IS IT REALLY ZERO?

CHEE K. YAP

Algebra is generous, she often gives more than is asked of her.

— Jean Le Rond d'Alembart (1717-83)

The history of the zero recognition problem is somewhat confused by the fact that many people do not recognize it as a problem at all.

— Daniel Richardson (1996)

1. What appears to be the problem?

In the previous issue of this magazine (KIAS Newsletter No.32, Summer 2006), Dr. Andreas Bender asked if $2 + 2 = 5$? Now I want to ask the apparently simpler question: *is a number really equal to zero*? It is simpler, not only because zero seems to be simpler than the other numbers, but because our question only requires deciding between a "yes" or "no" answer. Such problems are called *decision problems*.

We need to explain why this question requires any effort to answer. First of all, numbers have canonical names such as *zero, one, two, half, negative ten, square root of two, pi*, etc. In symbols, the canonical names are written as $0, 1, 2, \frac{1}{2}, -10, \sqrt{2}, \pi$, etc. Of course, when we are faced with a canonical name, we instantly recognize whether it is zero or not. So the problem arises when the number that we need to recognize is given by an *expression* such as

$$1 - 1, \quad 2^2 + 5 - 3^2, \quad 1 - \sum_{n=1}^{\infty} 2^{-n}, \quad \sqrt{2} + \sqrt{3} - \sqrt{5 + 2\sqrt{6}}, \quad \textit{etc.}$$

Each expression in this list is indeed equal to 0, although the last two may require a bit of mathematical sophistication to see. In the last expression, we could try a pocket calculator:

$$\left.\begin{aligned} \sqrt{2} + \sqrt{3} - \sqrt{5 + 2\sqrt{6}} &= 1.4142 + 1.7320 - \sqrt{5 + 2 \times 2.4494} \\ &= 3.1462 - \sqrt{9.8989} \\ &= 3.1462 - 3.1462 \\ &= 0? \end{aligned}\right\} \tag{1}$$

This seems to be 0, except for the uncertainty that stems from numerical rounding or truncation. How could we be sure? One response is to say that we must compute with higher and higher precision, until we discover the truth (zero or non-zero). But the two possible truth values here have asymmetric properties: increasing the precision can confirm *non-zero-ness* of an expression, but it can never confirm *zero-ness*. This is the central difficulty about deciding zero.

In our little example here, we can easily resolve the zero question with a little algebra: if we square $\sqrt{x} + \sqrt{y}$ the result is $x + y + 2\sqrt{xy}$. Hence, $\sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$ is identically 0, regardless of the values of x and y. Our expression corresponds to the case $x = 2, y = 3$. By the same token $1 + \sqrt{2} - \sqrt{3 + \sqrt{8}} = 0$. This illustrates how powerful algebra is: it can solve such zero problems decisively, without any approximation! We will return to this phenomena on the power of algebra.

Date: August 31, 2006.

You might have noticed that I phrased the title of this article in a curiously un-mathematical language, by talking about "really zero", as if there is such a thing as "apparently but not zero". Mathematically, I should simply put the question as: *is it zero?* The reason for my language is computational, as illustrated by our approximations in equation (1) which yield something that is apparently zero. Here is another apparent zero from Ron Graham, but this time it is non-zero:

$$\begin{aligned}&\left(\sqrt{1000001}+\sqrt{1000025}+\sqrt{1000031}+\sqrt{1000084}+\sqrt{1000087}+\sqrt{1000134}+\sqrt{1000158}+\sqrt{1000182}+\sqrt{1000198}\right)\\ -\quad&\left(\sqrt{1000002}+\sqrt{1000018}+\sqrt{1000042}+\sqrt{1000066}+\sqrt{1000113}+\sqrt{1000116}+\sqrt{1000169}+\sqrt{1000175}+\sqrt{1000199}\right)\end{aligned}$$

The number is less than 10^{-36}, so you need to approximate each value to over 36 digits beyond the decimal point to see that this is non-zero.

On Algorithms. Implicit in our discussion so far is the concept of an *algorithm*. Our question about recognizing zero amounts to asking if there exists an algorithm that can do this. To a first approximation, you may identify an algorithm with a computer program. So we want to know if there is a computer program that can decide zero. The concept of an algorithm is central in computer science and mathematics. You may not be familiar with the word "algorithm" but you know at least two algorithms. By grade three, modern school children learn how to add two integers, and by grade five, they learn how to multiply. In other words, children learned two algorithms. The multiplication algorithm here is often called[1] the *high-school multiplication algorithm*. In general, an algorithm is a *systematic procedure to manipulate representations of objects in order to carry out a specific computational task in finite time.* This is very general, because we did not say what are the "objects", how they are "represented", what operations you are allowed to use for "manipulation", and what is the "task". For the purposes of this article, our objects will be numbers and their expressions. In the addition or multiplication algorithms above, we assume that integers are represented in decimal notation.

