Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SURVEY PAPER

The Ins and Outs of Solving Quadratic Equations with Floating-Point Arithmetic

Frédéric Goualard

[1]Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, Nantes, France

Correspondence
*Frédéric Goualard. LS2N, 2, rue de la Houssinière, BP 92208, F-44322 NANTES CEDEX 3. Email: frederic.goualard@ls2n.fr

Summary

Solving quadratic equations with radicals on a computer with floating-point arithmetic requires great care to handle correctly all possible parameters. Literature on the subject glosses over the details, often considered as important but tedious to present. As a consequence, most implementations are flawed in one way or another. After having reviewed both the literature and the actual implementations in several programming languages and applications, we present an algorithm inspired from an exposition by Pat Sterbenz from 1974, adapted to take advantage of more recent researches in the field, which leads to a robust quadratic equation solver.

KEYWORDS:
Floating-point arithmetic, numerical errors, robustness, quadratic equation

1 | INTRODUCTION

How many solutions to the quadratic equation:

$$p(x) = x^2 + \left(1 + 2^{-52}\right) x + \frac{2^{51} + 1}{2^{53}} = 0? \tag{1}$$

Only one solution, as GNU GSL 2.7, Scilab 2023.0.0, and the Rust 1.69.0 mathematical library would have us believe? Two real solutions, as the Boost C++ libraries 1.82.0 and the Racket mathematical library v.8.8[cs] report? Or, maybe, two complex solutions, as found by Octave 8.2.0, the Python Numpy library 1.22.3, and MATLAB R2023a? What about $2^{600}p(x) = 0$? It definitely should have the same solutions as $p(x) = 0$, yet GNU GSL, the Rust mathematical library, the Racket mathematical library and the Boost C++ libraries now consider that the equation has no real solution at all!

Equation (1) can be solved easily by hand, and there are indeed two real solutions very close to one another. On devising an up to par quadratic equation solver, Forsythe[1] would say in 1966: "*I venture to guess that no more than five quadratic solvers exist anywhere that meet the general level of the specifications.*" More than half a century later, it seems things have not changed, or maybe for the worst.

The implementers of mathematical libraries are not the only ones to blame, however. Since Forsythe's 1966 article, aptly titled "*How do you solve a quadratic equation?*", several authors have written about the dangers lurking in the implementation of a robust quadratic equation solver, but very few, if any, have described in all the excruciating details the steps to take in order to avoid them. As Forsythe[1] put it, "*They* [The details] *are extremely important to actual computing, but carry less general interest than ideas just presented.*" But, as the saying goes, the devil is in the details. For want of a precise exposition of a reference algorithm to robustly handle all possible inputs, actual implementations seem all to be flawed in one way or another.

A robust algorithm to solve quadratic equations with radicals is the cornerstone of many more elaborate algorithms, like Muller's method[2] for example. As Forsythe quips, "*School examples do factor with a frequency bewildering to anyone who has done mathematics outside of school!*", but real life problems tend to exert a solver to the utmost, requiring it to deliver

correct solutions for inputs large and small, and to handle gracefully exceptional values such as IEEE 754[3] *Not-a-Number*s and infinities, as well as quadratics degenerating into linear equations.

We present in Section 5 a complete algorithm to solve quadratic equations with radicals, for real solutions only—finding complex solutions does not bring new problems, anyway. Contrary to our predecessors, no detail was considered too puny to be exposed. Readers who only wish to implement a robust solver can jump straight to that section after having read Section 2 for the necessary notations. An implementation of the algorithm in the Julia language is given in Program 7; it is also available on github. Most of the ideas underlying the algorithm were sketched by Sterbenz[4] in 1974 for a computer using hexadecimal pre-IEEE 754 floating-point arithmetic, in a book that appears to have been long out-of-print. It is rather distressing to observe that many writers have tried since to reimplement, often less correctly, such an algorithm. In Section 3, we take a look at algorithms presented in the literature since 1974, pointing out some of their flaws in the process; Section 4 does the same for actual implementations in current software.

2 | FLOATING-POINT ARITHMETIC

The IEEE 754 standard[3] defines the representation and the properties of floating-point arithmetic on most modern computers. We only present in this section the elements that are relevant to the understanding of the algorithms presented in Sections 3 to 5. A more in-depth exposition can be found by the interested reader in Muller *et al.*'s book.[5]

The IEEE 754 standard defines a floating-point number v ("*float*", for short) as a binary[1] value of the form:

$$v = (-1)^s \times \sigma \times 2^E,$$

where:

- $s \in \{0, 1\}$ is the *sign bit*;
- $\sigma = b_0.b_{-1}b_{-2}\cdots b_{1-p}$ ($b_i \in \{0, 1\}$), is the *significand*, with one bit for the integer part and $p-1$ bits for the *fractional part* f;
- $E \in [E_{\min}, E_{\max}]$ is the *exponent*.

We note $m = (-1)^s \times \sigma$ the signed significand. Given v a floating-point number, let $\operatorname{sign}(v)$ be defined as:

$$\operatorname{sign}(v) = \begin{cases} -1 & \text{if } v < 0; \\ 1 & \text{if } v \geqslant 0. \end{cases} \tag{2}$$

A floating-point number is stored as a binary string in memory, with s, E, and f stored contiguously; the integer part of the significand is not stored as it is inferred from the value of the stored exponent. The size of the string determines the precision of the floating-point format. Two formats of note are:

- The *single precision* format, stored as a 32-bit string;
- The *double precision* format, stored as a 64-bit string.

To ensure continuity of the computation, the IEEE 754 standard defines special floating-point values:

- $-\infty$ and $+\infty$, to obtain an affine extension of the real number system. When a result is too large in magnitude to be represented in the floating-point format, it is replaced by an infinity. This is an *overflow* situation;
- When some computation has no meaning over the reals (e.g., $\sqrt{-1}$, or $0/0$), its result is represented by an *Not-a-Number* (NaN).

Some configurations of the binary string used to store the exponent E are reserved to encode these special values.

A *normal float* is a float whose integer part of the significand is "1"; a *subnormal float* (with an integer part equal to "0") has an exponent equal to $E_{\min}$. An *underflow* occurs when the result of a computation is so small that it must be coded by a subnormal float. This is a situation we usually try to avoid as subnormals may lead to a loss of precision. For the normal floats, the

[1] The latest revisions of the standard define both decimal and binary floating-point arithmetic. The binary version is still the one used the most, though.

multiplication and division by a power of 2 is errorless in the absence of underflow and overflow, as it only requires manipulating their exponent; the division by a power of 2 that underflows may not be errorless as we cannot decrement the exponent past $E_{\min}$, and the binary point of the significand needs to be right-shifted in order to complete the operation, potentially losing bits in the process.

Figure 1 shows the position on the real line of the floats from a format with 2 bits for the exponent and 4 bits for the significand. As can be seen on the figure, the distance $\mathrm{ulp}(v)$ from a float $v = \sigma \times 2^E$ to the next is not uniform, and equal to $2^{1-p} \times 2^E$. The quantity 2^{1-p} is called the *epsilon* "ε" of the format. It is sometimes defined as the distance from 1 to the smallest float greater than 1.

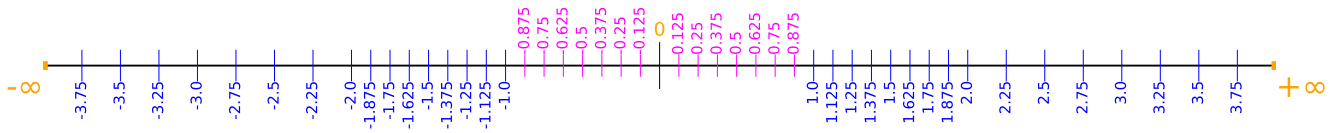

Figure 1 Floating-point numbers on the real line (2 bits for the exponent and 4 bits for the significand). Normal number are in blue below the line, and subnormals are in magenta above it.

The set of floating-point numbers is not closed for arithmetic operations (the sum of two floats may not be a float, for example). The results may, therefore, need *rounding* to be represented as floats. For the arithmetic operations, the IEEE 754 standard mandates that the floating-point result be the closest float to the real result (*correct rounding*). For a real value w, we note $\mathrm{fl}(w)$ its floating-point representation. Given two floating-point values v_1 and v_2, we have:

$$\begin{cases} \left|\mathrm{fl}\left(v_1 \top v_2\right) - \left(v_1 \top v_2\right)\right| \leqslant \frac{\mathrm{ulp}\left(\mathrm{fl}\left(v_1 \top v_2\right)\right)}{2}, \quad \text{for } \top \in \{+, -, \times, \div\}; \\ \left|\mathrm{fl}\left(\sqrt{v_1}\right) - \sqrt{v_1}\right| \leqslant \frac{\mathrm{ulp}\left(\mathrm{fl}\left(\sqrt{v_1}\right)\right)}{2}. \end{cases}$$

thanks to the correct rounding. As a shorthand, we will write $\mathrm{fl}\langle \mathrm{expr} \rangle$ to express the fact that each step of the evaluation of the numerical expression "expr" is rounded according to the floating-point format in use. Example:

$$\mathrm{fl}\left\langle x^2 - y^2 \right\rangle \equiv \mathrm{fl}\left(\mathrm{fl}\left(x^2\right) - \mathrm{fl}\left(y^2\right)\right).$$

When solving quadratic equations, there are two phenomena that will be of importance because they may induce very large errors in the resulting solutions, or even in the counting of the number of solutions:

Absorption. Summing or subtracting two floats requires equating their exponents. This is done by incrementing the exponent of the smallest value in magnitude, and right-shifting the binary point of its significand accordingly. As such, we may lose some bits from the significand that cannot be represented anymore. Consider, for example the addition with 4-bit significands $1.010 \times 2^{10} + 1.101 \times 2^2$:

$$\begin{array}{rl} 1.010 & \times 2^{10} \\ +\ 1.101 & \times 2^2 \end{array} \longrightarrow \begin{array}{rl} 1.010 & \times 2^{10} \\ +\ 0.000\underline{00001101} & \times 2^{10} \end{array}$$

The underlined red bits cannot be represented in the registers of the machine and are lost. The actual computation is then $1.010 \times 2^{10} + 0.000 \times 2^{10}$. The larger value absorbed the smaller one;

Cancellation. When subtracting very close values, the only bits kept are the significands' rightmost ones. If the operands of the subtraction originate from a previous computation, these bits may be the result of successive roundings. Consider, for example the subtraction with 8-bit significands $1.0010\underline{111} \times 2^0 - 1.0010\underline{010} \times 2^0$, where the underlined red bits are the results of roundings in previous computations:

$$\begin{array}{rl} 1.0010\underline{111} & \times 2^0 \\ -\ 1.0010\underline{010} & \times 2^0 \\ \hline 0.0000\underline{101} & \times 2^0 \end{array}$$

After renormalizing the result, we get a value 1.01×2^{-5} whose bits are all the uncertain ones. The subtraction of close operands does not add new errors but it reveals and magnifies the past ones. The cancellation becomes a *catastrophic cancellation* when most or all correct bits disappear in the subtraction.

