CS40

HOW DO YOU SOLVE A QUADRATIC EQUATION?

BY

GEORGE E. FORSYTHE

TECHNICAL REPORT NO. CS40
JUNE 16, 1966

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

HOW DO YOU SOLVE A QUADRATIC EQUATION?

by

George E. Forsythe

Abstract

The nature of the floating-point number system of digital computers is explained to a reader whose university mathematical background is very limited. The possibly large errors in using mathematical algorithms blindly with floating-point computation are illustrated by the formula for solving a quadratic equation. An accurate way of solving a quadratic is outlined. A few general remarks are made about computational mathematics, including the backwards analysis of rounding error.

1. Stages of scientific computation.

The automatic digital computer is one of man's most powerful intellectual tools. It forms an extension of the human mind that can only be compared with the augmentation of human muscle provided by the most powerful engines in the world.

Computers are used in a wide variety of applications, ranging from the control of artificial satellites to the automatic justification and hyphenation of English prose, and even to the storage and searching of vast libraries of medical literature. However, computers were originally invented with the sole aim of permitting arithmetical computations to be carried out rapidly and accurately, and this remains one of the major uses of computers today.

For example, as early as World War I, L. F. Richardson had indicated how the weather might be forecast with the aid of a vast computation then

---

Prepared under Contract Nonr-225(37)(NR-044-211) at Stanford University.

far beyond human capability, provided that enough upper-air weather observations were available as input data. By the 1940's the upper-air observations were beginning to appear in quantity, Hence, when John von Neumann and others asked the Government for funds to support computer development, they promised that computers would make it possible to carry out the arithmetical part of a modern version of Richardson% program. It was expected that the weather would soon be forecast routinely by computer, and this has occurred, It was even hinted that computers might make it possible to predict, for example, the future course of hurricanes after a variety of human interventions, and thus lead to theeventual control of the weather!

There are many intellectual steps involved in a project like weather forecasting by computer. In the first place, a reasonable model of the weather must be reduced to systems of equations, both algebraic and differential, The actual solution of such systems of equations is completely beyond the powers of any computer, because the equations involve the infinite number of variables needed to represent, for example, the wind at each of an infinite number of points of space.

Consequently, the second stage of numerical weather prediction is to replace the actual meteorological equations by a finite number of equations, This is done by first replacing the infinite number of points of space by a finite number of points arranged in a cubical mesh which looks like a number of huge coarse screens spaced above each other, made of squares perhaps 100 miles square, Instead of trying to describe and predict the wind at each point of space, one describes and predicts it at the points of the mesh (i.e. the corners of the squares of the screens). The equations which describe the exact flow of air and moisture are replaced by much simpler equations which relate these quantities at neighboring points of the mesh. A great deal of mathematical analysis and experimental computation are needed before one can discover simple equations for the mesh which in fact reasonably well simulate the actual equations for continuous space. This is a subject that has interested mathematicians very much. It is, however, much too difficult and technical for discussion here.

At the end of the second stage just described, we have a finite number of equations to solve. Each equation deals with unknown quantities which are real numbers. Recall that real numbers may be thought of as infinite decimal expansions like

-3.3333333333 3333333333 3333333333 .....

(3's continued without end), or

3.1415926535 8979323846 2643383279 .....

(digits continued without end, but without a predictable pattern). Since a computer is necessarily a finite collection of parts, it cannot hold even a single general real number, with its infinite number of decimals. Hence the third stage of the use of computers for weather prediction involves the use of a finite number system, to simulate the real number system of mathematics.

The purpose of this note is to describe this computer number system and some difficulties involved in using it. To illustrate the difficulties we shall consider a mathematical problem that is very much simpler than the equations of meteorology - namely the quadratic equation,

2. Floating-point numbers

We shall first describe a simplified computer number system, the so-called floating-point numbers, and then show some of its behavior with a simple mathematical computation.

The usual number system of a computer reduces the infinite number of decimal places of real numbers to a fixed finite number. We first consider decimal numbers with a sign and one nonzero digit to the left of the decimal point, and exactly seven zero or nonzero numbers to the right of the decimal point. Examples of such numbers are -7.3456780, +1. 0000000, +3 03333333 , -9 . 9808989 . We say that such numbers have 8 significant digits. One can represent approximately 200,000,000 different numbers in this way, but they all lie between -10 and -1, or between +1 and +10 .