2. Representations of Numbers

We have now established that deciding zero is a nontrivial problem, and that its solution requires some representation of the numerical expressions and an algorithm to manipulate these representations towards answering the question: it is zero?

Why is this problem important? A short answer is this: *whether or not we can decide zero determines whether or not we can compute correctly.* This may come as a surprise to many people who assume that whatever a computer computes must be correct. The reason computers make mistakes in numerical calculations is that numbers must be represented in some finite way. This finiteness constraint may lead to representations that has some error: the fraction 1/3 in decimal $1/3 = 0.3333\cdots$, or in binary $1/3 = 0.0101010\cdots$, are not valid representations since such strings are infinite. We could introduce conventions to represent the periodic part of such strings: $1/3 = 0.\overline{3} = 0.3\overline{33}$ (decimal) or $1/3 = 0.\overline{01} = 0.010\overline{10}$ (binary). The convention is to mark a suffix of the string (the part under the bar) which is to be repeated infinitely often. Alternatively, we could just use rational numbers: we represent 1/3 as a pair $(1, 3)$ of integers. Both extensions are valid representations, but as the representations become more general, the corresponding zero problem becomes more difficult. E.g., we must now recognize $(150111101515, 450300034545) - (1, 3)$ is zero.

There are more difficult numbers such as $\sqrt{2} = 1.4142\cdots$. The ancient Greeks already knew that $\sqrt{2}$ is not a rational number. So we need other ways of representing them. Numbers like $\sqrt{2}$ or the golden ratio $\phi = (1 + \sqrt{5})/2 = 1.6180\cdots$ are examples of *algebraic numbers*. Such numbers satisfy some polynomial equation with integer coefficients. E.g., $X = \sqrt{2}$ satisfies the polynomial equation $X^2 - 2 = 0$. We say $\sqrt{2}$ is the *zero* or *root* of the polynomial $X^2 - 2$. Similarly, ϕ is the zero (or

[1]This name is a misnomer today – will we someday call it the "kindergarten multiplication algorithm"?

root) of the polynomial $X^2 - X - 1$. A common representation of algebraic numbers goes as follows: if α is a real algebraic number, an *isolating interval representation* for α is a pair, $(p(X), [a, b])$ where $p(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$ is a polynomial with integer coefficients $a_0, \ldots, a_n$ such that $p(\alpha) = 0$, and a, b are rational numbers such that $a < \alpha < b$, and there are no other roots of $p(X)$ between a and b. We have algorithms to add, subtract, multiply, divide two algebraic numbers in this representation, and we can also decide zero in this representation.

Once we have a method of representing numbers, we may extend the representation to *expressions* over such numbers. Expressions are determined by the set of operations we allow: for instance, $\sqrt{2}+\sqrt{3}$ is an expression over the integers, involving the operations of $+$ and $\sqrt{\cdot}$. Typically, we allow operations from the set $\pm, \times, \div$ and possibly other operations such as $\sqrt{x}, \sin x, \exp x, \log x$, etc. It is clear that the zero problem depends critically on the class of expressions we allow as input. The problem of recognizing if two expressions are equal is just the zero problem in disguise, provided our expressions admit subtraction: two expressions α, β are equal iff $\alpha - \beta$ is zero.

We must clarify that even though we insist on finite representation, we do not go so far as to demand a *fixed precision* representation of our numbers. Modern computer languages typically represent integers and floating point numbers with a fixed budget of bits (typically 32 or 64 bits). In contrast to this, we insist on representations that can use as many bits as desired: such numbers are often called *Big Numbers*. There are standard libraries that support such Big Number computation.

Beyond Algebraic Numbers. By definition, numbers which are not algebraic are called *transcendental numbers*. The two most famous examples are $\pi = 3.1415\cdots$ and $e = 2.7182\cdots$. It is comparatively recent that these were shown to be transcendental (e in 1873 by Hermite, π in 1882 by Lindemann). We might say that *the fundamental problem of transcendental number theory is to prove that certain expressions are zero or non-zero.* For instance, proving that π is transcendental amounts to saying that for integers $a_0, \ldots, a_n$, not all zero, the expression $a_0 + a_1\pi + a_2\pi^2 + \cdots + a_n\pi^n$ is non-zero. Not much is known about recognizing zero for general expressions that involve transcendental functions or constants. For instance, we do not know whether $\pi + e$ or π^e are transcendental (even though e^π is provably transcendental). In the computer, we may represent transcendental numbers by some procedure that can generate approximations to any desired precision.