The code and the examples presented in the next sections consider double precision only, even though the exposition of the main algorithm in Section 5 is done for an arbitrary size. Table 1 synthesizes the various parameters of the single and double formats.

Table 1 Parameters of single and double precision IEEE 754 floating-point formats.

Name	p	$E_{\min}$	$E_{\max}$	ϵ
Single precision	24	-126	127	2^{-23}
Double precision	53	-1022	1023	2^{-52}

3 | A REVIEW OF THE LITERATURE

The method to solve quadratic equations of the form:

$$ax^2 + bx + c = 0, \quad (a, b, c) \in \mathbb{R}^3$$

has been known for a very long time.According to Smith,[6] the ninth century indian mathematician Srīdhara presented it in its current form in his book *Ganita-Sara*: First, compute the *discriminant* Δ:

$$\Delta = b^2 - 4ac \tag{3}$$

Then, depending on the sign of Δ, there are either no real solution, one, or two real solutions:

$$\begin{cases} \Delta < 0: & \text{No real solution;} \\ \Delta = 0: & x = -\frac{b}{2a}; \\ \Delta > 0: & x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}. \end{cases} \tag{4}$$

Even though the naive algorithm based on Equation (4) was used without modification on early machines,[7] computer scientists quickly understood that it was not a good idea to use an algorithm devised for real numbers to solve quadratics on computers with floating-point arithmetic. As summarized by Metropolis (of Monte Carlo method fame) in a 1973 article,[8] there are mainly three pitfalls:

1. The values a, b, and c may be so small or so large that an underflow or an overflow arises when computing Δ, even though the solutions are perfectly representable;
2. When $b^2 \gg |4ac|$, the formula for one of the solutions—depending on the sign of b—suffers from catastrophic cancellation,[2] leading to a very inaccurate result;
3. When $b^2 \approx 4ac$, the computation of the discriminant itself suffers from catastrophic cancellation. A large error in Δ may lead to inferring the wrong number of solutions.

Solutions to the first problem were already pretty well understood and practiced early on, even though they were not always implemented systematically: to compute a complex division—a problem close to the computation of a discriminant—, Smith[9] proposed in 1962 a scaled formula in order to avoid underflows and overflows. Forsythe,[1] again, had presented in 1966 some examples of equations that would lead to overflows or underflows, and he had suggested some scaling strategy to avoid them.

[2] Or "*fantastic cancellation*," as Metropolis puts it.

Sadly, as many others after him, he did not go through with it in that article or in any of his later ones, alluding to the fact that it was important but too wearisome to present.

In 1956, Muller[2] used an alternative formula—amusingly called the "*citardauq*,[10]" for "quadratic" in reverse—originating from Fagnano's XVIIIth century work,[11] to avoid the second problem:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a} = \frac{2c}{-b \mp \sqrt{\Delta}},$$

where "$\mp$" is to be understood as the opposite sign to the one chosen for "$\pm$." Since the sign between $-b$ and $\sqrt{\Delta}$ is different in both formulas, it is possible to avoid catastrophic cancellation by using one formula for a solution and the other formula for the other solution, depending on the sign of b.

For the third problem, it seems that Kahan had already suggested to compute the discriminant with a precision twice the size of the working precision during a lecture at Stanford University[12] in 1966. In 2004, he introduced an algorithm[13] that avoided the need for a larger precision by simulating it.

The major problems in the implementation of quadratic solvers were cleared up, at least theoretically, very early in the history of modern computers. What has been lacking ever since is an exposition of all the steps to take in order to obtain a robust solver. Authors have highlighted the necessity of handling all possible cases: large and small parameters, quadratics degenerating into linear or constant equations, cancellation problems… but, despite our best efforts, we could not find a presentation summarizing all these pitfalls and presenting their solutions in a clear algorithm. Details are always dismissed as trivial or tedious.

This absence of an accessible reference algorithm has led to some naive or flawed algorithms benefiting from undeserved attention. Search on the web how to solve a quadratic equation and you will find over and over again the same naive algorithm implementing the mathematical formulas cited above. Even more troubling, this is also the algorithm presented in many "educational settings", such as the CASIO manual[14] or the Numworks calculator documentation.[15]

Consider such a naive algorithm, as implemented in Program 1.

Program 1: Naive implementation in Julia of a quadratic equation solver.

```
function naive(a,b,c)
    a=Float64(a);b=Float64(b);c=Float64(c)
    delta = b*b - 4*a*c
    if delta < 0
        return (NaN64,NaN64)
    elseif delta == 0
        return -b/(2*a)
    else
        x1 = (-b + sqrt(delta))/(2*a)
        x2 = (-b - sqrt(delta))/(2*a)
        if x1 > x2
            return (x2,x1)
        else
            return (x1,x2)
        end
    end
end
```

According to Forsythe,[12] a satisfactory quadratic equation solver should handle correctly the cases where either a, b, or c are zero, and for each representable solution x_i, it should compute an approximation $\widehat{x_i}$ of it with a precision such that:

$$\frac{|x_i - \widehat{x_i}|}{|x_i|} \leqslant \frac{3}{2}\varepsilon.$$

In addition, Forsythe requires that extended precision be used only to compute the discriminant, if necessary.

Program 1 fails on all these requirements. If a is zero, we get a division by zero. In addition, consider, for example, the equation:

$$p_1(x) = x^2 + 2^{27}x + \frac{3}{4} = 0.$$

It has two solutions:

$$\begin{cases} x_1 = \frac{-2^{27}+\sqrt{2^{54}-3}}{2} \approx -5.59 \times 10^{-9}; \\ x_2 = \frac{-2^{27}-\sqrt{2^{54}-3}}{2} \approx -134217728. \end{cases}$$

Catastrophic cancellation prevents to compute x_1 accurately, and we get with Program 1:

$$\begin{cases} \widehat{x_1} = -7.450580596923828 \times 10^{-9}; \\ \widehat{x_2} = -134217728. \end{cases}$$

Scale the equation to $2^{500}p_1(x) = 0$, and Program 1 will return one solution… NaN. Scale the equation in the other direction to $2^{-1000}p(x) = 0$, and there is only one solution again: -67108864, this time. Evidently, such lack of scale invariance is not acceptable.

Some authors have been more diligent than others, however, and there are some works that still deserve some attention. The rest of this section highlights some of these references in chronological order, pointing out where they are lacking. The Julia code for all of the methods described below is available in the same Julia package on github as the final procedure in Section 5.

3.1 | Nonweiler, 1968

In "Roots of low-order polynomial equations,"[16] Nonweiler presents procedures to solve quartic, cubic, and quadratic equations. He explicitly states at the beginning of the quadratic solving procedure that it will fail on overflow when a is zero "*and in other cases*", which does not bode well for its overall robustness.

The polynomial:

$$p(x) = ax^2 + bx + c = 0$$

is rewritten as the monic polynomial:

$$q(x) = x^2 - 2b'x + c', \text{ with } \begin{cases} b' = \frac{\frac{-b}{a}}{2} \\ c' = \frac{c}{a} \end{cases}.$$

We can then compute Δ:

$$\Delta = b'^2 - c'.$$

If Δ is strictly negative, there are no real solutions. If $\Delta = 0$, we return b'; otherwise:

$$\begin{cases} x_1 = \begin{cases} \sqrt{\Delta} + b' & \text{if } b' > 0; \\ b' - \sqrt{\Delta} & \text{if } b' \leqslant 0; \end{cases} \\ x_2 = \frac{c}{x_1}. \end{cases}$$

The discriminant is evaluated as badly as in the naive procedure and is therefore plagued by the same cancellation problem. On the other hand, the author avoids cancellation in computing the solutions by using the standard formula discerningly, according to the sign of b, for one solution, and by using one of Viète's formulas:[17]

$$\begin{cases} x_1 + x_2 = -\frac{b}{a} \\ x_1 x_2 \quad = \frac{c}{a} \end{cases}$$

for the other solution.

Apart from the transformation into a monic polynomial, no effort is made towards avoiding overflows and underflows.

3.2 | Jenkins, 1975

Jenkins[18] presented in 1975 a generic algorithm to find the roots of a polynomial, which is based on a quadratic solver. This solver handles separately the degenerate quadratics:

1. If $a = 0$ and $b = 0$, it returns $(0, 0)$;
2. If $a = 0$ and $b \neq 0$, it returns $(-\frac{c}{b}, 0)$;
3. If $c = 0$, it returns $(-\frac{b}{a}, 0)$.

In the general case, it tries to avoid overflows when computing Δ by first computing the quantity e:

$$e = \begin{cases} 1 - \frac{2a}{b} \times \frac{2c}{b}, & \text{if } \frac{|b|}{2} \geqslant |c|; \\ \frac{b}{2} \times \frac{b}{2|c|} - \text{sign}(c) \cdot a, & \text{otherwise.} \end{cases}$$

When $e \geqslant 0$ (there is no real solution otherwise), it computes $\frac{\sqrt{\Delta}}{2}$ as:

$$\frac{\sqrt{\Delta}}{2} = \begin{cases} \sqrt{e} \times \frac{|b|}{2}, & \text{if } \frac{|b|}{2} \geqslant |c|; \\ \sqrt{e} \times \sqrt{|c|}, & \text{otherwise.} \end{cases}$$

Lastly, the solutions are computed as:

$$\begin{cases} x_1 = \frac{-\frac{b}{2} - \text{sign}(b) \cdot \frac{\sqrt{\Delta}}{2}}{a}; \\ x_2 = \begin{cases} \frac{\frac{c}{x_1}}{a} & \text{if } x_1 \neq 0, \\ 0 & \text{otherwise.} \end{cases} \end{cases}$$

in order to avoid cancellation.