To enable computers to hold much bigger and much smaller numbers, we add a sign and two more decimal digits to serve as an exponent of 10 . The exponent is allowed to range from - 50 to + 49 . Thus the number - 87 2/3 is represented by

$$- 8.7666667 \times 10^{+01} .$$

In this system, which is much like so-called scientific notation, the representable numbers all have eight significant decimal digits, They range from

$$-9.9999999 \times 10^{49} \text{ to } - 1.0000000 \times 10^{-50}$$

and from

$$+ 1.0000000 \times 10^{-50} \text{ to } 9.9999999 \times 10^{49} .$$

The number zero is added, and represented by $+ 0.0000000 \times 10^{-50}$. Approximately 20,000,000,000 distinct real numbers are thus representable in the computer, and these take the place of the infinite system of mathematical real numbers.

This computer number system is called the floating-point number system, The "point" is the decimal point. The exponent permits the decimal point effectively to "float" as much as 50 places away (to the left or the right) from its home position.

By special programs it is possible also to use so-called double precision numbers--numbers which have not 8, but 16 significant digits, with the exponent kept between - 49 and + 50 . There is a penalty in time for using these double precision numbers, but this penalty varies greatly among different computers.

In this paper we shall write floating-point numbers in various ways, but there will be understood to be exactly 8 significant digits, For example, we may write the number eleven as 11 or 11.0 or $+ 1.1000000 \times 10^{+01}$ or 1.1×10^{1} .

Actual computers more frequently use number bases other than 10 - for example, 2 or 8 or 16 - and the actual number of significant digits

varies over wide ranges. However, the reader need not be distracted by those considerations; our 8-digit decimal system illustrates all the essential matters very well.

3. Computer arithmetic

Resides holding floating-point numbers, every scientific computer must be able to perform on them the elementary arithmetic operations of addition and subtraction, multiplication and division. Let us consider addition first. Sometimes the exact sum of two floating-point numbers is itself a floating-point number. For example,

$$(+\ 2.1415922 \times 10^{+00}) + (+\ 9.7182818 \times 10^{+00})$$

$$= +\ 1.1859874 \times 10^{+01}\ .$$

In this case, the computed sum is the same as the exact sum, and the computation is said to be without rounding error. More frequently the exact sum is not a floating-point number. For example, the exact sum of $+\ 6.6666667 \times 10^{+01}$ and $+\ 6.6666667 \times 10^{+01}$ is 133.333334, a number with 9 significant digits. Hence the exact sum cannot be held in the computer, but must be rounded to the nearest floating-point number - in this case to $+\ 1.3333333 \times 10^{+02}$. This is a typical example where computer addition is only approximately the same as mathematical addition.

An even worse defect of computer addition appears when the numbers are numerically very large, so that the sum exceeds the capacity of the floating-point system. For example, the true sum of $+\ 9.9900000 \times 10^{+49}$ and $+\ 9.9990000 \times 10^{+49}$ is 1.9989×10^{50}, a number greater than the largest possible floating-point number. The computer should signal in some manner than an overflow has occurred, and give the problem-solving program some option about what action to take. But it is impossible to store an answer which represents the exact sum to even one significant digit.

Analogous effects occur in computer subtraction.

Computer multiplication suffers from the same two defects of computer addition - the necessity for rounding answers, and the possibility of exponent overflow. While ordinary rounding is no more serious than with addition, overflow can be far worse, for the following reason. The exact sum of two floating-point numbers cannot exceed 2×10^{50}, but the exact product of two floating-point numbers can be as large as 9.9999998×10^{99}, and the product of two numbers as small as 10^{25} can lead to overflow, Moreover, there is a possibility of underflow in multiplication. For example, the true product of 1.01×10^{-30} and 1.01×10^{-35} is 1.0201×10^{-65}, a number smaller by a factor of 10^{-15} than the smallest nonzero floating-point number. The most we can expect from the computer is that it replace the product by zero and give the program a signal that underflow has occurred.