3. Again, What is Geometry?

In the last section, we said that the ability to decide zeros will help us to compute correctly. Most of computing does not appear to be about zeros. But surprisingly, a large part of computing *depends* on knowing zero. For instance, the algorithms in computational geometry often need to know whether a point p is on a line L. If $p = (x, y)$ in the usual Cartesian coordinates, and the equation of L is $aX + bY + c = 0$ (for some constants a, b, c), then p lies on L iff $ax + by + c = 0$.

This illustrates the principle that in mathematics, we use zeros to determine properties. If we are talking about *real* numbers, then we often need to know more – if a number is non-zero, we want to know whether it is positive or negative. Therefore, closely related to the zero problem is the *sign problem*: given a real number x, is it > 0, < 0 or $= 0$? We define $\texttt{sign}(x) = +1$ if $x > 0$, $\texttt{sign}(x) = -1$ if $x < 0$, and otherwise $\texttt{sign}(x) = 0$. In the above example of points $p = (x, y)$ and lines $L : aX + bY + c = 0$, deciding whether p lies on, to the left of, or to the right of the line L amounts to computing the sign of the expression $ax + by + c$. To encapsulate this property, define the *sided-ness predicate*, $SIDE(p, L)$ which returns $\texttt{sign}(ax+by+c) \in \{-1, 0, +1\}$. In practice, the sign decision problem is more important than the zero problem. We can usually reduce the sign problem to the zero problem as follows: first decide whether the number is zero. If non-zero, we can decide whether it is positive or negative by approximating the number to higher and higher precision until its sign is clear. In some domains such as the complex numbers, the zero problem is meaningful even when the sign problem is not. Hence we prefer to focus on the zero problem.

Thus the geometric relation between a point and a line is ultimately reduced to a zero problem. This application of the zero problem not an isolated phenomenon, but the defining characteristic of geometry: *all geometric phenomena arise from the discrete relations determined by suitable sign or zero predicates.* In recent years, computational geometers have become very conscious of this characterization because whenever they design geometric algorithms, they ask themselves: *what are the geometric predicates needed to implement this algorithm?* We have seen such predicates as $SIDE(p, L)$ above. The location of such predicates in our programs tells us exactly where the sign problem is needed in a computation. We conclude that geometric computation depends on knowing zero. Since the geometric phenomenon appears in almost every branch of mathematics, this establishes the centrality of the zero problem for computation.

The above characterization of geometry yields yet another answer to the age-old question about the nature of geometry. Mathematicians throughout history have repeatedly asked this question: Euclid, Descartes, Klein, Hilbert, Tarski, Dieudonné, Erdös, etc, have revealed different aspects of the geometric phenomenon. Many authors (Alexandrov, Fejes Tóth, Atiyah, Chern) have written articles with the title "What is Geometry?". What is unique about our characterization of geometry is that it comes from the computational perspective. Although the computational view point has deep roots in mathematics, it is a relative newcomer as a discipline. Even more recent is the subfield of computational geometry. In the last two decades, computational geometers have been struggling to understand the *nonrobustness phenomenon* in geometric algorithms. Many programmers, engineers and computational scientists know very well from experience that geometric software are notoriously nonrobust – they are liable to crash, fall into an infinite loop, or produce inconsistent results. Based on our understanding of geometry, it is easy to see why: zeros of geometric predicates correspond to discontinuities in the geometry. Singularities can be viewed as extreme forms of such discontinuities. So geometric computation is inherently discontinuous, and highly unstable at singularities.

A simple prescription for nonrobustness. Although there have been many proposals for solving such nonrobustness, conceptually the simplest is to tackle the problem head-on: *we simply must decide zero.* This kind of computing has been termed *exact geometric computation* (EGC). To date, it is the most successful approach for solving the robustness issue. Though obvious in retrospect, we must realize that this EGC prescription is anathema to some entrenched way of thinking about computation. That idea goes as follows: *because numbers on a computer have finite precision (cf. previous section), it must be inherently imprecise. Therefore computer programs should never try to branch by comparing a number x to zero. Instead, a comparison "$x : 0$" should be replaced by the test "$|x| \leq \varepsilon$".* If you look at numerical programs in the industry, you will find lots of such ε constants (each constant is determined empirically). So, the EGC prescription runs counter to a whole generation of computational practice.