The formulas used lead to inaccuracies for large or small parameters. Consider for example the equation $2^{-1073}(x^2 - x - 1) = 0$, which has the same two solutions as $x^2 - x - 1 = 0$: $x_1 \approx -0.6180339887498948$ and $x_2 \approx 1.618033988749895$. This algorithm will compute the solutions -0.5 and 1.5.

3.3 | Franklin, 1977

In Chapter 1 of "Fundamental formulas of physics,"[19] edited by Menzel in 1977, Philip Franklin presents the solving of a quadratic equation as the very first formula. He does not consider the case $a = 0$, which is explicitly excluded; in addition, he only takes care of the possibility of cancellation in computing the solutions by using the *citardauq* when appropriate in the *modified Fagnano's formulas*:

$$\begin{cases} x_1 = \frac{-b - \text{sign}(b)\sqrt{b^2 - 4ac}}{2a}; \\ x_2 = \frac{-2c}{b + sign(b)\sqrt{b^2 - 4ac}}. \end{cases}$$

Overflows and underflows are not handled. The case $b^2 \approx 4ac$, which could lead to an inaccurate computation of Δ, is not considered either.

3.4 | Hamming, 1986

The method presented in "Numerical methods for scientists and engineers,"[20] authored by Hamming in 1986, does not offer much that is new: Hamming draws the attention to the risk of cancellation when computing one of the solutions when $b^2 \gg |4ac|$. His solutions is to compute the solution that is not affected by cancellation with the usual formula and to use Viète's formula for the other, as Nonweiler does:

$$\begin{cases} x_1 = \frac{-b - \text{sign}(b)\sqrt{b^2 - 4ac}}{2a}; \\ x_2 = \frac{c}{ax_1}. \end{cases}$$

What is interesting is that, though he is aware of all the other problems that can plague a quadratic equation solver, he dismisses them with typical offhandedness: "*This is not the complete answer on how to evaluate the formula; we still need to worry about (1) underflow, (2) overflow, and (3)* $b^2 - 4ac < 0$*, but* these are not relevant here[3]."

3.5 | Young & Gregory, 1988

The procedure proposed by Young and Gregory in 1988 in their book "A survey of numerical mathematics"[21] is particularly well thought out, but with peculiar features, and it uses floating-point arithmetic from a made-up computer. If one of the parameters

[3] Our emphasis.

a, b, or c is "sufficiently small"—that is, smaller than some user-defined threshold for each parameter—, it is flushed to zero before proceeding. Their procedure is quite convoluted, because, as they put it "*[i]f we wish to develop a computer program which can solve* [the quadratic] *for* any *given value of [a], [b], and [c], then we must first analyze certain special cases.*"

All special cases for either a, b, and c being zero are taken into account separately. For the general case, the algorithm rewrites the formulas to avoid overflows and underflows as much as possible. Cancellation in computing the solutions is handled as in Section 3.4.

Overall, Young and Gregory's algorithm handles correctly almost of the traps, except for the accurate computation of the discriminant. The authors are aware of the necessity to compute it with a precision that is double the working one—and write as much—but they do not offer any algorithm to do so on a computer without the proper format. In addition, being defined for a non IEEE 754 arithmetic, some care needs to be put into adapting the procedure for actual modern computers.

3.6 | Baker, 1998

The 1998 article "You could learn a lot from a quadratic: I. Overloading considered harmful,"[22] by Baker, is particularly interesting as it starts by castigating the authors of the "Ada Language Reference Manual" from 1983 for presenting a flawed procedure (the naive one) for solving quadratic equations. Baker proceeds by recalling the properties of floating-point arithmetic and by presenting two procedures to avoid underflows and overflows. We will only consider the second one, as the first one is less efficient and precise.

Baker consciously avoids considering the cases where either a or c are zero. To avoid underflows and overflows, he takes advantage of the representation of floating-point numbers to scale the parameters. His scaling is not error-free, however.

Given the quadratic polynomial:

$$p(x) = ax^2 + bx + c,$$

we may express a as:

$$a = m_a 2^{r_a} 2^{e_a},$$

with e_a an even integer ($e_a = 2k_a$) and $r_a \in \{0, 1\}$.

Let us divide $p(x)$ by $m_a 2^{r_a}$:

$$p_s(x) = \frac{p(x)}{m_a 2^{r_a}} = 2^{e_a} x^2 + \underbrace{\frac{b}{m_a 2^{r_a}}}_{b_1} x + \underbrace{\frac{c}{m_a 2^{r_a}}}_{c_1}.$$

As Baker puts it, "*we can divide the equation through [...] without much risk of underflow or overflow*", which is not the same thing as no risk at all: if either b or c are very large, for example, the division may be enough to trigger an overflow.

In the same way as for a, let $c_1 = m_{c_1} 2^{r_{c_1}} 2^{e_{c_1}}$, with $r_{c_1} \in \{0, 1\}$ and $e_{c_1} = 2k_{c_1}$. Let us make a change of variable:

$$x = -y \operatorname{sign}(b_1) 2^{k_{c_1} - k_a}.$$

We get:

$$q(y) = 2^{2k_{c_1}} y^2 - |b_1| 2^{k_{c_1} - k_a} y + m_{c_1} 2^{r_{c_1}} 2^{2k_{c_1}}.$$

Divide $q(y)$ by $2^{2k_{c_1}}$:

$$q_s(y) = \frac{q(y)}{2^{2k_{c_1}}} = y^2 - 2\underbrace{|b_1| 2^{-k_{c_1} - k_a - 1}}_{b_2} y + \underbrace{m_{c_1} 2^{r_{c_1}}}_{c_2}.$$

If $b_2 \geqslant \sqrt{|c_2|}$, we have two solutions:

$$\begin{cases} y_1 = b_2 + \sqrt{b_2^2 - c_2} \\ y_2 = \frac{c_2}{y_1} \end{cases}.$$

Note that c_2 is obtained from a computation. Hence $b_2^2 - c_2$ may lead to some cancellation. Baker proposes to compute y_1 as:

$$y_1 = b_2 \left(1 + \sqrt{1 - \frac{\frac{c_2}{b_2}}{b_2}} \right).$$

Eventually, we can resubstitute x for y to get the solutions of the original equation:

$$\begin{cases} x_1 = -\operatorname{sign}\left(b_1\right) 2^{k_{c_1}-k_a} y_1; \\ x_2 = -\operatorname{sign}\left(b_1\right) 2^{k_{c_1}-k_a} y_2. \end{cases}$$

Baker avoids cancellation when computing solutions in the same way as Hamming in Section 3.4 and Young & Gregory in Section 3.5. He does not take any step to accurately compute the discriminant, however, which leads to errors in the number of solutions reported. Additionnally, his scaling procedure is slightly flawed when considering subnormal parameters: since the largest exponent (1023, in double precision) is smaller than the absolute value of the smallest exponent (-1074, in double precision), we may get an overflow when computing the exponent of b_2, as $-k_{c_1} - k_a - 1$ may be larger than $E_{\max}$. This is, for example, the case with the equation:

$$2^{-1073}\left(x^2 - x - 1\right) = 0,$$

which has the two solutions:

$$x_{1,2} = \frac{-1 \pm \sqrt{5}}{2}.$$

Baker's procedure will return the two solutions 0 and ∞.

3.7 | Higham, 2002

"Accuracy and stability of numerical algorithms,"[23] by Higham, is a great book on computing with floating-point arithmetic. Concerning quadratic equations, the author has not much to say, however: he warns against the problem of computing the discriminant accurately, pointing to Kahan's work on the subject, and he suggests to use the method already seen several times to avoid cancellation when computing the solutions:

$$\begin{cases} x_1 = \frac{-b-\operatorname{sign}(b)\sqrt{b^2-4ac}}{2a}; \\ x_2 = \frac{c}{ax_1}. \end{cases}$$

As many authors, Higham evokes the problem of underflows and overflows without elaborating, stating that "*[t]hese ideas can be built into a general strategy [...] but the details are nontrivial*."

3.8 | Nievergelt, 2003

True to its title, "How (not) to solve quadratic equations,"[24] by Nievergelt, explores various algorithms, solving with radicals being only of them. In particular, Nievergelt notes that several softwares solve quadratics by computing the eigenvalues of the companion matrix,[25] which leads to inaccuracies, something we witnessed with Numpy, MATLAB and Octave for the example in the introduction of this article.

To avoid cancellation when computing the solutions, Nievergelt suggests to use the modified Fagnano's formulas:

$$\begin{cases} x_1 = \frac{-c}{(b/2)+\operatorname{sign}(b)*\sqrt{(b/2)^2-4ac}}; \\ x_2 = \frac{(b/2)+\operatorname{sign}(b)\sqrt{(b/2)^2-4ac}}{-a}. \end{cases}$$

The value $b/2$ is used instead of b, which raises the possibility of some additional rounding error, should the value of b be subnormal.

He also suggests to consider the cases where either a, b, or c are zero, without elaborating. Lastly, he advises to use a precision that is double the working one to avoid inaccuracies in computing the discriminant. Yet again, no method is proposed for systems that do not offer such a precision.

3.9 | Kahan, 2004

In the 2004 article "On the cost of floating-point computation without extra-precise arithmetic,"[13] Kahan presents an algorithm to simulate double precision in order to compute the discriminant accurately. He also points out the necessity to handle exceptional parameters and underflows/overflows in a quadratic solver, but without providing details.

A short MATLAB program to compute the solutions of a quadratic equation is provided in the article, without any provision to handle degenerate equations, or underflows/overflows, as the main focus seems to be on the accurate computation of the

discriminant, for which a very length MATLAB program is also provided. The necessary algorithm is actually quite short but Kahan had to consider various environments, whose arithmetic had different properties.

