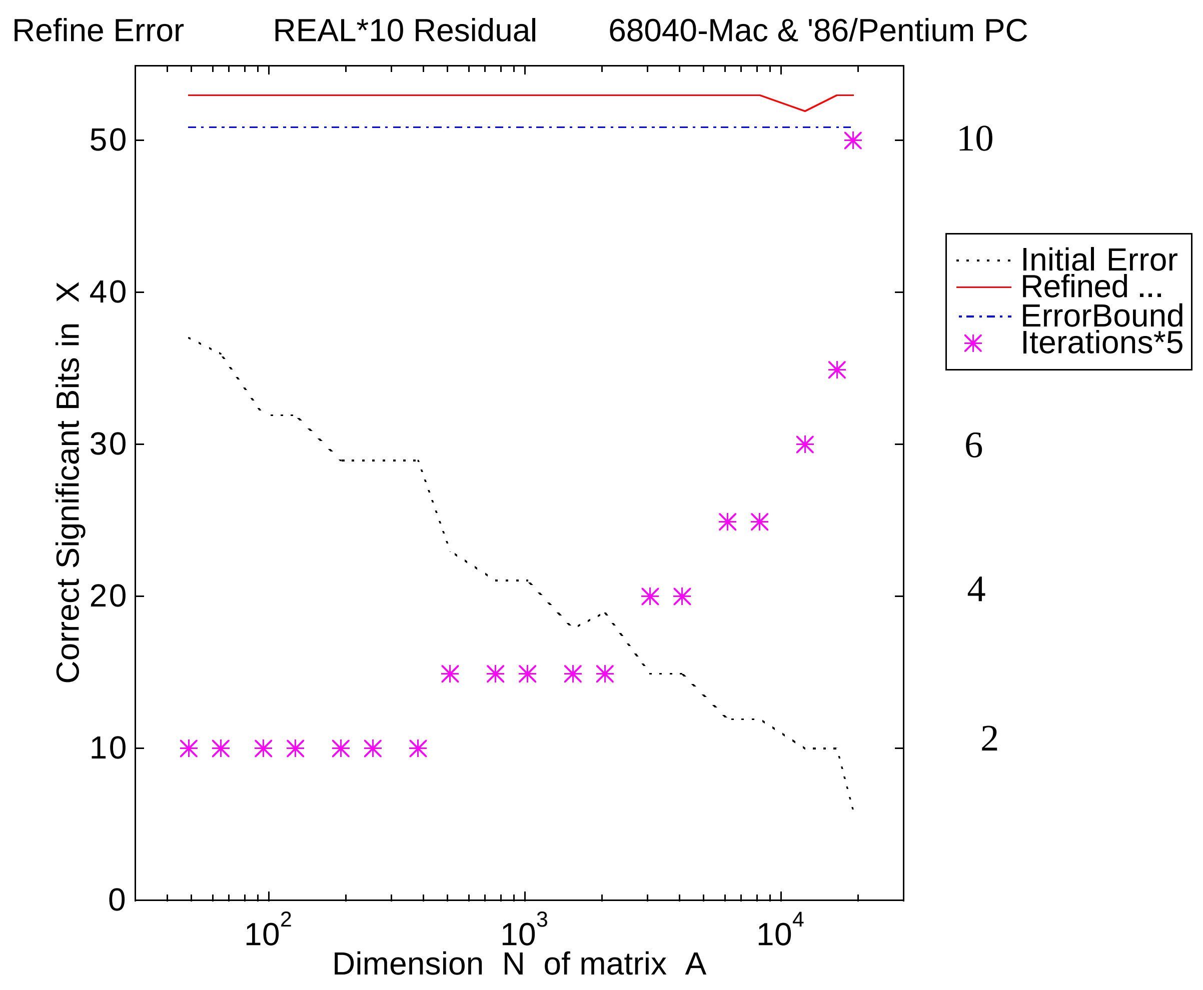


2

On RISC-based workstations, iterative refinement designed to attenuate the error usually doesn't do much good and, as with residual refinement, the estimated error-bound is roughly 1000 times too pessimistic.



