You Could Learn a Lot from a Quadratic: I. Overloading Considered Harmful

Author: *Henry G. Baker, http://home.pipeline.com/~hbaker1/home.html; hbaker1@pipeline.com*

(No sacred cows were physically harmed in the making of this column.)

Probably the single most memorable non-trivial algebraic formula that students ever see is the famous *quadratic formula* for finding the roots of the quadratic equation $Ax^2 + Bx + C = 0$:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

Thus, when students are given the problem of writing code for solving this equation in an elementary programming course, they dredge up this formula, and proceed to try to use it to solve for the roots of a quadratic in the "obvious" way. If the teacher is exceptional, he will provide enough test cases to show that computer arithmetic is not at all like algebraic arithmetic, and the students will learn something valuable when this formula fails in mysterious ways. Unfortunately, most computer scientists don't take this opportunity to provide valuable insights into a wide variety of mathematical and programming issues. Indeed, the *Ada Language Reference Manual* [Ada83] gives a very poor implemention of quadratic equation solving as an *example* to be emulated!

```
-- From [Ada83LRM], 10.1.1, Ex. 1.
with TEXT_IO, REAL_OPERATIONS;
use REAL_OPERATIONS;
procedure QUADRATIC_EQUATION is
  A, B, C, D : REAL;
  use REAL_IO, TEXT_IO, REAL_FUNCTIONS;
begin
  GET(A); GET(B); GET(C);
  D := B**2 - 4.0*A*C;
  if D < 0.0 then
    PUT("Imaginary Roots.");
  else
    PUT("Real Roots : X1 = ");
    PUT((-B - SQRT(D))/(2.0*A));
    PUT(" X2 = ");
    PUT((-B + SQRT(D))/(2.0*A));
  end if;
  NEW_LINE;
end QUADRATIC_EQUATION;
```

Most of the careful work of the Ada language designers on high quality numeric datatypes has gone down the drain with these few lines of careless code. Suppose that the `REAL` variables in this program are implemented as ubiquitous IEEE-754 binary floating point numbers having a 24-bit mantissa and an 8-bit exponent. How can this program fail? If $\mathsf{A} = 0$, then we can get a divide-by-zero exception (or an 'infinity'). If $\mathsf{A} = 1$, $\mathsf{B} = 2^{14} = 16384$, $\mathsf{C} = -1$, then $\mathsf{B}^2 - 4\mathsf{AC} = 2^{28} + 4 \approx 2^{28} = \mathsf{D}$, so $\mathtt{SQRT(D)} = 2^{14}$ and $\mathtt{-B+SQRT(D)} = -2^{14} + 2^{14} = 0$, even though the true roots are approximately $-2^{14} = -16384$ and $2^{-14} \approx .000061$! For another example, if $\mathsf{A} = -\mathsf{C} = 2^{80}$ (about the size of Avagadro's Number) and $\mathsf{B} = 0$, then $\mathsf{B}^2 - 4\mathsf{AC} = 2^{162}$, which *cannot* be represented within the 8-bit exponent range even though the roots ± 1 *can* be represented.

Thus, if `-B` and `SQRT(D)` are of approximately the same magnitude, then we can get massive cancellation, and perhaps produce a root which is *zero* even when $\mathsf{C} \neq 0$, which is impossible, since C/A is the product of the roots![1] If `B` is particularly large or small, then the mere computation of B^2 may cause an exponent underflow or overflow, even when both roots are well within exponent range. Aside from these problems, the program is quite inefficient, since it recomputes `SQRT(D)` and `2.0*A` twice.[2] Given the class of real-time systems that Ada is targeting, it is possible that the end-users may die of more than embarrassment if this example is followed.

Examples like this prove the utter futility of trying to

[1] [Casio86] suggests making the same mistake, thus vastly reducing one's confidence in Casio products. [Menzel60,1.1], [Kahan96], and [Press86] avoid cancellation, and [Young72] provides an exhaustive analysis of cases of the quadratic. Although [Press86] doesn't handle overflow/underflow gracefully, his cookbook is at least elegant:

$$Q = -(1/2)[B + \mathrm{sgn}(B)\sqrt{B^2 - 4AC}]$$
$$x_1 = Q/A$$
$$x_2 = C/Q.$$

[2] Yes, I know that many compilers can do *common subexpression elimination*, but this ability for subroutines like `SQRT` is rare, and what good is an optimization that can't be relied upon?

make programs look more like mathematical formulæ—which hubris is the core premise of FORTRAN ('FORmula TRANslator') and its descendant wannabees. Computer arithmetic doesn't follow most of the rules required by algebra, so trying to make program expressions look like mathematical expressions is foolhardy and dangerous. Such confusion is especially dangerous to the poor students, who don't yet have enough of a solid grasp of either mathematics or programming to be able to navigate these subtle minefields.

Mathematics of Quadratic Equations

For the past 35 years, American high schools have been engaged in a massive Federally-funded study to determine how little mathematics and science can be taught to the populace before a first-world country collapses into a third world economy. Freshmen now arrive at college blissfully ignorant of much of basic algebra, including the algebra necessary to understand and solve a quadratic equation. In particular, most cannot derive the quadratic formula, or even describe the simple symmetries of a quadratic equation.

The usual derivation of the quadratic formula involves "completing the square," but since this step is completely unmotivated, it is (quite properly) dismissed by the student as a mere "trick" and quickly forgotten. A more fundamental approach involves looking at the symmetries of the equation $Ax^2 + Bx + C = 0$ with real coefficients A, B, C, where $A \neq 0$.