In section 5, we will present a short algorithm to compute the discriminant according to Kahan's method. It is interesting to note that the properties of this algorithm have only been proved two years later by Boldo and others.[26,27,28]

3.10 | Einarsson, 2005

Despite its name, the book "Accuracy and reliability in scientific computing,"[29] by Einarsson, seems only concerned, when considering the solving of a quadratic equation, by the potential cancellation when computing the solutions. His method is the same as Menzel's or Nievergelt's, relying on the *citardauq* for one of the solutions. As the author says himself, "*Even a simple problem such as computing the roots of a quadratic equation needs great care.*" Nevertheless, nothing more is said to that effect.

3.11 | Press et al., 2007

The often-cited book "Numerical recipes: the art of scientific computing,"[30] by Press *et al.*, has something to say about many numerical problems, the solving of quadratic equations being one of them. Sadly, it is disappointingly light on the matter, simply considering the cancellation problem when computing the solutions. Its solution is the one already seen several times: use the naive formula to compute the solution unaffected by cancellation and obtain the other solution through one of Viète's formulas. Nothing is said about exceptional parameters, or underflows and overflows. The problem of accurately computing the discriminant is not considered either.

3.12 | Dahlquist & Björck, 2008

The book "Numerical Methods in Scientific Computing,"[31] by Dahlquist and Björck, presents the same method as Press *et al.*, with the same shortcomings. Apart from computing the solutions with the naive formula and one of Viète's formulas, nothing is said about the computation the discriminant or the possibility of underflows and overflows.

3.13 | McNamee & Pan, 2013

The book "Numerical methods for roots of polynomials,"[32] part II, by McNamee and Pan, devotes its section 12.4 to the errors encountered when solving quadratic equations. The authors warn about the case where a is zero or near zero, remarking that it may happen often with some algorithms such as Muller's method, which use a quadratic equation solver as one of their subroutine. The cancellation in computing solutions is addressed by using modified Fagnano's formulas, as seen in Section 3.8, for example.

The authors allude to the problem of underflows and overflows without addressing it in any way. Nothing is said about the accurate computation of the discriminant and the handling of exceptional parameters.

3.14 | Panchekha et al., 2015

The article "Automatically improving accuracy for floating point expressions,"[33] by Panchekha *et al.* is particular in that it uses the Herbie tool to rewrite the naive formula in order to limit errors when evaluating it. For the purpose of demonstrating Herbie's abilities, the authors only consider the formula:

$$x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

The tool is able to rewrite the expression to avoid both catastrophic cancellation and overflows for the double precision format:

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} \rightsquigarrow \begin{cases} \dfrac{\frac{4ac}{-b+\sqrt{b^2-4ac}}}{2a} & \text{if } b < 0; \\ \dfrac{-b-\sqrt{b^2-4ac}}{2a} & \text{if } 0 \leqslant b \leqslant 10^{127}; \\ -\frac{b}{a} + \frac{c}{b} & \text{if } b > 10^{127}. \end{cases}$$

Though interesting, the article does not present a full-fledged algorithm to solve quadratic equations and some of the problems evoked at the beginning of this section are not addressed.

3.15 | Mastronardi & van Dooren, 2015

In "Revisiting the stability of computing the roots of a quadratic polynomial,"[34] Mastronardi and van Dooren present a scheme to solve quadratic equations accurately. They exclude the case $a = 0$, but consider the posssibilities that b or c be zero.

If $c = 0$, we have:

$$\begin{cases} x_1 = \frac{-b}{a}; \\ x_2 = 0. \end{cases}$$

If $b = 0$, we have:

$$\begin{cases} x_1 = \sqrt{\frac{-c}{a}}; \\ x_2 = -x_1. \end{cases}$$

The authors consider both complex and real roots, so they do not have to handle differently the case $ac > 0$. On the other hand, they do not consider the possibility of underflow or overflow when computing $\sqrt{-c/a}$.

If a, b, and c are all different from 0, we start by computing the coefficients $(1, b_1, c_1)$ of the monic quadratic polynomial:

$$p(x) = ax^2 + bx + c \rightsquigarrow p_m(x) = x^2 + \underbrace{\frac{b}{a}}_{b_1} x + \underbrace{\frac{c}{a}}_{c_1},$$

and we then make a change of variable:

$$y = \frac{-x}{\alpha}, \text{ with } \alpha = \operatorname{sign}(b_1)\sqrt{|c_1|},$$

and consider the polynomial $q(y) = p_m(-\alpha y)/\alpha^2$:

$$q(y) = y^2 - 2\beta y + e,$$

with:

$$\beta = \frac{|b_1|}{2\sqrt{|c_1|}}, \quad e = \operatorname{sign}(c_1).$$

The roots of $q(y)$ are then:

$$\begin{cases} y_1 = \beta + \sqrt{\beta^2 - e}; \\ y_2 = \beta - \sqrt{\beta^2 - e}. \end{cases}$$

They must be computed as follows:

$$\begin{cases} \left.\begin{array}{l} y_1 = \beta + \sqrt{\beta^2 + 1} \\ y_2 = -\frac{1}{y_1} \end{array}\right\} & \text{if } e = -1; \\ \left.\begin{array}{l} y_1 = \beta + \sqrt{(\beta+1)(\beta-1)} \\ y_2 = -\frac{1}{y_1} \end{array}\right\} & \text{if } e = 1 \wedge b \geqslant 1; \\ \text{No real solution} & \text{if } e = 1 \wedge b < 1. \end{cases}$$

The roots of the original polynomial are then computed as:

$$\begin{cases} x_1 = -\alpha y_1; \\ x_2 = -\alpha y_2. \end{cases}$$

No special care is taken in computing the discriminant accurately. Note also that the transformations performed are not error-free and, therefore, introduce more rounding errors in the process.

3.16 | Beebe, 2017

The book "Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library,"[35] by Beebe, is a very large book that is a treasure trove of numerical algorithms. The entire chapter 16 is devoted to quadratic equations.

Beebe's exposition is one of the few that explicitly states that the solver must handle all inputs, even NaNs, infinities and zeros, and that presents the formulas to use in those cases. He also presents the three major problems when solving a quadratic, noting that we need extra precision for computing the discriminant. He gives an algorithm to simulate the double precision needed, which is akin to Kahan's.[13]

Given an equation of the form:

$$ax^2 + bx + c = 0 = m_a 2^{e_a} x^2 + m_b 2^{e_b} x + m_c 2^{e_c},$$

the algorithm starts by scaling all the coefficients by dividing them by $e = \max(e_a, e_b, e_c)$:

$$m_a 2^{e_a} x^2 + m_b 2^{e_b} x + m_c 2^{e_c} = 0 \rightsquigarrow m_a 2^{e_a - e} x^2 + m_b 2^{e_b - e} x + m_c 2^{e_c - e} = 0.$$

This reduces the possibilities of underflows and overflows without averting them entirely (The equation $2^{-1073}(x^2 - x - 1) = 0$ is not correctly solved, for example, and does not give the same answers as $x^2 - x - 1 = 0$). The solutions are then computed with the modified Fagnano's formulas.

4 | A REVIEW OF THE IMPLEMENTATIONS IN SOFTWARE

An article may dismissively consider some of the problems that can plague a quadratic equation solver, referring the reader to other publications or to "common sense" for details. A piece of software does not have that luxury, and must implement each and every strategy that is necessary to handle correctly all inputs.

Not all programming languages offer standard facilities to solve quadratic equations, maybe because it is considered so simple that any programmer can implement his/her own when needed. A look at the information available on websites for programmers, such as StackOverflow, shows that the suggestions for the implementation of a quadratic solver usually range from the downright flawed to the quite incomplete. Is the code implemented in "standard" libraries better?

4.1 | C++ Boost libraries

The C++ Boost libraries are a huge set of code that is oftentimes the antechamber for code that will be included into a future standard of C++. As such, it is usually of high quality.

Boost implements the `boost::math::tools::quadratic_roots()` function to solve quadratic equations with radicals. The comment in the header of the source code for that function refers to a discussion on StackOverflow about a numerically stable method for solving quadratic equations.

The algorithm used considers separately the various cases where either a, b, or c are zero. Nothing is done about the possibility of underflow or overflow, however. To avoid cancellation in computing the solutions, it uses the modified Fagnano's formulas. The discriminant is accurately computed using Kahan's algorithm,[13] with an `fma` instruction when available.

Overall, the algorithm used is quite good, apart for the absence of underflow/overflow management.

4.2 | GNU Scientific Library

The GNU Scientific Library is a project with a long history starting in 1996 to provide the C language with a "standard" library for numerical code. It offers the function `gsl_poly_solve_quadratic()` to solve quadratic equations with radicals.

The algorithm handles separately the case of a quadratic degenerating into a linear equation. Strangely, when a and b are zero, the function returns that the equation has no solution, without testing the value of c (if c is zero, it should report that any value is a solution).

Nothing is done to avoid underflows and overflows. In addition, the discriminant is computed with the textbook formula ($\Delta = b^2 - 4ac$) with no extra precision. The documentation for the function acknowledges that by stating that "[the] *errors may cause a discrete change in the number of roots.*". Cancellation in computing the solutions is avoided by using the modified Fagnano's formulas, as the Boost libraries do.

4.3 | Numworks calculator

The Numworks calculator is a French scientific calculator extensively used in France high schools. It is also on the list of permitted calculators for national exams in many other countries. It is an interesting device as it offers the possibility to program it using the Python language, and its source code is fully available.

There are two ways to solve quadratic equations with the calculator: either use the module `polynomial` in the Python emulator, or use the "Equations" application.

The `polynomial` module implements the naive algorithm from Program 1, plain and simple. It has, therefore, all its flaws.

The "Equations" application is embedded in an environment that provides it with symbolic facilities. As such, some underflows and overflows may be averted through behind-the-scene scaling of the arguments. This is not always the case, however: for the equation $2^{-1073}(x^2 - x - 1) = 0$, it finds 0.5 as the only solution, while it has two:

$$\frac{-1 \pm \sqrt{5}}{2}.$$

According to the source code, the discriminant is computed without any extra precision, and the two solutions are obtained with the naive formulas. As a consequence, the application falls prey to all the traps laid on the path of quadratic equation solvers.

Though a fantastic tool in its own right, with an unusual policy regarding its source code, it is rather sad that a teaching device such as the Numworks calculator should not offer any robust means to solve a "simple" quadratic equation.

4.4 | Racket "math" library

The `math` module of the Racket language offers the function `quadratic-solutions` to solve quadratic equations.