Whenever iterative refinement on PCs and old Macs refines the residual it reduces the error too but the user can't know since the error-bound doesn't change (it becomes about a million times too pessimistic).



On PCs and old Macs, iterative refinement designed to attenuate the error succeeds spectacularly, and the estimated error-bound reveals this improvement to the user who can now rely upon it.

The cheaper (and more popular) machines delivered results more accurate by far. They can do the same for eigensystems iteratively refined from occasionally (and inevitably) inaccurate results of MATLAB's "eig" function, thereby enhancing the designs of optimized control systems.

How do the cheaper (and more popular) machines get the more accurate results? They accumulate matrix products in extra-precise registers (11 extra sig. bits) at full speed, though MATLAB 4.2 affords users no access to `long double` variables. `http://http.cs.berkeley.edu/~wkahan/ieee754status/baleful.ps` presents more details. On 680x0-based Macs the current version 5.2 of MATLAB still works that way, but … MATLAB 5.2 on MS Windows no longer accumulates extra-precisely. Why not?

Microsoft's current compilers seem to have turned off the 11 extra bits of precision in Intel-based PC's registers. You paid for it, but Microsoft denies you its benefits.

Why? Intel's 8087 floating-point coprocessor was imminent in 1980 when Bill Gates predicted the sockets built into the IBM PC for it would almost all stay empty. Actually 8087s and later 80287s and 387s filled millions of these sockets — so many that several '87 clone makers entered that market. Cyrix started that way. Meanwhile Gates' prophecy shaped Microsoft's policy and practice; its Basic, Fortran and **C** compilers were optimized for software-simulated floating-point without the '87s' `long double` format. Its support by Borland's **C** forced Microsoft's **C** grudgingly to support it too for a while but it was dropped later when Borland was deemed no longer a threat and Microsoft had begun the development of Windows NT on the DEC Alpha chip, which lacks the `long double` format. Gates' business decisions took no account of the format's value to you.

And now Java forbids you to mention or use extra-precise `long double` arithmetic, though IEEE Standard 754 recommends its use and over 95% of computers on desktops have it built into their hardware. You paid for it, but Java denies you its benefits.

Does this denial make more sense than if Microsoft or Java similarly forbade you to use your …

XGA or SVGA video display, projector or printer, with higher resolution or more colors?
Sound–Board with higher-fidelity audio or four-way stereo?
Higher resolution pointing device? 3–D surface sensing or holographic display?
Microphone? Camera? Radio? TV? Fax board? Scanner?
Faster modem or Ethernet for faster reaction to competitive situations?
Faster CPU capable of supporting higher-resolution Virtual Reality?
Bigger memory, and bigger and faster disks?
… ?

Of course Java does not forbid you to use these extraordinary hardware capabilities if you have them. Quite the contrary; it continually accretes APIs and revisions to its AWT to cope with them. Why should you be denied the same access to better floating-point hardware if you have it?

Was somebody at JavaSoft burnt by Sun's numerically benighted compilers on the old Sun IIIs ? Their Motorola 68020+68881/2 chips' superb floating-point was crippled by anomalies caused by their compiler's denial, to programmers for their own declared variables, of the `long double` register-format in which the compiler evaluated all floating-point expressions. To make matters worse, the compiler rounded registers down to `double` when their contents spilled from the register file. Consequently programmer's could neither predict nor control arithmetic precision. Will Jim Gosling's " Loose Numerics" unleash similar anomalies again?

"Compatibility" is often intoned to excuse doing nothing to fix floating-point. But Java has already inflicted incompatibilities upon JVM implementors in the course of passing from version 1.0 to 1.2 to add features some programmers find useful. Why should floating-point be denied similar relief?