The first symmetry of this equation is the observation that the solution does not change when the equation is "multiplied through" by any non-zero constant, including $1/A$ itself (assuming that $A \neq 0$). Thus, we can force the coefficient of the quadratic term to be non-negative by multiplying the equation by $\text{sgn}(A)$:[3]

$$\begin{aligned} 0 &= \text{sgn}(A)Ax^2 + \text{sgn}(A)Bx + \text{sgn}(A)C \\ &= |A|x^2 + \text{sgn}(A)Bx + \text{sgn}(A)C \end{aligned}$$

More importantly, we can *simplify* the equation by *eliminating a parameter* if we divide a non-trivial quadratic equation through by the coefficient of the squared, or "quadratic" term. This produces the "monic" equation:

$$x^2 + (B/A)x + (C/A) = 0.$$

[3]We use the convention that $\text{sgn}(A) = 1$ if $A > 0$ and $\text{sgn}(A) = -1$ if $A < 0$. When $A = 0$, we require only that $\text{sgn}(A)^2 = |\text{sgn}(A)| = 1$, so that $1/\text{sgn}(A)$ is non-singular and $A = \text{sgn}(A)|A|$.

Assuming that $A \neq 0$, we can then assume "without loss of generality" that we have already performed this step, which reduces 3 parameters to 2, for a savings of 33%. This allows us to focus our attention on the simpler equation:

$$x^2 + Bx + C = 0.$$

We now consider the effect on the structure of the equation of performing the substitution $x = -y$. We get:

$$(-y)^2 + B(-y) + C = y^2 - By + C = 0.$$

In other words, negating x *negates* the linear term while *preserving* the signs of the quadratic and constant terms. If we wanted to, we could use this symmetry to force the coefficient of the linear term to be negative with the substitution $x = -\text{sgn}(B)y$:

$$\begin{aligned} 0 &= (-\text{sgn}(B)y)^2 + B(-\text{sgn}(B)y) + C \\ &= y^2 - (\text{sgn}(B)B)y + C \\ &= y^2 - |B|y + C \end{aligned}$$

We can generalize this symmetry by considering the *dilation* $x = ay$, where a is a non-zero real number:

$$(ay)^2 + B(ay) + C = (a^2)y^2 + (aB)y + C = 0.$$

The dilation $x = \sqrt{|C|}y$ can be used to normalize $C \neq 0$ such that the constant term has absolute value 1:

$$\begin{aligned} 0 &= (\sqrt{|C|}y)^2 + B(\sqrt{|C|}y) + C \\ &= |C|y^2 + (B\sqrt{|C|})y + C. \end{aligned}$$

Dividing by $|C|$ produces:

$$\begin{aligned} 0 &= y^2 + (B/\sqrt{|C|})y + \text{sgn}(C) \\ &= y^2 + (B/\sqrt{|C|})y \pm 1. \end{aligned}$$

Dilations also give us another way to get rid of the coefficient $A > 0$ in the equation $Ax^2 + Bx + C = 0$: use the substitution $x = y/\sqrt{A}$:

$$A(y/\sqrt{A})^2 + B(y/\sqrt{A}) + C = y^2 + (B/\sqrt{A})y + C = 0.$$

Finally, we can perform all three simplifications at the same time with the substitution $x = -y\,\text{sgn}(B)\sqrt{|C|/A}$:

$$\begin{aligned} 0 &= A\left(-y\,\text{sgn}(B)\sqrt{\frac{|C|}{A}}\right)^2 + B\left(-y\,\text{sgn}(B)\sqrt{\frac{|C|}{A}}\right) + C \\ &= |C|y^2 - \left(|B|\sqrt{|C|/A}\right)y + C \end{aligned}$$

Now, dividing by $|C|$, we get

$$\begin{aligned} 0 &= y^2 - \left(\frac{|B|}{\sqrt{A|C|}}\right) y + \text{sgn}(C) \\ &= y^2 - \left(\frac{|B|}{\sqrt{A|C|}}\right) y \pm 1. \end{aligned}$$

With this substitution, we have achieved a 67% reduction in parameters, from 3 to 1.

If $C \neq 0$, then we can consider the substitution $x = 1/y$:

$$A(1/y)^2 + B(1/y) + C = A/y^2 + B/y + C = 0.$$

Now if $C \neq 0$, then any root $y \neq 0$, so we can multiply through by y^2 to get:

$$y^2(A/y^2 + B/y + C) = Cy^2 + By + A = 0.$$

The substitution $x = 1/y$ reverses the quadratic end-for-end and exchanges the roles of A and C!