The function uses an algorithm devised by Pavel Panchekha in 2021, and presented in an entry on his blog. As the documentation states, the algorithm tries to handle cancellations and overflows correctly.

Degenerate cases are not handled correctly however, and the function may return incorrect results when either a, b, or c are zero (Issue #81, raised by the author). Cancellations in computing the solutions are avoided by using the modified Fagnano's formulas.

Panchekha did put some thought in the implementation of the code computing the discriminant to avoid losing accuracy and raising overflows. However, his algorithm relies on modified expressions to avoid overflows and underflows instead of scaling. As a consequence, some accuracy may be lost for very large and very small parameters (for example, the function reports that the equation $2^{-1073}(x^2 - x - 1) = 0$ has the two solutions $-2/3$ and 1.5, while the true solutions are $(-1 \pm \sqrt{5})/2$). There is also an error in the computation of the discriminant, whereby some accuracy may be lost, misdiagnosing the number of solutions of an equation (for example, the equation $-312499999999x^2 + 707106781186x - 400000000000 = 0$, taken from Nievergelt's article,[24] has the two very close solutions ≈ 1.131369396027 and ≈ 1.131372303775, while Racket's algorithm considers that it has no real solution (Issue #83, raised by the author).

4.5 | Rust "roots" module

The `roots` module of the Rust language offers the function `roots::find_roots_quadratic()` to solve quadratic equations.

According to its source code, the algorithm used handles separately the linear case. The discriminant is computed with the naive formula without any extra precision, and no provision is made to avoid underflows and overflows. On the other hand, cancellation is avoided in computing the solutions by using the modified Fagnano's formulas.

4.6 | Scilab

The primary way to solve a quadratic equation in Scilab seems to be with the `roots()` function, which, by default, computes the eigenvalues of the companion matrix. There is, however, a report titled "Scilab is not naive"[36] on the Scilab website, which presents an algorithm to solve quadratics with radicals. That algorithm seems to be used for quadratic equations in `roots()` when we pass the right parameter.

To avoid cancellation when computing the solutions, the algorithm uses the modified Fagnano's formulas as seen already in Section 3.8. The report does not consider the problem of accurately computing the discriminant, and restricts its handling of

overflows and underflows to the computation of the discriminant only. Two different expressions are used to compute $s = \sqrt{\Delta}$ in order to limit the possibility of overflow. Given $b' = b/2$:

- If $|b'| > |c|$, we compute:
$$\begin{cases} e = 1 - \frac{a}{b'}\frac{c}{b'}, \\ s = \operatorname{sign}(e) \times |b'| \sqrt{|e|}; \end{cases}$$
- If $|c| > |b'|$, we compute:
$$\begin{cases} e = b'\frac{b'}{c} - a, \\ s = \operatorname{sign}(e) \times \sqrt{|c|}\sqrt{|e|}. \end{cases}$$

Strangely, that strategy for averting—but not completely eliminating—overflows is neither included in the algorithm presented in the report, nor present in the code of the C++ function `FindQuadraticPolynomialRoots()` of the Scilab source code on github.

5 | A COMPLETE AND ROBUST ALGORITHM

We have witnessed in recent articles and current implementations the consequences of the very traps we were warned against in the literature from the sixties. A robust quadratic equation solver should:

1. Handle exceptional parameters (NaNs and infinities);
2. Handle degenerate cases where either a, b or c are zero;
3. Avoid cancellation when computing Δ with $b^2 \approx 4ac$;
4. Avoid underflows and overflows when the solutions are representable;
5. Avoid cancellation when computing the solutions when $b^2 \gg |4ac|$.

In his book "Floating-point computation" from 1974, Sterbenz had already considered in details four of the five problems for a computer using hexadecimal pre-IEEE 754 arithmetic. The fifth problem, the accurate computation of Δ, he had dismissed by advocating to perform the computation with twice the working precision, which may not always be possible. Our presentation will address this problem by using Kahan's algorithm from 2004.[13] We will also flesh out some of his remarks about underflows or overflows affecting intermediate computations.

We will consider the five pitfalls in order, before giving the complete algorithm. Pitfall 5 will be considered together with Pitfall 4.

5.1 | Pitfall 1: Exceptional parameters

Handling exceptional parameters simply requires to detect them at the beginning of the function. Programming languages usually provide methods to determine whether a float is an NaN or an infinity (e.g., `isnan()` and `isinf()`, in C and C++). We also have to decide what we will return in that case. In our Julia implementation, we have chosen to return a different value than when there are no real solutions ($\Delta < 0$).

5.2 | Pitfall 2: Degenerate cases

Table 2 shows all possible degenerate cases. When all three parameters are zero, all reals are solution. What should we return then? We have decided to return the pair $(\infty, -\infty)$ in this situation. Since we will always return two solutions sorted in ascending order, this pair cannot be mistaken for the case where two solutions are not representable and their computation overflowed.

When there are no solutions, real or complex, we will return one NaN. When there are only two non-real solutions, we will return a pair of NaNs. This is of course possible because the Julia language is flexible enough to permit it. With a more rigid language, we would have to use a different convention.

Table 2 Degenerate cases when solving a quadratic equation.

Case	Real solutions
1. $a = 0 \wedge b = 0 \wedge c = 0$	$\mathbb{R}$
2. $a = 0 \wedge b = 0 \wedge c \neq 0$	$\varnothing$
3. $a = 0 \wedge b \neq 0 \wedge c = 0$	$\{0\}$
4. $a = 0 \wedge b \neq 0 \wedge c \neq 0$	$\{-\frac{c}{b}\}$
5. $a \neq 0 \wedge b = 0 \wedge c = 0$	$\{0\}$
6. $ac > 0 \wedge b = 0$	$x \in \mathbb{C}$
7. $ac < 0 \wedge b = 0$	$\{-\sqrt{-\frac{c}{a}}, \sqrt{-\frac{c}{a}}\}$
8. $a \neq 0 \wedge b \neq 0 \wedge c = 0$	$\{-\frac{b}{a}, 0\}$

In Cases 4 and 8 in Table 2, the computation of $-c/b$ and $-b/a$ may underflow or overflow. That would mean that the solution is not representable with the floating-point format used, and such an event is therefore unavoidable. The solution returned will then be 0 or $\pm\infty$, depending on the event. In a private communication with Gardiner and Metropolis,[37] Householder suggested to return in such a situation an unevaluated product as a pair of floats; We did not consider that solution as it would have complexified the kind of output managed by the function.

Case 7 is different: we may get an underflow or an overflow when computing $-c/a$ while the square root would have brought back the result into the representable domain. To avoid that, we will exploit our knowledge of the representation of IEEE 754 floating-point numbers. We have:

$$\sqrt{-\frac{c}{a}} = \sqrt{-\frac{m_c 2^{E_c}}{m_a 2^{E_a}}} = \sqrt{-\frac{m_c 2^{E_c - E_a}}{m_a}}$$

Given M and E two integers (with $E \in \{0, 1\}$) such that:

$$E_c - E_a = 2M + E$$

We may then write:

$$\sqrt{-\frac{m_c 2^{E_c - E_a}}{m_a}} = \sqrt{-\frac{m_c 2^{2M+E}}{m_a}} = 2^M \sqrt{-\frac{m_c 2^E}{m_a}} \tag{5}$$

This is an error-free transformation.

The fraction below the square root cannot trigger any underflow or overflow anymore. In order to implement that expression, the programming language must allow isolating the signed significand and the exponent of a floating-point number, which is the case for many languages, particularly those that inherit most of their library from C. However, we have to be wary of the actual value of the exponent reported by the dedicated function. For subnormal numbers, the `exponent()` function in Julia will report a value smaller than $E_{\min}$ as it always considers the significand as normalized. As a consequence, the value of M may be outside the domain $[E_{\min}, E_{\max}]$, which would lead 2^M to underflow or overflow; in the case of an overflow, for example, if the square root is less than 1, the result might still be representable. In order to avoid that situation, we set $M = M_1 + M_2$ with $(M_1, M_2) \in [E_{\min}, E_{\max}]^2$, and we compute Expression 5 as:

$$2^M \sqrt{-\frac{m_c 2^E}{m_a}} = 2^{M_1} \left(2^{M_2} \sqrt{-\frac{m_c 2^E}{m_a}} \right) \tag{6}$$

using the function `keep_exponent_in_check()` in Program 2.

5.3 | Pitfall 3: Cancellation in computing Δ

The computation of the discriminant Δ according to the textbook formula:

$$\Delta = b^2 - 4ac$$

Program 2: Ensuring that an exponent is always in the domain $[E_{\min}, E_{\max}]$.

```
function keep_exponent_in_check(Kin)
    if -1022 <= Kin <= 1023
        return (Kin, 0)
    elseif Kin < -1022
        return (-1022, Kin+1022)
    else
        return (1023, Kin-1023)
    end
end
```

entails the subtraction of two computed values, which leads to a possibility of catastrophic cancellation. Consider, for example, our introductory example:

$$x^2 + (1 + 2^{-52})x + (\frac{1}{4} + 2^{-53}) = 0$$

We get:

$$\Delta = (1 + 2^{-52})^2 - 4(\frac{1}{4} + 2^{-53}) = 2^{-104}$$

Unfortunately, $b^2 = 1 + 2^{-104} + 2^{-51}$ cannot be represented with double precision, and will be rounded to $\mathrm{fl}\left(b^2\right) = 1 + 2^{-51}$. Hence:

$$\mathrm{fl}\langle\Delta\rangle = (1 + 2^{-51}) - (1 + 2^{-51}) = 0$$

That small rounding error leads us to misidentify the number of solutions.

In 2004, Kahan[13] proposed an algorithm to accurately compute the discriminant by taking advantage of the fact that the error performed when multiplying two floats is itself a representable float.[38] It is therefore possible to collect those errors and to reinject them into the computation. Program 5 presents a version of Kahan's algorithm that takes advantage of that when no `fma`[4] instruction is available. That version requires two auxiliary functions:

- A function `veltkamp_split(x)` (Program 3) to separate a float `x` into two non-overlapping floats `xhi` and `xlo` (with $\mathtt{x} = \mathtt{xhi} + \mathtt{xlo}$);
- A function `exactmult(x,y,pxy)` (Program 4), which computes the error $\mathtt{e} = \mathtt{x} \times \mathtt{y} - \mathrm{fl}(\mathtt{x} \times \mathtt{y})$.

