Rounding errors are usually avoidable, and sometimes we can afford to avoid them.

BY HANS-J. BOEHM

Small-Data Computing: Correct Calculator Arithmetic

COMPUTERS COMMONLY PERFORM numerical computations using floating point arithmetic,^a typically representing numbers as specified by the IEEE 754 standard. Numbers are represented as $m \times b^e$, where b is base, m is a fixed bit length length fraction (*mantissa*), with an implicit “decimal point” on the left, and e is an exponent. For conventional IEEE “double precision” floating point, the base b is 2, and the mantissa m is 53 bits (approximately 16 decimal digits) long. For a hardware calculator, we might use $b = 10$, with a 12-digit mantissa.^b

Floating-point representations are used pervasively, from large-scale scientific computing problems down

to pocket calculators. They provide a great time-honored compromise between speed of computation and sufficient accuracy to usually provide meaningful results. A 53-bit mantissa can often provide 10 to 15 decimal digits of accuracy in the final result, and modern processors can often perform more than one floating-point operation per cycle.

But conventional floating point arithmetic remains a compromise. The results can be computed quickly, but they are only usually precise enough to be meaningful. Usually the precision loss from rounding to 53 bits is not noticeable, because we are usually computing on measured physical quantities that are far less accurate to start with, and usually well-designed algorithms do not compound these incoming measurement errors too much. But none of those “usually” qualifiers can be dropped, and algorithms are not always well designed.

Most of us are familiar with some of the programming hazards of floating point. We may have observed, for example, that the loop

```
for (x = 0.0; x != 10.0; x += 0.1) { ... }
```

usually fails to terminate. But we are willing to deal with these issues, and write more careful code, in order to get high performance numerical computation.

But sometimes performance, at least in the conventional sense, really doesn't matter. Calculators, which normally target expressions with few operations, are probably the canonical example for this category of applications. That is particularly true when the calculator is really an application running on a smartphone with four 2GHz processor cores. This is also an environment in which users are unlikely to think much about formulating algorithms to optimize floating point precision properties.

Even for calculators, the hazards of floating point extend to more than a few digits off in the last place. For example, if we try to compute:

^a Goldberg, D. What every computer scientist should know about floatingpoint arithmetic. *ACM Computing Surveys* 23, 1 (1991), 5–48.

^b Cochran, D.S. Internal programming of the 9100A Calculator. *HP Journal*, Sept. 1968.

$$\sqrt{1 + 10^{-16} - 1 - 10^{-16}}$$

which is clearly equivalent to $\sqrt{0}$, on any standard calculator, the result is just an error message. The problem is that $1 + 10^{-16}$ is rounded to 1. When we subtract 1, we get 0 instead of 10^{-16} . This is commonly known as “catastrophic cancellation.” We are subtracting two nearly equal numbers, effectively magnifying the input error, yielding a result with few or no meaningful digits. When we then subtract 10^{-16} , the result is a negative number, which can now be represented accurately. Taking the square root of a negative number produces the error. For a more interesting and subtle example along these lines, see the sidebar entitled “A Fixed-Precision Fail.”

There are other cases for which we do get correct answers, but the first 16 digits fail to expose the interesting properties of the result. We may want to see when the decimal expansion of a rational number repeats. Or we may want to see how close Ramanujan’s constant ($e^{\pi\sqrt{163}}$) is to an integer. These tend to be “mathematical” rather than “physical” problems. But we suspect a significant fraction of calculator use is in schools for just that purpose.

The Space Beyond Machine Floating Point

Perhaps the most serious problem with conventional calculator arithmetic is that it has trained us not to even attempt certain problems.

Every calculus class teaches us that we can approximate derivatives with finite differences. We can, usually very accurately, approximate $f'(x)$ by $(f(x + h) - f(x))/h$, with a sufficiently small h . For example, since the derivative of e^x is e^x , we should expect that $(e^{1+h} - e^1)/h$ evaluates to a very good approximation of e , if we use $h = 10^{-1000}$.

This of course does not work on normal calculators. The expressions e^{1+h} and e^1 agree to far more digits than the evaluation precision, and the numerator evaluates to zero, yielding a “derivative” of 0 rather than e .