The final symmetry we consider is that of *translation*, in which we perform the substitution $x = y + b$:

$$\begin{aligned} 0 &= A(y+b)^2 + B(y+b) + C \\ &= Ay^2 + 2Aby + Ab^2 + By + Bb + C \\ &= Ay^2 + (2Ab + B)y + (Ab^2 + Bb + C). \end{aligned}$$

This last symmetry provides for the possibility of arranging for the linear coefficient of y to be zero if $2Ab + B = 0$, i.e., $b = -B/2A$:

$$\begin{aligned} 0 &= Ay^2 + (2Ab + B)y + (Ab^2 + Bb + C) \\ &= Ay^2 + \left(\frac{B^2}{4A} - \frac{B^2}{2A} + C\right) \\ &= Ay^2 + \left(C - \frac{B^2}{4A}\right). \end{aligned}$$

In other words, $y^2 = B^2/4A^2 - C/A$, in which case

$$\begin{aligned} y &= \pm\sqrt{\frac{B^2}{4A^2} - \frac{C}{A}} \\ &= \pm\sqrt{\frac{B^2 - 4AC}{4A^2}} \\ &= \pm\frac{\sqrt{B^2 - 4AC}}{2A} \end{aligned}$$

Substituting now for x, we now get

$$\begin{aligned} x &= y + b \\ &= y - \frac{B}{2A} \\ &= \pm\frac{\sqrt{B^2 - 4AC}}{2A} - \frac{B}{2A} \\ &= \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}. \end{aligned}$$

In other words, by studying the symmetries of the equation $Ax^2 + Bx + C = 0$, we were able to find the quadratic formula by ourselves.

We now turn the problem around, and study the quadratic equation that results from the two roots x_1 and x_2:

$$(x - x_1)(x - x_2) = x^2 - (x_1 + x_2)x + x_1x_2 = 0.$$

In other words, if we have a *monic* ($A = 1$) quadratic equation $x^2 + Bx + C = 0$, then the *sum* of the roots is $-B$, and the *product* of the roots is C. In particular, if $C = 0$, then (at least) one of the roots is zero, while if $B = 0$, then $x_1 + x_2 = 0$, i.e., $x_2 = -x_1$.

Furthermore, if $C \neq 0$ and if we have already found one root $x_1 = r$ (which must therefore be non-zero), then we can trivially find the second root: $x_2 = C/r = C/x_1$. In particular, if $|C| = 1$, then $x_2 = \pm 1/x_1 = \pm 1/r$, and the equation has the following simple form:

$$(x - r)(x \pm 1/r) = x^2 - (r \pm 1/r)x \pm 1 = 0.$$

Let us revisit the quadratic formula for $x^2 + Bx + C = 0$ once more, now that we know that $-B = x_1 + x_2$ and $C = x_1x_2$ (x_1, x_2 both real):

$$\begin{aligned} x &= \frac{-B \pm \sqrt{B^2 - 4C}}{2} \\ &= \frac{(x_1 + x_2) \pm \sqrt{(-(x_1 + x_2))^2 - 4(x_1x_2)}}{2} \\ &= \frac{(x_1 + x_2) \pm \sqrt{(x_1 + x_2)^2 - 4x_1x_2}}{2} \\ &= \frac{(x_1 + x_2) \pm \sqrt{x_1^2 + 2x_1x_2 + x_2^2 - 4x_1x_2}}{2} \\ &= \frac{(x_1 + x_2) \pm \sqrt{x_1^2 - 2x_1x_2 + x_2^2}}{2} \\ &= \frac{(x_1 + x_2) \pm \sqrt{(x_1 - x_2)^2}}{2} \\ &= \frac{(x_1 + x_2) \pm |x_1 - x_2|}{2} \\ &= \frac{x_1 + x_2}{2} \pm \frac{|x_1 - x_2|}{2}. \end{aligned}$$

In other words, the first term of the quadratic formula provides the *average*/*mean* (*center of mass*) of the two roots, while the second term of the quadratic formula provides half the (absolute value of the) *difference* of the two roots![4]

Trigonometric Solutions

In the previous section, we saw that the quadratic equation $y^2 - By \pm 1 = 0$ for $B \geq 0$ is a particularly interesting 'universal' quadratic, because the general quadratic can be reduced to this form without performing addition/subtraction, which can sometimes cause spectacular cancellation errors. We now investigate 'trigonometric' solutions to this equation.

We first take up the case where $C = +1$, i.e., $y^2 - By + 1 = 0$, for $B \geq 0$. There are two subcases: $B \geq 2$, and $0 \leq B < 2$. Consider the quadratic formed by the two (positive) roots $y_1 = e^{\theta}$, $y_2 = e^{-\theta}$:

$$\begin{aligned} 0 &= (y - y_1)(y - y_2) \\ &= (y - e^{\theta})(y - e^{-\theta}) \\ &= y^2 - (e^{\theta} + e^{-\theta})y + e^{\theta}e^{-\theta} \\ &= y^2 - 2\cosh(\theta)y + 1 \\ &= y^2 - By + 1. \end{aligned}$$

This last equation is well-defined if $B \geq 2$, since $\cosh(\theta) \geq 1$, for all *real* θ, so we can solve for θ to produce the roots y_1, y_2:

$$\begin{aligned} \theta &= \text{acosh}(B/2) \\ y_1 &= e^{\theta} = e^{\text{acosh}(B/2)} \\ y_2 &= e^{-\theta} = e^{-\text{acosh}(B/2)}. \end{aligned}$$

When $0 \leq B < 2$ in the equation $y^2 - By + 1 = 0$, then we have 2 *complex* roots because $B^2 - 4AC = B^2 - 4 < 0$. We can then utilize hyperbolic trigonometric functions with complex angles, or alternatively, we can identify $B/2$ with $\cos(\phi)$, for some real angle ϕ:

$$\begin{aligned} 0 &= (y - y_1)(y - y_2) \\ &= (y - e^{i\phi})(y - e^{-i\phi}) \\ &= y^2 - (e^{i\phi} + e^{-i\phi})y + e^{i\phi}e^{-i\phi} \\ &= y^2 - 2\cos(\phi)y + 1 \\ &= y^2 - 2\sin(\pi/2 - \phi)y + 1 \\ &= y^2 - 2\sin(\alpha)y + 1 \\ &= y^2 - By + 1. \end{aligned}$$