Analogous effects occur in computer division

We assume that our computer operations of addition, subtraction, multiplication, and division, in the absence of overflow or underflow, will yield as an answer the floating-point number which is closest to the exact real answer. (In case of a tie, we permit either choice.) In fact many actual computer systems achieve this accuracy, and none give very much less.

4. Are floating-point numbers satisfactory?

Any one who uses a digital computer for scientific computation is faced with a number system which is only approximately that of mathematics, and arithmetic operations which are only approximately those of true addition, subtraction, multiplication, and division, The approximations appear to be very good, being generally correct to less than one unit in the eighth decimal digit. Only the most sophisticated of all scientific and engineering computations (those in optics) deal with numbers accurate to anything close to eight decimal places, We might therefore presume that rounding errors would provide no trouble in most practical computations. Moreover the range of magnitudes from 10^{-50} to 10^{+49} safely covers the range of all important physical and engineering constants, so that we might presume that would have no trouble with overflow or underflow.

Is the floating-point arithmetic system so good that we can use it without fear to simulate the real number system of mathematics? Computer designers certainly hope so, and chose the numbers of significant digits and the exponent range with this expectation.

The answer by now is clear: we may not proceed without fear! There are real difficulties. On the other hand, it is often possible to proceed with intelligence and caution, and get around the difficulties. However, it has required an astonishing amount of mathematical and computer analysis to get around the difficulties, particularly in large problems. And so far we know well how to handle only relatively simple mathematical problems.

We will illustrate some of the difficulties and their solution in the context of an elementary but important problem, the well known quadratic equation of elementary algebra.

5. The quadratic equation

The reader will recall considerable time spent in the ninth grade or thereabouts, finding the two roots of equations like

$$6x^2 + 5x - 4 = 0 \quad . \tag{1}$$

One first acquires some experience in factoring the quadratic. School examples do factor with a frequency bewildering to anyone who has done mathematics outside of school! For example,

$$6x^2 + 5x - 4 = (2x-1)(3x+4) \quad , \tag{2}$$

as the reader could have discovered after some trial and error. If the lefthand side of (2) is to be 0, then either $2x - 1$ or $3x + 4$ must be 0. The two possibilities tell us that the roots of (1) are $1/2$ and $-4/3$.

However, factoring in whole numbers is not always possible, and turns out to be unnecessary. For one soon learns a formula which gives the two roots of any quadratic equation without having to factor anything. The main result is the following, the so-called quadratic formula:

If a, b, and c are any real numbers, and if $a \neq 0$, then the quadratic equation

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

is satisfied by exactly two values of x, namely

(3) $$x_1 = \frac{-b + \sqrt{b^2-4ac}}{2a}$$

and

(4) $$x_2 = \frac{-b - \sqrt{b^2-4ac}}{2a} .$$

As an example of the use of the quadratic formula, the roots of equation (1) are

$$x_1 = \frac{-5 + \sqrt{5^2 - 4(6)(-4)}}{12}$$

$$= \frac{-5 + \sqrt{121}}{12} = \frac{-5 + 11}{12} = \frac{6}{12} = \frac{1}{2} ,$$

$$x_2 = \frac{-5 - \sqrt{121}}{12} = \frac{16}{12} = -\frac{4}{3} .$$

The roots agree with those found by factoring, of course.

The great power of the quadratic formula is that it provides a straightforward series of steps proceeding from the real numbers a, b, c to the solutions x_1, x_2. The steps are those of evaluating the expressions in (3) and (4) in some systematic fashion. The assumption that $a \neq 0$ is necessary to be sure that an illegal division by 0 is not called for.

Any such systematic process for computing some desired answer is called an algorithm. In an algorithm no guesses are allowed--one proceeds directly from the data to the answer The importance of algorithms is that computers have been expressly designed to be able to carry out algorithms and nothing but algorithms. That is to say, the logical steps performed by a computer are exactly those of an algorithm.

Next we give a detailed algorithm for evaluation of the quadratic formula (3). (It could be simplified.)

Algorithm for computing one root x_1 of the quadratic equation $ax^2 + bx + c = 0$.