In fact, the idea of limited precision seems to be sufficiently drilled into us that it occurs to few people to even try something like this. And

A Fixed-Precision Fail

Although conventional calculators usually provide plenty of precision for most purposes, they are prone to occasional misbehavior on surprisingly simple problems. These are more likely to be problems encountered by high school or college students experimenting with mathematical laws, than they are real engineering problems.

One particularly small example involves the trigonometric tangent ($\tan$) and arctangent ($\tan^{-1}$) functions. The expression $\tan(\tan^{-1}(x))$ maps a slope x to the corresponding angle and back, and should clearly always give us back x . But evaluating this expression correctly for large x tends to be quite tricky, since $\tan^{-1}$ of a large argument produces a result very close to $\pi/2$ radians (or 90 degrees), which is a singularity for the $\tan$ function.

On software-based calculators (Android or Web), a result something like the one shown here is common for $\tan(\tan^{-1}(10^{20}))$.

On old-fashioned hardware calculators we tried, the arguably better answer shown below is more common.

The standard Microsoft Windows calculator we tried uses more precision, and hence fails only with even larger values of x .

people often seem surprised when we suggest it.

For numerical differentiation with machine floating point, there is a subtle trade-off in the choice of h , which is likely to be well beyond the expertise of a high school student trying to check a formula on a calculator. And yet there is no reason calculations like this shouldn't just work, even with $h = 10^{-1000}$. The sidebar entitled "Derivatives on a Calculator" pushes this example a bit further.

Our Starting Point

The Android Open Source Project has always included a relatively simple calculator application. This is the default calculator on Pixel phones and many other Android devices. Historically some other third-party Android calculators have also extended the same source code base. This calcula-

tor is designed to be as simple as possible, targeting primarily nontechnical users rather than engineers. It has always offered "scientific calculator" functionality, with trigonometric functions, among others. But the emphasis has been on simple use cases and conformance to Android user interface guidelines.

In versions prior to Android 6.0 Marshmallow, the calculator internally used the "arity" expression evaluation library.^c The calculator uses this library to evaluate traditional infix expressions. Conventional syntax is mildly extended to allow dropping of trailing parentheses and a few other shortcuts. The actual evaluation is performed using double precision machine floating point.

^c Mihai Preda, <https://code.google.com/p/arity-calculator/>

The calculator rounded the result to a smaller number of decimal digits than the 16 provided by the underlying arithmetic. This reduced the probability that a number with a finite decimal expansion like 12.34, but an infinite binary expansion, would be displayed as 12.399999999. But there was an unavoidable tension between not dropping enough digits and generating unpleasant representations, and dropping too many. The latter would either introduce additional error for intermediate results, or it would force the calculator to display a result significantly different from its internal representation. Both could produce unpleasant surprises. A Web search for "Android Calculator bug" shows some of the results. The nature of the complaints also confirms that most users are not happy to tolerate the kind of floating-point issues that a numerical analyst would expect.

Accurate Answers From a Calculator

Our goal was to replace the arithmetic evaluation engine in the Android Calculator with one that was not subject to floating point-rounding errors. We wanted to ensure at a minimum that the displayed final answer was never off by one or more in the final displayed digit.

This can be done using "constructive real arithmetic."^{d,e} Rather than computing a fixed number of digits for each intermediate result, each subexpression is computed to whatever precision is needed to ensure sufficient accuracy of the final result.

Let's say we want to compute $\pi + \frac{1}{3}$, and the calculator display has 10 digits. We would compute both π and $\frac{1}{3}$ to 11 digits each, add them, and round the result to 10 digits. Since π and $\frac{1}{3}$ were accurate to within 1 in the 11th digit, and rounding adds an error of at most 5 in the 11th digit, the result is guaranteed accurate to less than 1 in the 10th digit, which was our goal.

Other operations are somewhat more complex to implement. Multiplication to 10 digits beyond the decimal point might involve evaluating one argument to five digits. If that is zero, we

^d E. Bishop, and D. Bridges. *Constructive Analysis*. Springer Science & Business Media, 1985.

^e Boehm, HJ., and Cartwright, R. Exact real arithmetic: Formulating real numbers as functions. Rice University, Department of CS, 1988.