We then solve for α, y_1, y_2:[5]

$$\begin{aligned} \alpha &= \text{asin}(B/2) \\ y_1 &= e^{i\phi} = e^{i(\pi/2-\alpha)} = ie^{-i\alpha} = \text{cis}(-\alpha)i = \text{cis}(-\text{asin}(B/2))i \\ y_2 &= e^{-i\phi} = e^{-i(\pi/2-\alpha)} = -ie^{i\alpha} = -\text{cis}(\alpha)i = -\text{cis}(\text{asin}(B/2))i. \end{aligned}$$

($\text{cis}(\phi) = \cos(\phi) + i\sin(\phi) = e^{i\phi}$.)

The other major case involves $C = -1$, i.e., $y^2 - By - 1 = 0$, for $B \geq 0$. Consider the roots $y_1 = e^{\theta}$, $y_2 = -e^{-\theta}$:

$$\begin{aligned} 0 &= (y - y_1)(y - y_2) \\ &= (y - e^{\theta})(y + e^{-\theta}) \\ &= y^2 - (e^{\theta} - e^{-\theta})y - e^{\theta}e^{-\theta} \\ &= y^2 - 2\sinh(\theta)y - 1 \\ &= y^2 - By - 1. \end{aligned}$$

Thus, we can now solve for θ, y_1, y_2:

$$\begin{aligned} \theta &= \text{asinh}(B/2) \\ y_1 &= e^{\theta} = e^{\text{asinh}(B/2)} \\ y_2 &= -e^{-\theta} = -e^{-\text{asinh}(B/2)}. \end{aligned}$$

For completeness, we express a root x_1 of the original quadratic $Ax^2 + Bx + C = 0$ trigonometrically:

$$x_1 = \sqrt{\frac{-C}{A}}e^{-\text{asinh}\sqrt{\frac{B^2}{-4AC}}}.$$

"Suitably interpreted," this formula is equivalent to the classical quadratic formula! (Hint: use the mathematical definition: $\text{asinh}(z) = \log\left(z + \sqrt{1 + z^2}\right)$ and the property $\text{asinh}(-z) = -\text{asinh}(z)$.)

An important reason for expressing the solutions of the quadratic equation in this trigonometric form is that all

[4] A classic 'hack' for the max and min functions involves the identities $\max(x_1, x_2) + \min(x_1, x_2) = x_1 + x_2$ and $\max(x_1, x_2) - \min(x_1, x_2) = |x_1 - x_2|$, which yield the formulæ:

$$\max(x_1, x_2) = \frac{x_1 + x_2}{2} + \frac{|x_1 - x_2|}{2}$$

and

$$\min(x_1, x_2) = \frac{x_1 + x_2}{2} - \frac{|x_1 - x_2|}{2}.$$

We have thus shown that these formulæ have the *same* cancellation problems as the quadratic formula, and are thus a *terrible* way to compute max and min!

[5] We utilize the function $\text{asin}(B/2)$ rather than $\text{acos}(B/2)$ because the inverse sin function is better behaved near $B/2 = 0$.

of the operations leading up to this form are numerically stable,[6] thus 'passing the buck' to the trigonometric[7] and exponential functions to properly handle the numerical subtleties, instead of trying to handle them one's self!

Floating Point Arithmetic

Computers have (at least) 2 kinds of arithmetic operations: "integer" ("fixed point") operations and "floating point" operations. Fixed point addition and multiplication are commutative and associative over a limited range in traditional algebraic fashion, whereas floating point addition and multiplication are usually commutative, but almost never associative.

Most algebraic systems encountered by students are commutative and associative, with *matrices* providing the first encounters with non-commutative algebra. Other than floating point arithmetic, the only non-associative algebra normally encountered is that of vector "cross-products," which are neither commutative nor associative. Unfortunately, computer science classes rarely use the student's encounters with floating point arithmetic to point out its non-associativity and other "weird" features.

In floating point arithmetic, there exist numbers $y \neq 0$, such that $x \oplus y = x$,[8] i.e., y 'drowns' in x (why do you think they call it 'floating point'?). For example, on many computers $10^8 \oplus 1 = 10^8$. As a result, one can write loops which continually increment x with y, but will never reach $z > x$! The student usually gets this rude awakening the first time he tries to perform approximate integration by adding up the little rectangles as his calculus class suggests.

In many implementations, there exist numbers $x \neq 0$ such that $x \oslash 2 = 0$. In other words, x is so small that dividing it by 2 can make it identically zero (or crash the program). This situation is called numerical *underflow*. There also exist numbers $x \neq 0$ such that computing $x \otimes 2$ causes either the program to crash with a numerical *overflow*, or to produce a 'number' that prints as `NaN` ('Not-a-Number') or 'infinity'. However, for *binary* floating point implementations, division and multiplication by powers of 2 lose no accuracy in the absence of overflow/underflow,[9] so we can write $x \otimes 2^k$ and $x \oslash 2^k$ as $x2^k$ and $x/2^k = x2^{-k}$, respectively.