How to support extra-precise arithmetic

Upward compatibility from the Java language, and minimal changes to the JVM, have led Borneo to follow a different approach than has been put before ANSI X3J11 in the C9X proposal. A crucial requirement for both proposals is Control. Exact reproducibility has to be available to a programmer who needs it and who exercises the modest self-discipline required to achieve it. At the same time, a programmer who aims for the widest possible market has to be able to specify what he wishes *not* to control, and in this case his program must able to discover what the compiler has chosen to do. And all this is to be accomplished as parsimoniously as possible without obscurantism or excessive length. To keep this document’s length down, some simplifications and omissions have been perpetrated with a view to persuading the reader that extra-precise arithmetic can be insinuated into Java without destroying its spirit or advantages. For more details see the Borneo specification.

Names for primitive floating-point types or for Borneo floating-point `classes`:

`float`	=	4-byte IEEE 754 Single with 24 sig. bits, usually hardware supported.
`double`	=	8-byte IEEE 754 Double with 53 sig. bits, usually hardware supported.
`long double`	=	10+-byte IEEE 754 Double Extended with at least 64 sig. bits etc.

{ longdouble(k)	=	k-byte IEEE 754 Double Extended (for future use only with k >> 10.)
quadruple	=	long double(16) with 113 sig. bits rounded as IEEE 754/854 requires.
DoubledDouble	=	16-bytes with at least about 106 sig. bits perhaps rounded perversely. }

`indigenous` = the widest floating-point format supported in hardware at full speed
= `long double = double extended` on hardware that does it
= `double` on computer hardware that does nothing wider.

The `anonymous` declaration

Except for the simplest floating-point expressions, temporary values are needed to hold intermediate results of subexpressions, conversions from `integer` types or non-binary formats, and arguments passed to subprograms. If the programmer has not declared the types of these anonymous values explicitly, the language must adopt rules to determine these types. Java's rules are defined by a pass strictly bottom-up through the expression tree, widening the narrower of two operands to match the wider before they are combined. K-R **C** widens to `double` everything narrower and contemplates nothing wider. To fit in with Java's linguistic proclivities, Borneo allows a programmer to declare a minimum width to which everything narrower is to be widened before Java's rules are invoked:

`anonymous float`	follow Java's rules (Borneo's default; it must match Java's)
`anonymous double`	widen every narrower operand to `double` as does K-R **C**
`anonymous long double`	widen every narrower operand to `long double` (use on Intel)
`anonymous indigenous`	widen every narrower operand to `indigenous`.

Of course, Java should be repaired promptly to adopt `anonymous double` as its default, which would then become Borneo's default too. The scope of an `anonymous` declaration is a block.

The `anonymous` declaration is adequate when hardware-supported formats are few. It functions properly with Java's `method` resolution only if some subprogram's arguments explicitly cast to a width narrower than the `anonymous` width are not widened again. This is not what we would have chosen to do had we started from scratch. To diminish the language's capture cross-section for error when augmented by Interval Arithmetic and dynamically variable arbitrarily high precision, we should not widen operands but rather control the accuracy of *generic* (in the Fortran sense) operations and functions. But that is a story for another day.

Gosling's "Loose Numerics" doesn't offer programmers the control our scheme gives them: they can choose our `anonymous double` for reproducibility or `anonymous indigenous` to exploit hardware fully.

Optimizations by the Compiler

Their purpose is to speed the execution of a program without invalidating its output, not to achieve high ratings on benchmarks that pay scant attention to much about programs besides their speed. An optimization that changes a program’s output in a way not licensed by the language nor by the programmer in the text of his program is best deemed a compiler malfunction. Only two such licenses are worth granting for the “optimization” of floating-point operations. One licenses associativity; the other licenses the fused multiply-accumulate on PowerPCs, HP 8000s and MIPS R10000s.

Unlike commutativity (nowadays), associativity can be spoiled by roundoff or over/underflow. For that reason, compilers must always honor the parentheses (see the better formula for Δ on p. 58) or conventions that programmers depend upon to control the order of operations. However, matrix multiplication is one of a few instances in which associativity (here of addition) is worth licensing to keep pipelines full and caches hit at the cost of a usually tolerable change in output; see p. 16.