(i) Compute $z = -b$.
(ii) Compute $y = b^2$.
(iii) Compute $w = 4a$.
(iv) Compute $v = w \cdot c$.
(v) Compute $u = y - v$.
(vi) Compute $t = \sqrt{u}$
(vii) Compute $s = z + t$
(viii) Compute $r = 2a$
(ix) Compute $x_1 = s/r$.

Notes: 1. For simplicity we here assume that $u = b^2 - 4ac$ is not negative, to avoid having to deal with imaginary numbers like $\sqrt{-1}$.

2. An algorithm for computing x_2 requires the replacement of steps (vii) and (ix) by

(vii)' Compute $s' = z - t$.
(ix)' Compute $x_2 = s'/r$.

In mathematics the above algorithm is implicitly understood to use real numbers, and to carry out with them exact arithmetic operations including addition, subtraction, multiplication, division, and even extraction of the square root of $u = b^2 - 4ac$. As we showed in Sec. 3, a computer cannot carry out these exact arithmetic operations and, indeed, cannot even hold arbitrary real numbers a, b, c . Thus, although a real digital computer can carry out the exact logical steps of the algorithm, it must replace all numbers by floating-point numbers, and all arithmetic operations by approximate operations.

The question, then, is this: will the limitations of actual computer floating-point systems make any appreciable difference to their use in solving quadratic equations?

The answer is: sometimes yes, and sometimes no. We shall give examples to illustrate both cases.

6. Examples of the quadratic formula on a computer

Example 1. $6x^2 + 5x - 4 = 0$

For this example of (1), the algorithm of Sec. 5 offers no difficulty for a computer with the precision we have given, except possibly for the square root required in step (vi). Let us make the reasonable assumption that we have a method (indeed, another algorithm) for computing square roots with an error not exceeding 0.8 of a unit in the least significant decimal place. In that case, we will find $t = \sqrt{u} = \sqrt{b^2 - 4ac}$ to be 11.000000 .

Then we find that

$$x_1 = (-5 + 11.000000)/12 = .50000000 ,$$

a perfect result. The computation of x_2 leads to no loss of accuracy until the final division:

$$x_2 = -16.000000/12.000000 = -1.3333333 ,$$

as rounded on the computer. Since this is the correctly rounded value of the true x_2, we conclude that the computer algorithm has done as good a job as it could possibly do.

Example 2. $x^2 - 10^5 x + 1 = 0$.

Before examining the computer solution, we note that the true solutions, rounded to eleven significant decimals, are

$$x_1 = 100000.00001 = 1.0000000001 \times 10^5 ,$$

and

$$x_2 = .0000099999999999 = 9.9999999999 \times 10^{-6} .$$

Moreover, it can be shown that x_1 and x_2 are well determined by the data, in that small changes of the coefficients 1, -10^5, 1 cause only slight changes in x_1 and x_2 .

Now let us apply the algorithm of Sec. 5, and see what are the computed values of x_1 and x_2 .

We have

$$a = 1.0000000 \times 10^{+00}$$
$$b = -1.0000000 \times 10^{+05}$$
$$c = 1.0000000 \times 10^{+00}$$

First, to get x_1:

$$z = -b = 1.0000000 \times 10^{+05}$$
$$y = b^2 = 1.0000000 \times 10^{+10}$$
$$w = 4a = 4.0000000 \times 10^{+00}$$
$$v = w \cdot c = 4.0000000 \times 10^{+00}$$
$$u = y-v = 1.0000000 \times 10^{+10} \quad \text{(see below)}$$
$$t = \sqrt{u} = 1.0000000 \times 10^{+05}$$
$$s = z+t = 2.0000000 \times 10^{+05}$$
$$r = 2a = 2.0000000 \times 10^{+00}$$
$$x_1 = r/s = 1.0000000 \times 10^{+05}$$

The step that calls for comment is the computation of u = y - v, where the value of v is completely lost in rounding the value of u to eight decimals. The final answer x_1 is correct to eight decimals.