Although floating point multiplication and division are not associative, even in the absence of overflow/underflow, they are relatively stable operations in the sense that the floating point result is not too far from the mathematical value (which is not itself usually representable). Square root is even more stable, as it cannot produce overflow/underflow, and fails only for negative arguments.

Probably the most common (and most severe) problem with floating point arithmetic occurs when very precise numbers of the opposite sign and nearly the same value are algebraically summed. In this case, the resulting value may be very far from the correct numerical value, and may be almost totally garbage. Thus, while $x \ominus x = 0$, $(y \oplus x) \ominus x$ may be very far from y, and may even be identically zero, if y first drowns in x.

Let us consider the two roots $x_1 > 0$, $x_2 > 0$ of the quadratic equation $x^2 - (x_1 + x_2)x + x_1x_2 = x^2 + Bx + C = 0$. If x_2 is many orders of magnitude smaller than x_1, then $B = -(x_1 \oplus x_2) = -x_1$, when evaluated in floating point. Thus, if we look at the operation of the quadratic formula when computed using floating point:

$$x_2 = \frac{x_1 \oplus x_2}{2} \ominus \frac{x_1 \ominus x_2}{2} = \frac{x_1}{2} \ominus \frac{x_1}{2} = 0,$$

even when $C = x_1x_2 \neq 0$!

Thus, when implemented with floating point arithmetic, the naive quadratic formula may get one of the roots correct, but completely flub the other one. The naive formula may still produce very poor results even when both of the answers produced from the floating point arithmetic are non-zero.

Exponent Range Analysis

Consider again the quadratic equation (with real roots) $x^2 - Bx \pm 1 = 0$, where $B \geq 0$. We note that since $|x_1| = 1/|x_2|$, if $|x_1| = 2^k$, then $|x_2| = 2^{-k}$, so the two

[6] When $B \approx 2$, we compute either $\cosh(\theta) \approx 1$ or $\cos(\phi) \approx 1$, which implies that $\theta \approx 0$ or $\phi \approx 0$, respectively. The loss of accuracy near $B/2 \approx 1$ is unavoidable due to the approximation of a double root.

[7] If you want to try these trigonometric solutions, you may need to implement the inverse hyperbolic functions $\text{acosh}(x)$, $\text{asinh}(x)$ yourself—either because they weren't included in your language, or because they are broken (inverse hyperbolic functions are rarely tested). In such cases, the mathematical definitions are [Steele90]:

$$\text{asinh}(z) = \log\left(z + \sqrt{1 + z^2}\right)$$
$$\text{acosh}(z) = 2\,\log\left(\sqrt{(z+1)/2} + \sqrt{(z-1)/2}\right).$$

[8] We follow [Knuth81] in using $\oplus, \ominus, \otimes, \oslash$ for the floating point operations of $+, -, \times, /$.

[9] Except for 'denormalized' numbers, which should have been called 'subnormal' numbers.

roots have exponents that are *symmetrically distributed* about $2^0 = 1$. Since floating point exponent range limits are usually more-or-less symmetric about $2^0 = 1$, we can usually be assured (for this equation) that if x_1 is within exponent range, then so will x_2.

Now $B = x_1 + x_2 \geq 0$, so the root of larger magnitude must be non-negative (regardless of the sign of x_2). Call this larger root $x_1 = r > 0$. If the magnitude of B is very large, say $B = 2^k$, for $k \gg 1$, then B will *equal* $x_1 = r$, because $|x_2| = 1/r$ will drown in x_1. So, in this case, we get the equation

$$x^2 - (x_1 \oplus x_2)x + (x_1 \otimes x_2) = x^2 - (x_1)x \pm 1 = 0.$$

In short, if the coefficient $B \geq 0$ in the quadratic $x^2 - Bx \pm 1 = 0$ with real roots x_1, x_2 is in exponent range, then x_1 and x_2 must both also be in exponent range.[10]

So the quadratic equation $x^2 - Bx + C = 0$, $B \geq 0$, $C = \pm 1$, is particularly nice, because its real solutions are always representable. We now solve this equation. If $B \geq 2$, then $B^2 - 4C \geq 2^2 - 4C = 4(1 - C) \geq 0$, so the roots are always real. The larger magnitude root (which must be positive) can be computed as

$$\begin{aligned} B' &= B \oslash 2 \qquad (\text{so } B' \geq 1) \\ x_1 &= B' + \sqrt{B'^2 + C} \\ &\approx B' \otimes \left(1 \oplus \sqrt{1 \oplus (C \oslash B'^2)}\right) \\ x_2 &= C \oslash x_1. \end{aligned}$$

The only possible problem occurs in the step where we compute $C \oslash B'^2$. If B' is very large, say $B' = 2^k$ for $k \gg 1$, then $|C \oslash B'^2| = 2^{-2k}$, which can produce exponent underflow. However, in this case, the underflow isn't serious, because when it happens, we merely produce an extremely small (in absolute value) number which drowns when added to 1. This underflow should therefore be ignored, because we will already be getting the best answer possible.

If $0 \leq B < 2$, on the other hand, then we have two cases: $C = 1$ and $C = -1$. If $C = 1$, then $B^2 - 4AC < 2^2 - 4 = 0$, so both roots are complex. If $C = -1$, then $B^2 - 4AC = B^2 + 4 > 0$, so both roots are real. The larger magnitude root is also positive, so

$$\begin{aligned} B' &= B \oslash 2 \qquad (0 \leq B' < 1) \\ x_1 &= B' + \sqrt{B'^2 - C} \\ &\approx B' \oplus \sqrt{B'^2 \oplus 1} \\ x_2 &= C \oslash x_1 = -1 \oslash x_1. \end{aligned}$$

[10] The proof requires a certain minimum of precision, which is true of essentially all floating point implementations.