Program 3: Splitting a double precision float into two non-overlapping parts.

```
function veltkamp_split(x)
    gamma = 134217729*x # 134217729 = 2^((52/2+1))+1
    delta = x - gamma
    xhi = gamma + delta
    xlo = x - xhi
    return (xhi,xlo)
end
```

When an `fma` instruction is available, a shorter algorithm exists, shown in Program 6. Note also that, when $b^2 \not\approx 4ac$, the straight computation $\mathrm{fl}\left\langle b^2 - 4ac \right\rangle$ is accurate enough.

Given $\hat{\Delta}$, the value of the discriminant computed by Kahan's algorithm, Boldo[27] proved that:

$$\left|\Delta - \hat{\Delta}\right| \leqslant 2\,\mathrm{ulp}\left(\hat{\Delta}\right)$$

provided no underflow or overflow occurs. Jeannerod et al.[28] refined that bound further.

We will see in the next section that we will always perform the computation of the discriminant in the absence of underflow and overflow.

[4] An `fma` instruction is an instruction on three values x, y, z that can perform the operation $xy + z$ with only one rounding error.

Program 4: Computation of the error in the multiplication of two floats.

```
function exactmult(x,y,pxy)
    (xhi,xlo) = veltkamp_split(x)
    (yhi,ylo) = veltkamp_split(y)
    t1 = -pxy + xhi*yhi
    t2 = t1 + xhi*ylo
    t3 = t2 + xlo*yhi
    e = t3 + xlo*ylo
    return e
end
```

Program 5: Accurate computation of the discriminant with Kahan's method[13] without an `fma` instruction.

```
function kahan_discriminant(a,b,c)
    d = b*b - 4*a*c
    if 3*abs(d) >= b*b + 4*a*c # b^2 and 4ac are different enough?
        return d
    end
    p = b*b
    q = 4*a*c
    dp = exactmult(b,b,p)
    dq = exactmult(4*a,c,q)
    d = (p-q) + (dp-dq)
    return d
end
```

Program 6: Accurate computation of the discriminant with Kahan's method[13] with an `fma` instruction.

```
function kahan_discriminant_fma(a,b,c)
    d = b*b - 4*a*c
    if 3*abs(d) >= b*b + 4*a*c # b^2 and 4ac are different enough?
        return d
    end
    p = b*b
    dp = fma(b,b,-p)
    q = 4*a*c
    dq = fma(4*a,c,-q)
    d = (p-q) + (dp-dq)
    return d
end
```

5.4 | Pitfalls 4 & 5: Underflows/overflows and cancellation in computing the solutions

When computing the discriminant, b^2, $4ac$, or $b^2 - 4ac$, even, when $ac < 0$, may underflow or overflow while the solutions are perfectly representable. In order to avoid it, we take advantage of the fact that scaling by a factor does not change the solutions. Starting from:

$$ax^2 + bx + c = 0,$$

we use, once again, our knowledge of the representation of floats to express it as:

$$m_a 2^{E_a} x^2 + m_b 2^{E_b} x + m_c 2^{E_c} = 0.$$

The following scaled equation has the same solutions:

$$2^{E_a - 2E_b}\left(m_a 2^{E_a} x^2 + m_b 2^{E_b} x + m_c 2^{E_c}\right) = 0.$$

We perform a change of variable:

$$x = 2^{E_b - E_a} y.$$

We get:

$$\begin{aligned} & 2^{E_a-2E_b}\left(m_a 2^{E_a} x^2 + m_b 2^{E_b} x + m_c 2^{E_c}\right) = 0 \\ \iff & 2^{E_a-2E_b}\left(m_a 2^{2E_b-E_a} y^2 + m_b 2^{2E_b-E_a} y + m_c 2^{E_c}\right) = 0 \\ \iff & m_a y^2 + m_b y + m_c 2^{E_c+E_a-2E_b} = 0 \end{aligned}$$

Let us now consider the new equation:

$$m_a y^2 + m_b y + m_c 2^{E_c+E_a-2E_b} = 0,$$

where only the constant term may be the source of an underflow or overflow. Once we get the solutions for that equation, we will perform a new change of variable from y to x to get the solutions sought.

When computing the discriminant, the term $4m_a m_c 2^{E_c+E_a-2E_b}$ is the only one that can underflow or overflow. Let us consider the three separate cases:

- If $4m_a m_c 2^{E_c+E_a-2E_b} \in [2^{E_{\min}}, 2^{E_{\max}+1})$, we have neither underflow nor overflow;
- If $4m_a m_c 2^{E_c+E_a-2E_b} < 2^{E_{\min}}$, we have an underflow;
- If $4m_a m_c 2^{E_c+E_a-2E_b} \geqslant 2^{E_{\max}+1}$, we have an overflow.

Knowing that $|m_a|$ and $|m_c|$ take their value in $[2^{1-p}, 2)$, we deduce:

- $E_c + E_a - 2E_b \in [E_{\min} + 2p - 4, E_{\max} - 3)$: neither underflow, nor overflow;
- $E_c + E_a - 2E_b < E_{\min} + 2p - 4$: underflow when computing the discriminant;
- $E_c + E_a - 2E_b \geqslant E_{\max} - 3$: overflow when computing the discriminant.

However, if we use the function `veltkamp_split()` from Program 3 to compute the discriminant, we have to take into account the multiplication by $2^{\lfloor p/2 \rfloor + 1} + 1$ it performs. To avoid an overflow, we then also require:

$$(2^{\lfloor p/2 \rfloor + 1} + 1) m_c 2^{E_c+E_a-2E_b} < 2^{E_{\max}+1}.$$

To simplify, we choose a tighter bound and replace $2^{\lfloor p/2 \rfloor + 1} + 1$ by $2^{\lfloor p/2 \rfloor + 2}$. We get:

$$(2^{\lfloor p/2 \rfloor + 2}) m_c 2^{E_c+E_a-2E_b} < 2^{E_{\max}+1},$$

which gives:

$$E_c + E_a - 2E_b < E_{\max} - 2 - \left\lfloor \frac{p}{2} \right\rfloor$$

as our new upper bound to avoid an overflow.

Case $E_c + E_a - 2E_b \in [E_{\min} + 2p - 4, E_{\max} - 2 - \left\lfloor \frac{p}{2} \right\rfloor)$.
The discriminant of the equation:

$$\underbrace{m_a}_{a'} y^2 + \underbrace{m_b}_{b'} y + \underbrace{m_c 2^{E_c+E_a-2E_b}}_{c'} = 0,$$

can be computed without any risk of underflow or overflow using Kahan's algorithm. If it is negative—no real solution—we return the pair `(NaN,NaN)`; if it is zero, we return $(-b'/(2a')) \times 2^{E_b - E_a}$. The range of a' and b' ensure that we cannot have any spurious overflow or underflow from the division.

If the discriminant is positive, we have a risk of cancellation when computing one of the solutions. As seen in the previous section, we can avoid it by using the modified Fagnano's formulas:

$$\begin{cases} y_1 = \dfrac{-b' - \operatorname{sign}(b')\sqrt{b'^2 - 4a'c'}}{2a'} \\ y_2 = -\dfrac{2c'}{b' + \operatorname{sign}(b')\sqrt{b'^2 - 4a'c'}} \end{cases}$$

Then, we get:

$$\begin{cases} x_1 = y_1 2^{E_b - E_a} \\ x_2 = y_2 2^{E_b - E_a} \end{cases}.$$

As we have seen for Case 7 in Table 2, $E_b - E_a$ may be outside the domain $[E_{\min}, E_{\max}]$. We use the same workaround as in Equation 6 and use the expressions:

$$\begin{cases} x_1 = (y_1 2^{\Omega}) 2^{\Gamma} \\ x_2 = (y_2 2^{\Omega}) 2^{\Gamma} \end{cases},$$

with $\Omega + \Gamma = E_b - E_a$ and $(\Omega, \Gamma) \in [E_{\min}, E_{\max}]^2$. The same must also be done when $\Delta = 0$.

Case $E_c + E_a - 2E_b < E_{\min} + 2p - 4$.

The computation of $4a'c'$ underflows. Consequently, we have $\sqrt{\Delta} \approx b'$. We may compute the first solution as:

$$y_1 = -\frac{b'}{a'}$$

To get the second solution, we may reuse the scaling as in Equation (5) and express c' as:

$$c' = m_c 2^{2M+E} = 2^{2M} \underbrace{m_c 2^E}_{c''} \tag{7}$$

with M and E two integers such that $E_c + E_a - 2E_b = 2M + E$ and $E \in \{0, 1\}$. Using Viète's formula, we obtain:

$$y_2 = \frac{c''}{a' y_1}$$

Once again, we get our original solutions by rescaling:

$$\begin{cases} x_1 = y_1 2^{E_b - E_a} \\ x_2 = y_2 2^{2M + E_b - E_a} \end{cases}$$

Again, the exponent $E_b - E_a$ and $2M + E_b - E_a$ may be outside $[E_{\min}, E_{\max}]$. We ensure the change of variable is done without spurious underflow or overflow by splitting these exponents into two if necessary, as above.

Case $E_c + E_a - 2E_b \geqslant E_{\max} - 2 - \left\lfloor \frac{p}{2} \right\rfloor$.

We cannot solve the equation:

$$a' y^2 + b' y + c' = 0,$$

directly since the discriminant will overflow on computing $4a'c'$ or when calling `veltkamp_split()` on c'. If $ac > 0$, we have $b'^2 - 4a'c' < 0$ and there are no real solutions. If $ac < 0$, the value of $4a'c'$ is so large that we may ignore the contribution of b'^2 and b'. Therefore:

$$y_{1,2} = \frac{\pm\sqrt{-4a'c'}}{2a'} = \pm\sqrt{\left|\frac{c'}{a'}\right|} = \pm 2^M \sqrt{\left|\frac{c''}{a'}\right|}$$

The values of c'' and c''/a' may be computed without any risk of overflow, which was not necessarily the case for c' and c'/a'. We rescale to get our original solutions:

$$\begin{cases} x_1 = y_1 2^{E_b - E_a} \\ x_2 = -x_1 \end{cases}$$

As previously, the exponent $E_b - E_a$ is split into two if necessary to ensure no spurious underflow or overflow.