We now compute x_2:

$$z = -b = 1.0000000 \times 10^{+05}$$
$$y = b^2 = 1.0000000 \times 10^{+10}$$
$$w = 4a = 4.0000000 \times 10^{+00}$$
$$v = w \cdot c = 4.0000000 \times 10^{+00}$$
$$u = y-v = 1.0000000 \times 10^{+10}$$
$$t = \sqrt{u} = 1.0000000 \times 10^{+05}$$
$$s' = z+t = 0 \quad \text{(see below)}$$
$$r = 2a = 2.0000000 \times 10^{+00}$$
$$x_2 = r/s' = 0$$

This time the computation of s' results in complete cancellation, so s' and hence x_2 are both 0 . Thus our algorithm has yielded a value of x_2 which differs by approximately 10^{-5} from the correct

answer, and this might be considered a rather small deviation. On the other hand, our computed value of x_2 has a relative error of 100 per cent - not a single significant digit is correct! Can this be considered a reasonable computer solution of the quadratic?

A study of ways in which quadratics are applied leads to the conclusion that the measure of accuracy should be that of relative error. As long as a root of a quadratic is well determined by the data, a good algorithm should give it correctly to several or most of its leading digits, however large or small the root may be.

Thus we must conclude that the quadratic formula for x_2 gave us 'practically no useful information about the root x_2. It follows that the algorithms of Sec. 5 are an inadequate way of solving a quadratic equation, because an adequate algorithm must work in every case within its domain of applicability.

Example 3. $6 \times 10^{30} x^2 + 5 \times 10^{30} x - 4 \times 10^{30} = 0$.

The present example is simply that of Example 1, with all its coefficients a, b, c upscaled by the factor 10^{30}. Thus the roots are unchanged.

However, the algorithm of Sec. 5 breaks down at the second step, because $y = b^2$ is truly 2.5×10^{61}, a number outside the range of floating-point numbers. Thus the algorithm of Sec. 5 is again inadequate, though for a very trivial reason. A simple scaling of the coefficients would prevent the overflow.

Example 4. $10^{-30} x^2 - 10^{30} x + 10^{30} = 0$.

Here the true roots are extremely close to 10^{60} and 1 . One of the roots is outside the range of floating-point numbers, and we could hardly expect to get it from a computer algorithm. The problem is: can a reasonable computer algorithm get the root near 1 ?

Note that a simple scaling to make the first coefficient equal to 1 will cause the second and third coefficients to overflow. Hence a scaling suitable for Example 3 will break down with Example 4. Certainly our algorithm of Sec. 5 will not work.

Does the reader feel that equations with such a large root will not occur in practical computations? Let him be assured that they do. The final, physically important result of a computation is almost certain to lie safely inside the range of floating-point numbers. However, intermediate results often appear with nonzero magnitudes smaller than 10^{-50} or larger than 10^{49}.

Recently several computing experts agreed that one of the most serious difficulties with many current computer systems is that they automatically replace an underflowed answer by zero, without any warning message. In such a system, $10^{-30} \times 10^{-30} \times 10^{31} \times 10^{30}$ would be computed as 0, Whereas $10^{-30} \times 10^{31} \times 10^{-30} \times 10^{30}$ would be computed as 10.

Example 5. $x^2 - 4.0000000x + 3.9999999 = 0$.

The correctly rounded roots are, to 10 significant digits,

$$x_1 = 2.000316228$$

and

$$x_2 = 1.999683772 .$$

If we apply the algorithm of Sec. 5, we find that

$$z = -b = 4.0000000$$
$$y = b^2 = 16.\ 0000000$$
$$w = 4a = 4.0000000$$
$$v = w \cdot c = 16.0000000$$
$$u = y - v = 0$$
$$t = \sqrt{u} = 0$$
$$s = z+t = z-t = 4.0000000$$
$$r = 2a = 2.0000000$$
$$x_1 = x_2 = s/r = 2.0000000 .$$

The computed roots are both in error by approximately 0.0003162. I.e., out of 7 computed digits to the right of the decimal point, only 3 are correct. Also, the computer mistakenly finds a double root instead of two distinct real roots.