We thus conclude that the quadratic equation $x^2 - Bx \pm 1 = 0$ is a particularly nice quadratic, because it can be easily solved when its roots are real, and these roots are representable if and only if the equation itself is representable.

Reducing the General Case

Now that we have a robust quadratic-solver for the special case $x^2 - Bx \pm 1 = 0$, $B \geq 0$, we show how to reduce the general quadratic $Ax^2 + Bx + C = 0$, $A \neq 0$, $C \neq 0$, to this case, or die trying—i.e., if the general case cannot be so reduced, then its roots cannot be represented.

But we already know how to reduce the general equation $Ax^2 + Bx + C = 0$ into this form. We first 'multiply through' by $\text{sgn}(A)$ to produce $|A|x^2 + \text{sgn}(A)Bx + \text{sgn}(A)C = 0$. This step can always be performed without any exceptions, since changing the sign of a floating point number is a trivial operation. We assume that this has already been done in the following.

We next compute $\sqrt{|A|}$ and $\sqrt{|C|}$, which are both representable, since $|A| > 0$, $|C| > 0$, and the absolute value of the exponents of $\sqrt{|A|}$, $\sqrt{|C|}$ are less than the absolute value of the exponents of $|A|$, $|C|$, respectively.

We then form the product $\sqrt{|A|} \otimes \sqrt{|C|}$, which is representable because both $|A|$ and $|C|$ are both representable, so even in the worst case in which $|A| = |C| = M$, where M is the maximum representable value, then the product will be $\sqrt{M}\sqrt{M} = M$.

The most difficult step in the reduction is the formation of $|B| \oslash (\sqrt{|A|} \otimes \sqrt{|C|})$. This quantity may indeed not be representable. Consider, for example, the equation $2^{-k}x^2 - 2 \times 2^k x + 2^{-k} = 0$.

$$\begin{aligned} &|B| \oslash (\sqrt{|A|} \otimes \sqrt{|C|}) \\ &= 2 \times 2^k \oslash (2^{-k/2} \otimes 2^{-k/2}) \\ &= 2 \times 2^k \oslash 2^{-k} \\ &= 2 \times 2^{2k}, \end{aligned}$$

which will not be representable if k is the largest possible exponent. However, in this case, the roots are both equal to 2^{2k}, which is not representable, either.

So, we must allow for an exponent overflow in the formation of the quantity $|B| \oslash (\sqrt{|A|} \otimes \sqrt{|C|})$, and exceptions are not spurious, because they indicate that the roots do fall outside the representable range. However, an underflow here is benign.

Thus, through the use of the substitution $x = -y\,\text{sgn}(B)\sqrt{|C|}/\sqrt{|A|}$, we get the equation

$$y^2 - \left(\frac{|B|}{\sqrt{|A|}\sqrt{|C|}}\right) y + \text{sgn}(C) = 0.$$

We solve this equation, as shown above, and then "back substitute" to produce the roots of the original equation. In order to do this, we must form the ratio $\sqrt{|C|} \oslash \sqrt{|A|}$, which can always be represented, since $|A|$ and $|C|$ are representable. The original roots are then

$$x_1 = -\text{sgn}(B)(\sqrt{|C|} \oslash \sqrt{|A|}) \otimes y_1$$
$$x_2 = -\text{sgn}(B)(\sqrt{|C|} \oslash \sqrt{|A|}) \otimes y_2.$$

This rescaling of y_1, y_2 to form x_1, x_2 may cause exponent overflow and underflows, and these exceptions are *not* benign.

Let us solve the equation $-2^{-k}x^2 - 2x - 2^k = 0$, for k the largest representable exponent. After multiplying through by -1, we get $2^{-k}x^2 + 2x + 2^k = 0$, and $A = 2^{-k}$, $B = 2$, $C = 2^k$. We then form the equation

$$\begin{aligned} 0 &= y^2 - \left(\frac{|B|}{\sqrt{|A|}\sqrt{|C|}}\right) y + 1 \\ &= y^2 - \left(\frac{2}{2^{-k/2}2^{k/2}}\right) y + 1 \\ &= y^2 - 2y + 1. \end{aligned}$$

The solution of this equation is straightforward, and produces the roots $y_1 = y_2 = 1$. We now backsubstitute:

$$\begin{aligned} x_1 &= \left(-\text{sgn}(B)\frac{\sqrt{|C|}}{\sqrt{|A|}}\right) \otimes y_1 \\ &= \left(-\frac{2^{k/2}}{2^{-k/2}}\right) \otimes y_1 \\ &= -2^k \otimes y_1 \\ &= -2^k. \end{aligned}$$

If we now substitute x_1 back into the original equation (which we can't really do without causing overflow) to see if x_1 is really a root, we get

$$\begin{aligned} 0 &= -2^{-k}x^2 - 2x - 2^k \\ &= -2^{-k}(-2^k)^2 - 2(-2^k) - 2^k \\ &= -2^{-k}2^{2k} + 2 \times 2^k - 2^k \\ &= -2^k + 2 \times 2^k - 2^k \\ &= 0. \end{aligned}$$

We have thus succeeded in properly solving a quadratic that couldn't even be put into monic form without causing exponent overflow!