The *fused* multiply-accumulate (fused mac) computes expressions of the form $\pm x \cdot y \pm z$ with one final rounding error instead of two. Usually this enhances accuracy slightly as well as speed, but it can cause calamity in a few peculiar situations. For instance, $\sqrt{}(b^2 - a \cdot c)$ can signal *Invalid* because of a negative computed value for $(b^2 - a \cdot c)$ even though the predicate $(b^2 < a \cdot c)$ tests FALSE . For this reason, and to match results from computers that lack a fused mac, compilers must inhibit its use when a programmer withdraws an implicit license to use it. Java grants no such license now, but refusing to discuss the fused mac merely ensures that it will be used clandestinely to get higher scores on benchmarks with no provision for a programmer to inhibit it in the few places where it hurts. And programmers who wish to program for only machines that have it need a way to insist upon it. In hardware a fused mac can accelerate DoubledDouble substantially, and can compute expressions like $a \cdot x - b \cdot y$ in the formulas on p. 55 to nearly full `double` accuracy from `double` operands in three operations! Why should Java outlaw special software for special computer configurations?

Certain optimizations are necessary to prevent old-fashioned K-R **C** semantics from being blamed unnecessarily for poor performance. At first sight, a frequently occurring assignment X = Y¤Z involving `float`s X, Y, Z in just one algebraic operation ¤ appears to require that Y and Z be converted to `double`, and that Y¤Z be computed and rounded to `double`, and then rounded again to `float` to be stored in X . The same result X (and the same exceptions if any) are obtained sooner by rounding Y¤Z to `float` directly. In other words, Kernighan-Ritchie **C** runs here as fast as does Java now, so performance is no excuse not to change.

Certain "optimizations" that work on integers must not be used on floating-point. These prohibited optimizations fall into two categories: mistaken use of identities, and invalid statement reordering.

The identities to avoid are the ones that are invalidated by the existence of signed zeros, infinities and NaN. For instance, don't try to " simplify " 0±x , x±0 , x–x , x == x , x != x , 0·x , ∞·x , 0/x , x/0 , x/x , ∞/x or x/∞ . Practically the only identities left are x·y = y·x , x+y = y+x , x–y = –y+x but not –(y–x) , and 1·x = x which is safe only because Java and Borneo disallow signaling NaNs.

Reordering floating-point assignments is dangerous in the presence of floating-point traps, flags and modes. That is why the flags and modes discussed in this document should be made part of the language and thus recognized as floating-point assignments of a sort: the flags are like global variables alterable as side-effects of exceptional operations; the modes are like global variables that influence floating-point operations. Between references (they should be rare) to flags and modes, and within basic blocks, non-speculative floating-point code rescheduling is permissible except perhaps if floating-point traps are enabled. Perhaps floating-point exceptions are best handled without traps, but that is a topic for another day.

Conclusions

We think we’ve made our case. Java’s floating-point hurts everyone everywhere. It didn’t have to.

Java’s floating-point suffers from serious oversights. The same could be said of several other programming languages, some of them venerable, but Java lacks their historical excuses.

Java’s oversights undermine its mission, which is to liberate the world or a large part of it from Microsoft’s hegemony. This mission is like the conduct of a war on many fronts. It is a difficult war to win but easy to lose to a defeat on any front. One of these is the floating-point front.

To win, Java has to surpass Microsoft’s J++ in attractiveness to software developers. This means better design better thought through, less prone to error, easier to debug, … and many other things.

Java’s floating-point is not an example of better design etc., but it can be repaired. We think we have shown how and, more important, why. We think our repairs preserve what is valuable in Java at least as well as JavaSoft has in the course of its updates — nobody who wishes to avoid our flags, `indigenous`, `anonymous`, and directed roundings has to use them. But the repairs must be effected soon or it will be too late. In the computing world the costs to everyone everywhere of correcting mistakes grow horribly with the passage of time unless the mistakes are part of something that doesn’t matter.