The accuracy seems quite low. However, the roots of' Example 5 actually change very rapidly when the coefficients are changed, In fact, the two computed roots $x_1 = x_2 = 2.0000000$ are the exact roots of the nearby equation $0.999999992x^2 - 3.999999968x + 3.999999968 = 0$. Thus, though x_1 and x_2 are wrong roots of Example 5 by some 3162 units in their last decimal place, they are true roots of an equation with a, b, c differing from those of Example 5 by no more than 0.8 of a unit in their last decimal place.

Example 5 illustrates two different ways of measuring relative errors in any computation. In the so-called forward approach to relative error one notes that the computed roots x_1 and x_2 differ by so many units (here 3162) in the last place from the true roots of the given equation. In the so-called backward approach to relative error one says that the computed roots x_1 and x_2 are the exact roots of an equation with coefficients which differ by no more than so many units (here 0.8) in the last place from those of the given equation. The forward measure of error is perhaps more natural and certainly is traditional. The backward approach to error is more recent, but in many contexts turns out to be considerably easier to analyze and just as useful in practice, Backwards error analysis is one of the major ideas to be developed in the last decade of research in computational mathematics. Cornelius Lanczos devised the backwards approach in another context in the 1940's. Wallace Givens exploited it in 1954 for computing roots of certain equations. But James Wilkinson has done the most in the years since 1958 to exploit it as a basis for analyzing errors in floating-point computations on digital computers.

The reason why backwards error analysis is so useful is this: In the floating-point arithmetic system neither addition nor multiplication is an associative operation, and the two are not distributive. Thus the basic properties on which algebra is based fail to hold for floating-point arithmetic. Hence a forward error analysis, which is based directly on the floating-point operations, is extremely difficult to carry out. On the other hand, backwards error analysis interprets the result of each computer product, for example, as the true product of two real numbers which differ very slightly from the factors of the computer product. Thus in

backwards error analysis one deals with true mathematical multiplication and addition, which are associative and distributive. This permits analysis to be much more easily carried out, and often leads to closer bounds for the error.

This is not the place to develop these ideas further, but we hope to have given the reader an inkling of why backwards analysis, when applicable, is often so much more satisfactory.

7. Criteria of a good quadratic equation solver

The above examples illustrate the variety of behavior of the quadratic algorithm of Sec. 5. Examples 2, 3, and 4 make it clear that the algorithm is not satisfactory for all cases, and hence that it is an unacceptable algorithm. What do we really expect from a quadratic equation-solving algorithm?

Should we be content with the computer solution of Example 5, with its error of 3162 units in x_1 and x_2, since the computed roots do satisfy an equation which is so close to the given one?

We might be quite content with the results of Example 5, if we didn't know how to do better, but certainly not with Example 2. Quadratic equations arise in exceedingly many contexts of mathematics and computing. They are so basic that we should like to be able to compute their roots with almost no error, for almost any equation whose coefficients are floating-point numbers. Such performance can be achieved, and it is vastly important to have such algorithms in the computer library. Then, 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.

We want a quadratic equation solver that will accept any floating-point numbers a, b, c, and compute any of the roots x_1, x_2 that lie safely within the range of floating-point numbers. Any computed root should have an error in the last decimal place not exceeding, say, 10 units. If either x_1 or x_2 underflows, or overflows, there should be a message about what happened.

8. Some aspects of an accurate algorithm

Such an algorithm has been devised by William Kahan of the University of Toronto. The most difficult matter to take care of is the possibility of overflow or underflow. It will not be possible to describe the complete algorithm, but we can give some of the more accessible ideas,

First, we discuss the steps taken to overcome the great inaccuracy in root x_2, as computed in Example 2. In step (vii)' of Example 2, we subtracted two equal numbers z and t, to get $s' = 0$. The true value of t was not quite equal to z, because in step (v) the true u was not quite equal to y. But t and z, like u and y, could not be 'distinguished, with only 8 decimal digits at our disposal.

An easy cure for the difficulty is to use another method of computing x_2, in which the answer does not result from the subtraction of nearly equal numbers.