Although the use of square roots for scaling can be considered extravagant in terms of performance,[11] these square roots are quite stable and generate no spurious overflow/underflows. If no other means of scaling is available—e.g., in the *Postscript* language [Adobe90]—then square root scaling is an excellent way to maximize the dynamic range of a quadratic equation solver.

Scale Factors

Although the method presented in the previous sections works, it may not be quite as fast or accurate as one would like, due to the multiple square root operations. In this section, we would like to sketch an analogous scheme in which square roots are merely *approximated* by means of scale factors, so as to reduce the exponent ranges to reasonable values. Since scale factors which are powers of 2 preserve all the available precision (assuming that there is no underflow or overflow), we achieve the simultaneous goals of high efficiency, high precision and wide exponent range ('dynamic range').

We will need the following functions:

$$\text{exponent}(x) = \text{xp}(x) = \text{floor}(\log_2 |x|) + 1$$
$$\text{mantissa}(x) = \text{mn}(x) = |x|2^{-\text{exponent}(x)} = |x|2^{-\text{xp}(x)}$$

With this definition, $1/2 \le \text{mn}(x) < 1$.

Some examples are in order:

$$\begin{aligned} \text{xp}(1) &= \text{floor}(\log_2 |1|) + 1 = 0 + 1 = 1 \\ \text{xp}(2) &= \text{floor}(\log_2 |2|) + 1 = 1 + 1 = 2 \\ \text{xp}(2^k) &= \text{floor}(\log_2 |2^k|) + 1 = k + 1 \\ \text{xp}(2^k + 2^{k-1}) &= k + 1 \\ \text{mn}(-3) &= 0.75 \end{aligned}$$

We thus have for $A \neq 0$ the following factorization

$$A = \text{sgn}(A)\text{mn}(A)2^{\text{xp}(A)}.$$

We will also need a slightly more constrained version of $\text{xp}(x)$ which we will call $\text{xp2}(x)$ and which will be an *even* integer. Thus,

$$\begin{aligned} \text{xp2}(x) &= \text{xp}(x), \quad \text{if xp}(x) \text{ is even} \\ &= \text{xp}(x) - 1, \quad \text{if xp}(x) \text{ is odd.} \end{aligned}$$

[11] Unless the architecture has a fast square root implementation—e.g., Digital's *Alpha*.

We can also define a more constrained version of $\mathrm{mn}(x)$:

$$\mathrm{mn2}(x) = |x|2^{-\mathrm{xp2}(x)}.$$

Note that we have the alternate factorization:

$$A = \mathrm{sgn}(A)\mathrm{mn2}(A)2^{\mathrm{xp2}(A)}.$$

Some additional examples are in order:

$$\begin{aligned}\mathrm{xp2}(1) &= 0\\ \mathrm{xp2}(2) &= 2\\ \mathrm{xp2}(3) &= 2\\ \mathrm{xp2}(4) &= 2\\ \mathrm{xp2}(7) &= 2\\ \mathrm{xp2}(8) &= 4\end{aligned}$$

So, $1/2 \le \mathrm{mn2}(A) < 2$. $\mathrm{xp2}(x)$ and $\mathrm{mn2}(x)$ accomplish two things—they produce an *even* exponent, and make the mantissa symmetrical about 1.

Using these concepts, we can now tackle the general quadratic equation $Ax^2 + Bx + C = 0$, for $A \neq 0$, $C \neq 0$. Since $A = \mathrm{sgn}(A)\mathrm{mn2}(A)2^{\mathrm{xp2}(A)}$, and since $1/2 \le \mathrm{mn2}(A) < 2$, we can divide the equation through by $\mathrm{sgn}(A)\mathrm{mn2}(A)$ without much risk of underflow or overflow. Let us assume that this has already been done, so our new quadratic coefficient is 2^{2k}, for some integer k.

We now have an equation that looks like $2^{2k}x^2 + Bx + C = 0$, for some new B and C. We can rewrite this equation as

$$\begin{aligned}0 &= 2^{2k}x^2 + Bx + \mathrm{sgn}(C)\mathrm{mn2}(C)2^{\mathrm{xp2}(C)}\\ &= 2^{2k}x^2 + Bx + \mathrm{sgn}(C)\mathrm{mn2}(C)2^{2\ell},\end{aligned}$$

for some integer $\ell = \mathrm{xp2}(C)/2$.

Substituting $x = -y\,\mathrm{sgn}(B)2^{\ell-k}$:

$$\begin{aligned}&2^{2k}\left(-y\,\mathrm{sgn}(B)2^{\ell-k}\right)^2 + B\left(-y\,\mathrm{sgn}(B)2^{\ell-k}\right)\\ &\qquad + \mathrm{sgn}(C)\mathrm{mn2}(C)2^{2\ell}\\ &= 2^{2\ell}y^2 - \left(|B|2^{\ell-k}\right)y + \mathrm{sgn}(C)\mathrm{mn2}(C)2^{2\ell}\end{aligned}$$

Dividing through by $2^{2\ell}$, we get

$$\begin{aligned}0 &= y^2 - |B|2^{-\ell-k}y + \mathrm{sgn}(C)\mathrm{mn2}(C)\\ &= y^2 - 2B'y + C' = 0\end{aligned}$$

for $B' = |B|2^{-\ell-k-1}$ and $C' = C2^{-2\ell}$. Furthermore, we know that $B' \ge 0$ and $1/2 \le |C'| < 2$.