To follow the EGC prescription, we need general algorithms to decide zero. Are they available? As promised, we now return to the earlier application of algebra to determine if the expression

$$E = \sqrt{2} + \sqrt{3} - \sqrt{5 + 2\sqrt{6}} \tag{2}$$

is zero. Algebraic manipulation can indeed be used to decide zero for arithmetic expressions that involve square roots. It can even be generalized to expressions that encompass all algebraic number expressions, i.e., expressions constructed out of the integers, and composed under the operations of $\pm, \times, \div, \sqrt{\cdot}$ and $RootOf(p(X), i)$. Here $p(X)$ is an integer polynomial and $i \geq 1$ indicates that we want to extract the ith largest real zero of $p(X)$. Naturally, $\div, \sqrt{\cdot}$ and $RootOf$ are partial operations that may be undefined on some arguments. It is re-assuring to know that an algorithmic solution for this zero problem exists. Unfortunately, such algorithms are computationally expensive and impractical.

Here is where algebra alone fails us. Instead, we will combine the numerical approximation method (cf. equation (1)) with a little amount of algebraic information. The idea is that algebraic numbers have a certain discreteness (ultimately inherited from the integers, $\mathbb{Z}$) that allows us to bound non-zero

algebraic numbers away from 0. More precisely, for any algebraic expression E, we can compute a number $M(E) > 0$ such that *if the value of E is well-defined and non-zero, then* $|E| > 1/M(E)$. We call $1/M(E)$ a *zero bound*. The ability to zero bounds, combined with the ability to approximate E to any precision, solves the zero problem for E. One such $M(E)$ bound is the (Mahler) *Measure Bound*, but several other bounds are known.

We will illustrate this idea using our favorite expression E in equation (2). In this case, the Measure Bound is $M(E) = 2^{38}3^85^8 < 2^{38}16^8$. Now $\log_2(1/M(E)) < 38 + 32 = 70$. This means that if we compute an approximation $\widetilde{E}$ of E which has at least 71 bits of accuracy, then $E = 0$ if and only if $|\widetilde{E}| < 2^{-71}$.

Such techniques for deciding zero are very important because it has *semi-adaptive complexity*. That is, the cost to conclude that a number *is non-zero* is proportional to its absolute value. This is the adaptive part. But the cost of concluding that it *is zero* depends on the quality of the zero bound. Unfortunately, this part is not so adaptive, but the topic of current research.

Before closing, I mention two further topics, one practical and one theoretical. First, there is an important practical technique called (numerical) *filters* that can speed up the non-zero decision process. Because of this, a large class of problems in computational geometry can now be solved robustly and efficiently in practice. Second, very limited zero bounds are known for transcendental expressions. The main theoretical open problem is whether the zero problem for algebraic expressions, in combination with exp and log, is decidable. Richardson (1996) has answered this in the affirmative, *conditioned* on the truth of Schanuel's conjecture from transcendental number theory.

Short of a general positive result, such as the unconditional version of Richardson's result, we would like to know if there are any specific transcendental geometric problem that we can decide. Recently, several researchers at KIAS [1] succeeded in showing the first such example. This is the *shortest path problem amidst disc obstacles*, and it arises in robotics and computational geometry: suppose you have a set of circular discs in the plane, and you want to find the shortest path from one point p to another point q which avoid these discs. Conceptually, we can reduce this problem to Djikstra's shortest path algorithm on a discrete graph. But to implement Djikstra's algorithm on a computer (i.e., Turing machine), we need to decide zero for a certain class of transcendental expressions. It turns out that we could exploit Baker's theorem on linear form in logarithms to decide it.

4. Conclusion

The questions raised in this article form an active area of research in computation. It has connections to computational geometry, computer algebra, transcendental number theory, geometric modeling, automatic theorem proving and complexity theory. The questions also brings new insights into the theory of real computation, an important area whose foundations are still unsettled. To read more about these topics, you may consult the survey paper [2].

You may also be interested to know that there are software libraries where these techniques are easily accessible by any programmer. That is, one can write an ordinary looking program[2] with the property that no error is ever committed in its numerical operations, and where comparisons are error-free. This is true liberation from the uncertainty caused by numerical errors in conventional programming languages. We believe that such libraries will become a standard fare in the future.

Acknowledgment

I thank Andreas Bender and Sungwoo Choi for their insightful comments.

[2]Two current implementations are LEDA real and the Core Library. They both assume the `C++` programming language. The open-source Core Library is freely available from `http://cs.nyu.edu/exact/`.

References

[1] E.-C. Chang, S. W. Choi, D. Kwon, H. Park, and C. Yap. Shortest paths for disc obstacles is computable. *Int'l. J. Comput. Geometry and Appl.*, 16(5-6):567–590, 2006. Special Issue of IJCGA on Geometric Constraints. (Eds. X.S. Gao and D. Michelucci).

[2] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.

Korea Institute of Advanced Study, Seoul, Korea
Current address: Courant Institute of Mathematical Sciences, New York University, NY, NY 10012 USA
E-mail address: yap@cs.nyu.edu