Program 7 is the complete Julia function that embodies all the formulas from this section. We call it "`sterbenz()`" as it is largely inspired from Sterbenz's notes with Kahan's algorithm used to accurately compute the discriminant. The `sign1()` function that appears in its code implements the sign () function from Equation (2).

6 | TESTS

Kahan[13] proposed to test quadratic equation solvers by considering equations of the form:

$$M F_n x^2 - 2M F_{n-1} x + M F_{n-2} = 0,$$

Program 7: A robust quadratic equation solver using Sterbenz's notes[4], adapted to double precision IEEE 754 floats, and Kahan's algorithm[13] to compute the discriminant.

```
function sterbenz(a,b,c)
  try
    a=Float64(a);b=Float64(b);c=Float64(c)
  catch e
    println(stderr,"Parameters should be numbers!")
    return nothing
  end
  if isnan(a) || isnan(b) || isnan(c) ||
     isinf(a) || isinf(b) || isinf(c)
      return (NaN64)
  end
  # Degenerate cases
  if a == 0
     if b == 0
        if c == 0 # a==0, b==0, c==0
           return (Inf64, -Inf64)
        else      # a==0, b==0, c!=0
           return (NaN64)
        end
     else
        if c == 0 # a==0, b!=0, c==0
           return (0.0)
        else      # a==0, b!=0, c!=0
           return (-c/b)
        end
     end
  else
     if b == 0
        if c == 0 # a!=0, b==0, c==0
           return (0.0)
        else      # a!=0, b==0, c!=0
           if sign1(a) == sign1(c)
              return (NaN64, NaN64)
           else
              ea = exponent(a)
              ec = exponent(c)
              ecp = ec - ea
              a2 = significand(a)
              dM = ecp & ~1 # dM = floor(ecp/2)*2
              M = dM>>1     # M = dM/2
              E = ecp & 1   # E = odd(ecp) ? 1 : 0
              c3 = significand(c)*2.0^(E)
              S = sqrt(-c3/a2)
              (M1,M2) = keep_exponent_in_check(M)
              x1 = (S*2.0^M2)*2.0^M1
              return (-x1,x1)
           end
        end
     else
        if c == 0 # a!=0, b!=0, c==0
           if sign1(a) == sign1(b)
              return (-b/a, 0.0)
           else
              return (0.0, -b/a)
           end
        else      # a!=0, b!=0, c!=0
           ea = exponent(a)
           eb = exponent(b)
           ec = exponent(c)
           K = eb - ea
           L = ea - 2*eb
           ecp = ec + L
           a2 = significand(a)
           b2 = significand(b)
           if ecp >= -920 && ecp < 995
              c2 = significand(c)*2.0^ecp
              delta = kahan_discriminant_fma(a2,b2,c2)
              if delta < 0
                 return (NaN64, NaN64)
              end
              (K1,K2) = keep_exponent_in_check(K)
              if delta > 0
                 y1 = -(2*c2)/(b2+sign1(b)*sqrt(delta))
                 y2 = -(b2+sign1(b)*sqrt(delta))/(2*a2)
                 x1 = (y1*2.0^K2)*2.0^K1
                 x2 = (y2*2.0^K2)*2.0^K1
                 return (min(x1,x2),max(x1,x2))
              end
              return  ((-b2/(2*a2))*2.0^K2)*2.0^K1
           end
           dM = ecp & ~1 # dM = floor(ecp/2)*2
           M = dM>>1     # M = dM/2
           E = ecp & 1   # E = odd(ecp) ? 1 : 0
           c3 = significand(c)*2.0^(E)
           S = sqrt(abs(c3/a2))
           if ecp < -920
              y1 = -b2/a2
              y2 = c3/(a2*y1)
              (dMK1, dMK2) = keep_exponent_in_check(dM+K)
              (K1,K2) = keep_exponent_in_check(K)
              x1 = (y1*2.0^K2)*2.0^K1
              x2 = (y2*2.0^dMK2)*2.0^(dMK1)
              return (min(x1,x2),max(x1,x2))
           end
           # ecp >= 995
           if sign1(a) == sign1(c)
              return (NaN64, NaN64)
           else
              (MK1,MK2) = keep_exponent_in_check(M+K)
              x1 = (S*2.0^MK2)*2.0^(MK1)
              x2 = -x1
             return (x2, x1)
           end
        end
     end
  end
end
```

where F_n is the n-th element of the Fibonacci sequence (starting from F_0), and $M = \lfloor R/F_n \rfloor$, with R an integer drawn uniformly at random from $[2^{p-1}, 2^p - 1]$. The solutions are:

$$\frac{F_{n-1} \pm \sqrt{(-1)^n}}{F_n}.$$

In order to consider only equations with parameters that are representable with double precision floats, we cannot exceed F_{76}. To get real solutions, we also consider even values of n only.

Table 3 summarizes the results for all the methods presented in Sections 3 to 5. Out of twenty algorithms, almost half of them either compute the wrong number of solutions, or compute the wrong solutions. When a method computes only one result, it

Table 3 Results on the 38 randomized Fibonacci-based quadratic equations from F_2 to F_{76}.

Name	Wrong Number of solutions	Wrong solutions
Baker	0	1
Beebe	0	0
Boost	0	0
Dahlquist/Björck	0	0
Einarsson	0	0
GSL	0	0
Higham	0	0
Jenkins	0	16
Kahan	0	0
Mastronardi/van Dooren	0	0
Naive	18	0
Nievergelt	0	1
Nonweiler	0	0
Numerical recipes	0	0
Panchekha (PLDI15)	0	0
Panchekha (racket)	0	37
Rust	18	0
Scilab	18	0
Sterbenz	0	0
Young/Gregory	0	19

is counted as a wrong solution only if the value does not correspond to any of the correct solutions. In our implementation, a result $\widehat{x}$ is considered correct with respect to a solution x if:

$$|\widehat{x} - x| \leqslant \sqrt{\epsilon}\max\left(|\widehat{x}|, |x|\right) \tag{8}$$

The randomized Fibonacci-based tests advocated by Kahan[13] are difficult to solve correctly but not so challenging after all.

We have also defined a set of difficult quadratic equations that exercise the algorithms at their weakest points. All these tests, and more, are present in our Julia `QuadraticEquation` package available on github. There are five categories:

- Fifteen quadratics with no solution, not even complex ones (an NaN or an infinity is among the parameters);
- Four quadratics with no real solution;
- Seventeen degenerate quadratics (at least one parameter is zero);
- One quadratic with one solution;
- Eighteen quadratics with two real solutions.

Table 4 presents the results for all categories with the format x/y/z, where "x" is the number of correct results, "y" the number of cases where the solver found the wrong number of solutions, and "z" the number of cases where the solutions were wrong. If a solver finds the wrong number of solutions but at least one result is correct, it is counted in the first category of error but not in the second. A result is considered correct with respect to a solution following the same definition as Equation (8). Notice that, now, only Function `sterbenz()` keeps a perfect score.

Lastly, we generated one million quadratics randomly. The parameters are drawn among *all* finite floating-point numbers—even subnormals—in such a way that the resulting quadratic has two real solutions. We computed the solutions with arbitrary precision in order to be able to determine the size of the error for the results from each of our twenty methods. Table 5 reports the largest relative error overall. Unfortunately, since many methods do not handle underflows and overflows correctly, most results are distressingly large (when the result was an infinite value, the error was not considered, however). We may note that

Table 4 Correctness of the various algorithms on challenging quadratics

Name	No sol.	No real sol.	Degenerate	One sol.	Two sol.
Baker	0/0/15	4/0/0	2/0/15	0/1/0	16/0/2
Beebe	14/0/1	4/0/0	6/10/1	0/1/0	14/0/4
Boost	1/0/14	4/0/0	13/3/2	0/1/1	13/0/5
Dahlquist/Björck	0/0/15	4/0/0	3/10/11	0/1/1	14/0/4
Einarsson	0/0/15	4/0/0	4/10/5	0/1/1	14/0/4
GSL	1/0/14	4/0/0	13/3/2	0/1/1	13/0/5
Higham	0/0/15	4/0/0	3/10/11	0/1/1	14/0/4
Jenkins	0/0/15	4/0/0	6/10/1	0/1/0	17/0/1
Kahan	0/0/15	4/0/0	3/10/7	0/1/1	13/0/5
Mastronardi/van Dooren	0/0/15	1/0/3	6/2/9	0/1/1	14/0/4
Naive	1/0/14	4/0/0	5/9/11	0/1/1	10/2/7
Nievergelt	0/0/15	1/0/3	0/0/17	0/1/0	13/1/5
Nonweiler	0/0/15	4/0/0	7/1/9	1/0/0	13/3/4
Numerical recipes	0/0/15	4/0/0	4/10/5	0/1/1	14/0/4
Panchekha (PLDI15)	0/0/15	4/0/0	5/10/6	0/1/1	13/0/5
Panchekha (racket)	0/0/15	4/0/0	6/10/6	0/1/1	15/0/3
Rust	1/0/14	4/0/0	12/2/4	0/1/1	13/2/4
Scilab	0/0/15	4/0/0	7/9/1	1/0/0	16/2/1
Sterbenz	15/0/0	4/0/0	17/0/0	1/0/0	18/0/0
Young/Gregory	1/0/14	1/0/3	14/2/1	0/1/1	13/1/5

the algorithm used in the Racket language and the `sterbenz()` function exhibit both a good worst relative error and solve correctly all quadratics. This also shows that testing randomly a procedure may not be the best way to ascertain its qualities as it is still possible to devise tests that show flaws in Racket's algorithm (See Tables 3 and 4, for example).

To better assess the precision achieved by the `sterbenz()` procedure, we also tracked the worst error for 200 000 000 random quadratics. The largest relative error was around 1.76ϵ, a bit more than Forsythe's wishes,[12] but quite satisfactory nonetheless.

7 | CONCLUSION

Compare Program 1 and Program 7: one is 17 lines long and the other, 128 lines (if we count the ancillary functions `keep_exponent_in_check()` and `kahan_discriminant_fma()`); one must be used with great care and a restricted set of parameters—lest it return false results—, the other can take any parameter thrown at it and return precise solutions when they can be represented as floating-point numbers.