If $B' \ge \sqrt{2}$ ($\ge \sqrt{|C'|}$), then we can compute y as follows (ignoring any exponent underflow on $C \oslash B' \oslash B'$):

$$\begin{aligned}y_1 &= B' + \sqrt{B'^2 - C'}\\ &\approx B' \otimes (1 \oplus \sqrt{1 \ominus C \oslash B' \oslash B'})\\ y_2 &= C' \oslash y_1\end{aligned}$$

If $0 \le B' < \sqrt{2}$, then we have two cases: $D = B'^2 - C' < 0$, in which case the roots are imaginary, and $D = B'^2 - C' \ge 0$, in which case the roots are real. In the second case, we compute y as follows:

$$\begin{aligned}y_1 &= B' \oplus \sqrt{D}\\ y_2 &= C' \oslash y_1\end{aligned}$$

Finally, we can rescale the y's back into x's, watching out for overflow/underflows:

$$\begin{aligned}x_1 &= -\mathrm{sgn}(B)2^{\ell-k}y_1\\ x_2 &= -\mathrm{sgn}(B)2^{\ell-k}y_2\end{aligned}$$

We have thus shown how to get essentially the same dynamic range by careful scaling of the coefficients as we got before by taking square roots of the coefficients.

Conclusions

If a programming language is to be used for portable, high quality software, it is imperative that the language provide the programmer access to the *sign*,[12] *exponent*, and *mantissa* of a floating point number. Furthermore, this access must be blindingly fast—of the same order of magnitude as an absolute value, a negation, or a multiplication. If high-speed scaling is not easy to achieve in a language, then the quality and/or cost of the software will suffer. The quality will suffer in that many programmers will find it too difficult to get high speed without a mass of special cases, and the cost will suffer if this mass of special cases must be programmed and maintained.

We also learned the lesson that numerical software is often improved by *not* using a classical mathematical formula itself, but by following instead the *derivation* of this formula. This approach provides a number of advantages: it tends to *canonicalize* the problem by gradually reducing the number of parameters (and thus the

[12] Most hardware architects have heretofore refused to provide for a primitive $\mathrm{sgn}(x)$ operation, because they claim that it is easily emulated by means of a conditional branch operation. But conditional branch operations can be relatively expensive on pipelined architectures, and the penalty for conditional branching is expected to increase. Sounds like Catch-22 to me.

number of cases to consider!), the canonicalization itself removes common subexpressions which result in redundant calculations, and—assuming that the reductions preserve information/accuracy—the performance of operations likely to result in massive cancellation can be deferred until the problem is simple enough to easily understand the effect of this cancellation.

Although reducing the work involved in a calculation is important, it is not nearly as important as *getting the right answer*, or even getting *an* answer![13] In particular, computer arithmetic is very different from mathematical arithmetic: it has limited range and limited precision, and therefore violates some of the algebraic properties of mathematical numbers—most prominently *associativity*. The trend of programming languages to try to cover up these issues instead of facing up to them directly is quite distressing, and is likely to continue the trend of poor quality software and poor software productivity.

In a future column, we will consider other ways to solve a quadratic equation, including various iterative methods.

References

[Ada83] *Reference Manual for the Ada (R) Programming Language*. ANSI/MIL-STD-1815A-1983, 1983.

[Adobe90] Adobe Systems, Inc. *Postscript Language Reference Manual, 2nd Ed*. Addison-Wesley, Reading, MA, 1990. ISBN 0-201-18127-4.

[FORT77] ANSI. *American National Standard Programming Language FORTRAN*. ANSI X3.9-1978, 1978.

[Casio86] Casio, Inc. *Computing With The Scientific Calculator*. Casio, Inc., Japan, SA012200115B, 1986. Manual for Casio fx-115D calculator.

[Goldberg91] Goldberg, David. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys*, **23**, 1 (March 1991), 5-48.

[Kahan96] Kahan, W. "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic." http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

[Knuth81] Knuth, D.E. *Seminumerical Algorithms, 2nd Ed.* Addison-Wesley, Reading, MA, 1981.

[Menzel60] Menzel, D.H., *ed. Fundamental Formulas of Physics, Vol. I*. Dover Publs., New York, 1960, ISBN 0-486-60595-7.

[Moto87] Motorola, Inc. *MC68881/MC68882 Floating-Point Coprocessor User's Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Press86] Press, W.H., *et al. Numerical Recipes*. Cambridge Univ. Press, 1986, ISBN 0-521-30811-9. Sect. 5.5 "Quadratic and Cubic Equations."

[Steele90] Steele, G.L., Jr. *Common Lisp: The Language, 2nd. Ed*. Digital Press, 1990. ISBN 1-55558-041-6.

[Young72] Young, D.M., and Gregory, R.T. *A Survey of Numerical Mathematics, Vol. I*. Dover Publ., New York, 1972. Sect. 3.4 "The Quadratic Equation."

[13] Limited numerical precision effects were shown to cause substantial problems in the operation of the *Patriot* anti-missile system during the Gulf War.

Henry Baker has been diddling bits for 35 years, with time off for good behavior at MIT and Symbolics. In his spare time, he collects garbage and tilts at windbags. *This column appeared in ACM Sigplan Notices 33,1 (Jan 1998), 30-38.*