Indicating position. We would like the user to be able to see at a glance which part of the result is currently being displayed.

Conventional calculators solve the vaguely similar problem of displaying very large or very small numbers by using scientific notation. We use the same approach for the initial display.ⁿ If the user enters “ $1 \div 3 \times 10^{20}$ ”, computing $\frac{1}{3}$ times 10 to the 20th power, the result may be displayed as 3.333333333E19. In this version of scientific notation, the decimal point is always displayed immediately to the right of the most significant digit.

Once the decimal point is scrolled off the display, this style of scientific notation is not helpful; it essentially tells us where the decimal point is relative to the most significant digit, but the most significant digit is no longer visible. We address this by switching to a different variant of scientific notation, in which we interpret the displayed digits as a whole number, with an implied decimal point on the right. Instead of displaying 3.333333333E19, we hypothetically could display 333333333E9 or 3333333333 times 10^9 . In fact, we use this format only when the normal scientific notation decimal point would not be visible. If we had scrolled the above result two digits to the left, we would in fact be seeing ...333333333E7. This tells us the displayed result is very close to a whole number ending in 3333333333 times 10^7 . The two forms of scientific notation are easily distinguishable by the presence or absence of a decimal point, and the ellipsis character at the beginning.

Rounding vs. scrolling. Normally we expect calculators to try to round to the nearest displayable result. If the actual computed result were 0.66666666666667, and we could only display 10 digits, we would expect a result display of, for example 0.666666667, rather than 0.666666666. For us, this would have the disadvantage that when we scrolled the result left to see more digits, the “7” on the right would change to a “6”. That would be mildly unfortunate. It would be somewhat worse if the actual re-

Perhaps the most serious problem with conventional calculator arithmetic is that it has trained us not to even attempt certain problems.

sult were exactly 0.9999999999, and we could only display 10 characters at a time, we would see an initial display of 1.00000000. As we scroll to see more digits, we would successively see ...000000E-6, then ...000000E-7, and so on until we get to ...00000E-10, but then suddenly ...99999E-11. If we scroll back, the screen would again show zeroes. We decided this would be excessively confusing, and thus try to truncate toward zero rather than rounding.

It is still possible for previously displayed digits to change as we are scrolling. But we always compute a number of digits more than we actually need, so this is exceedingly unlikely.

Since our goal is an error of strictly less than one in the last displayed digit, we will never, for example, display an answer of exactly 2 as 1.9999999999. That would involve an error of exactly one in the last place, which is too much for us.

It turns out there is exactly one case in which the display switches between 9s and 0s: A long but finite sequence of 9s (more than 20) in the true result can initially be displayed as a slightly larger number ending in 0s. As we scroll, the 0s turn into 9s. When we immediately scroll back, the number remains displayed as 9s, since the calculator caches the best-known result (though not currently across restarts).

We prevent 9s from turning into 0s during scrolling. If we generate a result ending in 9s, our error bound implies that the true result is strictly less (in absolute value) than the value (ending in 0s) we would get by incrementing the last displayed digit. Thus we can never be forced back to generating zeros and explicitly ensure that we always continue to generate 9s, and 9s never turn into 0s.

Coping with Undecidability

The calculator essentially represents a number as a program for computing approximations. This representation has many nice properties, like never resulting in the display of incorrect results. It has one inherent weakness: Exact equality of two numbers is fundamentally undecidable. We can compute more and more digits of both numbers, and if they ever differ by more than one in the last

ⁿ Numbers that differ from zero by less than 10^{-320} may be displayed as 0.0000000000. See section Coping with Undecidability.

computed digit, we know they are not equal. But if the two numbers were in fact the same, this process would go on forever.

This still improves on floating-point arithmetic—equality is easily decidable, but tells us even less about equality of the true mathematical real numbers approximated by the floating point values.

This undecidability of equality does create some interesting issues. If we divide a number by x , the calculator will compute more and more digits of x until it finds some nonzero ones. If x was, in fact, exactly zero, this process will continue forever.