Incredibly elaborate numerical software is built from layer upon layer of ever simpler code. The simplest code, such as a quadratic equation solver, should be flawless so that, as Forsythe[1] puts it, "[...]*when a quadratic equation occurs in the midst of a complex and imperfectly understood computation, one can be sure that the quadratic equation solver can be relied upon to do its part well and permit us to concentrate attention on the rest of the computation.*"

The devising of Program 7 requires quite a good knowledge of the properties of IEEE 754 floating-point numbers. Should we expect that from all the programmers who implement numerical algorithms into programming language libraries and applications? Probably not, and this is why those who devise these algorithms in the first place should not leave any detail out when presenting them.

It is rather disquieting to compare what has been available in the literature for more than half a century—some of the algorithms published being quite good, if not perfect—with what is currently implemented in software. It is possible that the format chosen for the exposition of their algorithm by experts in numerical methods is often not the most suited to appeal to those who have to

Table 5 Worst precision for 1 000 000 random quadratics with two real solutions.

Name	Wrong solutions	Overflows	Worst relative precision
Baker	36	11	$14.43 \times \varepsilon$
Beebe	0	93804	$4.5 \times 10^{15} \times \varepsilon$
Boost	0	85388	$1.8 \times 10^{161} \times \varepsilon$
Dahlquist/Björck	0	244574	$1.8 \times 10^{161} \times \varepsilon$
Einarsson	0	244683	$1.8 \times 10^{161} \times \varepsilon$
GSL	0	244869	$3.87 \times 10^{302} \times \varepsilon$
Higham	0	244574	$1.8 \times 10^{161} \times \varepsilon$
Jenkins	0	26019	$4.5 \times 10^{15} \times \varepsilon$
Kahan	0	295388	$7.7 \times 10^{302} \times \varepsilon$
Mastronardi/van Dooren	0	196961	$4.4 \times 10^{15} \times \varepsilon$
Naive	33640	244574	$5.5 \times 10^{296} \times \varepsilon$
Nievergelt	0	243559	$4.7 \times 10^{161} \times \varepsilon$
Nonweiler	79013	223242	$4.5 \times 10^{15} \times \varepsilon$
Numerical recipes	0	244869	$1.8 \times 10^{161} \times \varepsilon$
Panchekha (PLDI15)	0	178856	$3.8 \times 10^{302} \times \varepsilon$
Panchekha (racket)	0	0	$2.29 \times \varepsilon$
Rust	33640	109	$3.1 \times 10^{155} \times \varepsilon$
Scilab	33839	71717	$8.4 \times 10^{306} \times \varepsilon$
Sterbenz	0	0	$1.52 \times \varepsilon$
Young/Gregory	194605	30596	$4.5 \times 10^{15} \times \varepsilon$

implement them. This rift has to be mended somehow if we expect the best numerical methods to drive out the bad ones from numerical software eventually.

This is not the end of the story, though. There is one problem that Program 7 cannot prevent, and it happens when the parameters of the quadratic equation are approximations themselves. This can occur because they are obtained from a previous computation, or because they are provided by the user as decimal values or expressions. Consider for example the equation:

$$x^2 + \sqrt{\frac{4}{5}}x + \frac{1}{5} = 0$$

Its discriminant is clearly zero, and the only solution should be $-\sqrt{1/5}$. Unfortunately, the parameters $\sqrt{4/5}$ and $1/5$ have to be rounded to be represented as floating-point numbers. Therefore, the actual equation that is being solved is not the one the user intended, and even Program 7 will not get the expected solutions: it will return that there are no real solutions, which is true of the equation really considered:

$$x^2 + \mathrm{fl}\left\langle\sqrt{\frac{4}{5}}\right\rangle x + \mathrm{fl}\left(\frac{1}{5}\right) = 0$$

One solution to this problem may be to use interval arithmetic[39] to handle such cases by enclosing the real parameters into intervals with floating-point bounds.

The code for all the algorithms presented in this article has been assembled into a Julia package, `QuadraticEquation`, which is available on github. The package also contains challenging tests for the various solvers.

ACKNOWLEDGMENTS

Discussions with Dr. Christophe Jermann helped shape this article as it currently is. An invitation to present computer arithmetic pitfalls to high school French mathematics teachers by Pr. Magali Hersant and Dr. Emmanuel Desmontils, co-directors of the

Institute for Research on Teaching Mathematics (IREM des Pays de la Loire), was the initial impetus for the research presented therein.

References

1. Forsythe GE. How do you solve a quadratic equation?. Technical Report CS-TR-66-40, Computer Science Department; Stanford University: 1966.
2. Muller DE. A Method for Solving Algebraic Equations Using an Automatic Computer. *Mathematical Tables and Other Aids to Computation* 1956; 10(56): 208–215. Publisher: American Mathematical Society.
3. IEEE Standard for Floating-Point Arithmetic. Tech. Rep. IEEE Std 754-2019 (Revision of IEEE 754-2008), Institute of Electrical and Electronics Engineers; Piscataway, New Jersey, United States: 2019.
4. Sterbenz PH. *Floating-point computation*. Prentice-Hall . 1974.
5. Muller JM, Brisebarre N, Dinechin F, et al. *Handbook of floating-point arithmetic*. Birkhäuser . 2010.
6. Smith DE. *General survey of the history of elementary mathematics*. New York, NY: Dover . 1980.
7. Duncan FG, Huxtable DHR. The DEUCE Alphacode Translator. *The Computer Journal* 1960; 3(2): 98–107.
8. Metropolis NC. Analyzed Binary Computing. *IEEE Transactions on Computers* 1973; C-22(6): 573–576.
9. Smith RL. Algorithm 116: Complex division. *Communications of the ACM* 1962; 5(8): 435.
10. Goff GK. The Citardauq Formula. *The Mathematics Teacher* 1976; 69(7): 550–551. Publisher: National Council of Teachers of Mathematics.
11. Fagnano GCd. *Produzioni matematiche del conte Giulio Carlo di Fagnano, marchese de'Toschi, e di Sant'Onorio, nobile romano, e patrizio senogagliese* . 1750.
12. Forsythe GE. What is a satisfactory quadratic equation solver?. Technical Report CS74, Computer Science Department; Stanford University: 1967.
13. Kahan W. On the cost of floating-point computation without extra precise arithmetic. http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf; 2004.
14. CASIO . *Computing with the scientific calculator*. CASIO . 1986.
15. *Numworks User Manual 20.0.0*. 2023. https://www.numworks.com/resources/manual/.
16. Nonweiler TRF. Roots of low-order polynomial equations. *Communications of the ACM* 1968; 11(4): 269–270.
17. Vietæ F. *Opera Mathematica*. Bonaventuræ & Abrahami Elzeviriorum . 1646.
18. Jenkins MA. Algorithm 493: Zeros of a Real Polynomial [C2]. *ACM Transactions on Mathematical Software* 1975; 1(2): 178–189.
19. Franklin P. Basic Mathematical Formulas. In: Menzel DH., ed. *Fundamental formulas of physics*. 1. Dover. 1977 (pp. 1–106).
20. Hamming RW. *Numerical methods for scientists and engineers*. New York: Dover. 2nd ed. 1986.
21. Young DM, Gregory RT. *A survey of numerical mathematics*. New York: Dover Publications . 1988.
22. Baker HG. You could learn a lot from a quadratic: overloading considered harmful. *ACM SIGPLAN Notices* 1998; 33(1): 30–38.

23. Higham NJ. *Accuracy and stability of numerical algorithms*. Philadelphia: Society for Industrial and Applied Mathematics. 2nd ed. 2002.

24. Nievergelt Y. How (Not) to Solve Quadratic Equations. *The College Mathematics Journal* 2003; 34(2): 90–104.

25. Edelman A, Murakami H. Polynomial roots from companion matrix eigenvalues. *Mathematics of Computation* 1995; 64(210): 763–776.

26. Boldo S, Daumas M, Kahan W, Melquiond G. Proof and certification for an accurate discriminant. Presentation at SCAM'2006: 12th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics; 2006.

27. Boldo S. Kahan's Algorithm for a Correct Discriminant Computation at Last Formally Proven. *IEEE Transactions on Computers* 2009; 58(2): 220–225.

28. Jeannerod CP, Louvet N, Muller JM. Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants. *Mathematics of Computation* 2013; 82(284): 2245–2264.

29. Einarsson B. *Accuracy and Reliability in Scientific Computing*. Society for Industrial and Applied Mathematics . 2005.

30. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. *Numerical recipes: the art of scientific computing*. Cambridge, New York: Cambridge University Press. 3rd ed. 2007.

31. Dahlquist G, Björck o. *Numerical Methods in Scientific Computing, Volume I*. Society for Industrial and Applied Mathematics . 2008.

32. McNamee JM, Pan VY. Chapter 12 - Low-Degree Polynomials. In: . 16 of *Numerical Methods for Roots of Polynomials - Part II*. Elsevier. 2013 (pp. 527–556).

33. Panchekha P, Sanchez-Stern A, Wilcox JR, Tatlock Z. Automatically improving accuracy for floating point expressions. In: PLDI '15. Association for Computing Machinery; 2015; New York, NY, USA: 1–11.

34. Mastronardi N, Dooren vP. Revisiting the stability of computing the roots of a quadratic polynomial. *Electronic Transactions on Numerical Analysis* 2015; 44: 73–82.

35. Beebe NHF. *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*. Springer International Publishing . 2017.

36. Scilab . Scilab is not naive. Technical Report, Dassault systèmes; Vélizy-Villacoublay, France: 2010. https://www.semanticscholar.org/paper/Scilab-Is-Not-Naive/c2e7b5c928eb18608d438e3b7a84142080d56164.

37. Gardiner V, Metropolis N. A comprehensive approach to computer arithmetic. Research Report LA-4531, Los Alamos Scientific Laboratory of the University of California; Los Alamos, New Mexico: 1970.

38. Dekker TJ. A floating-point technique for extending the available precision. *Numerische Mathematik* 1971; 18(3): 224–242.

39. Elishakoff I, Daphnis A. Exact enclosures of roots of interval quadratic equations by Sridhara's and Fagnano's or modified Fagnano's formulas. *Applied Mathematics and Computation* 2015; 271: 1024–1037.