If a, b, c are any real numbers, and if $a \neq 0$ and $c \neq 0$, then the quadratic equation

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

is satisfied by exactly two values of x, namely

(5) $$x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}$$

and

(6) $$x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \quad .$$

Formulas (5) and (6) can be proved, for example, by first applying formulas (3) and (4) to the following equivalent form of the given quadratic equation:

$$c\left(\frac{1}{x}\right)^2 + b\left(\frac{1}{x}\right) + a = 0 \quad .$$

Note that if b is negative, then there is cancellation in formula (4) for x_2, but not in formula (6), and there is cancellation in formula (5)

for X1, but not in formula (3). The reverse statements hold in case b is positive. So, for any quadratic equation in which neither a nor c is zero, one selects formulas (3) and (6) when b is positive or zero, and formulas (4) and (5) when b is negative. For Example 2, formula (6) leads to the computer result that

$$x_2 = \frac{2.0}{10^5 + 10^5} = 1.0000000 \times 10^{-5} ,$$

a perfectly rounded root.

The inaccuracy of Example 5 cannot be so simply cured, because it is inherent in working with only 8 decimals, as is revealed by the rapid change of the roots with changes of a, b, c. The best known cure is to identify the delicate part of the computation, and use greater precision for it. So Kahan's algorithm uses double precision (here 16 significant decimals) in the computation of $u = b^2 - 4ac$, followed by rounding to single precision. The rest of the computation does not need extra precision, and is done in the normal way. There is a small penalty in the extra time required for that double precision computation, but it is a negligible part of the total time, which goes mostly toward scaling and otherwise detecting and correcting overflow or underflow possibilities.

Recomputation of x_1 in Example 5 looks as follows

$z = -b = 4.0000000$

$y = b^2 = 16.0000000\ 0000000$

$w = 4a = 4.0000000\ 0000000$

$v = w \cdot c = 15.9999996\ 0000000$

$u = y - v = 0.0000004\ 0000000\ 0000000$

$= 0.0000004\ 0000000$, returning to single precision

$t = \sqrt{u} = 0.0006324\ 5553$

$s = z + t = 4.0006325$

$r = 2a = 2.0000000$

$x_1 = s/r = 2.0003163$, rounding up.

Note that x_1 is in error by only 0.72 of a unit in the last decimal place.

It is not practicable to discuss scaling and dealing with possible overflow and underflow. The details are many and technical, and depend intimately on particular features of the computer on which they are carried out. They are extremely important to actual computing, but carry less general interest than the ideas just presented. One of the obvious features involves testing whether any or all of a, b, or c are zero.

9. Conclusion

We have described some of the pitfalls of applying the quadratic formula blindly with an automatic digital computer. We have given sound cures for two of the pitfalls, and indicated what other work has been done to create a first-class algorithm for solving a quadratic equation.

The quadratic equation is one of the simplest mathematical entities, and is solved almost everywhere in applied mathematics. Its actual use on a computer might be expected to be one of the best understood of computer algorithms. In fact, it is not, and some more complex computations were studied first. The fact that the algorithm of Sec. 5 is so subject to rounding error is not very widely known among computer users, or among writers of elementary textbooks on computing methods, and certainly not by most writers of mathematics textbooks. Of course it is known to specialists in numerical analysis. Thus even in this elementary problem we are working at the frontiers of common computing knowledge,

The majority of practical computations are understood still less than the quadratic equation. A very great deal of difficult research and development remains to be done before computers will be used as wisely and well as they can be. It is almost certain, for example, that various parts of the computations for weather forecasting contain pitfalls like those of the quadratic equation, and that ignorance of these pitfalls is introducing computational errors that are interfering with progress in weather forecasting. The same can be said about most nontrivial fields of scientific computation.

The moral of the story is that users of computers for mathematical problems require some knowledge of numerical mathematics. It is not

sufficient to learn some programming language, and then simply translate formulas from a textbook of pure mathematics into the language of a computer. The formulas and algorithms to be found in most mathematics texts were devised for the exact arithmetic of the real number system. Few authors have given any attention to the robustness of the formulas--that is, to the behavior of the formulas when used with the approximate arithmetic of computers. Until attention is given to robustness in mathematics textbooks, the would-be scientific computer must consult people and writings specifically concerned with machine computation.