We deal with this problem using two complementary techniques:

1. We always run numeric computations in the background, where they will not interfere with user interactions, just in case they take a long time. If they do take a really long time, we time them out and inform the user that the computation has been aborted. This is unlikely to happen by accident, unless the user entered an ill-defined mathematical expression, like a division by zero.

2. As we will see, in many cases we use an additional number representation that does allow us to determine that a number is exactly zero. Although this easily handles most cases, it is not foolproof. If the user enters “1÷0” we immediately detect the division by zero. If the user enters “ $1 \div (\pi^2 \div \pi - \pi)$ ” we time out.

Zeros Further Than the Eye Can See

Prototypes of our calculator, like mathematicians, treated all computed results as infinite objects, with infinitely many digits to scroll through. If the actual computation happened to be $2 - 1$, the result was initially displayed as 1.00000000, and the user could keep scrolling through as many thousands of zeroes to the right of that as he desired. Although mathematically sound, this proved unpopular for several good reasons, the first one probably more serious than the others:

1. If we computed $\$1.23 + \7.89 , the result would show up as 9.1200000000 or the like, which is unexpected and confusing.

2. Many users consider the result of $1+2$ to be a finite number, and find it

confusing to be able to scroll through lots of zeros on the right.

3. Since the calculator could not ever tell that a number was not going to be scrolled, it could not treat any result as short enough to allow the use of a larger font.

These problems were largely addressed by evaluating expression to not just a constructive real number, but also to a rational number represented as a (numerator, denominator) pair. The latter is unavailable if the computation involved an irrational number, or the rational representation is too large.

This allows us to tell whether a result has a finite decimal representation, and if so, how many digits there are to the right of the decimal point. We simply look at the fraction in lowest terms. If the denominator has a prime factor other than 2 or 5, the decimal expansion is clearly infinite; no number of multiplications by 10 can turn the fraction into an integer. Otherwise the denominator can be factored as $2^m 5^n$ and the number of nonzero digits to the right of the decimal point is $\max(m, n)$.

If the expansion is finite, we prevent scrolling past that point. We also prevent scrolling through a large number of trailing zeroes to the left of the decimal point. This often leaves us with a short nonscrollable result, for which we can use a larger font. Unlike the floating-point case, such short, large font results are always exact, and never attributable to a number that was merely close to a short decimal representation.

This is, however, fallible in the other direction. For example, we do not compute a rational representation for $1 \div (\pi^2 \div \pi - \pi)$, and hence it is still possible to scroll through as many zeros of that result as you like.

This underlying fractional representation of the result is also used to directly detect, for example, division by zero, making it much less likely that a casual user will ever see a timeout.

Looking Back

The calculator described here is available through Google Play Store, and is also the default calculator distributed with Android 6.0 Marshmallow and later.

Initial reviews of the calculator liked several unrelated UI and functionality changes, but failed to notice the change in arithmetic.^o We were apparently successful in having the accuracy guarantees not interfere with normal use.

The calculator now exposes the rational representation to the user when it is available. That has turned out to be a useful feature on its own, though it was motivated by other considerations.

Feedback has been quite positive. But it, together with our own experience, has also suggested some improvements:

- Scrolling results have generated far more interest than the much more subtle precision improvements. The latter seem to have been recognized only by an absence of bug reports. As a result, performance of a different kind actually does matter: Users did notice sluggishness when scrolling through 30,000 digits of π . And we subsequently switched to the better-performing Gauss-Legendre algorithm for π .

- The semantics of calculator expressions are more subtle and controversial than we had realized. Is $50+10\%$ equal to 50.1 or 55? If the latter, what's $50 \times 5\%$? If 2π is a valid expression, what about π^2 ?

- The most recent versions of our calculator explicitly track rational multiples of π and some other common irrational constants. This allows us to compute a rational result for $\sin(\pi/6)$ in radian mode, as we already did for $\sin(30^\circ)$.

Acknowledgments

The calculator UI design and implementation of course relied on contributions from several others, most notably Justin Klaassen.

^o For example, see <http://www.androidpolice.com/2015/06/12/android-m-feature-spotlight-the-stock-calculator-app-has-been-overhauled-no-longer-uses-floating-point-arithmetic/>.

Hans-J. Boehm (boehm@acm.org) is a software engineer at Google, Palo Alto, CA.