Solving Polynomials with Computers

Speedy computer algorithms offer new answers to a mathematical problem as ancient as Babylon: finding the zeros, or roots, of high-degree polynomials

Victor Y. Pan

When we think of a robot, many of us conjure up a science-fiction image of a two-legged, two-armed machine with lightbulbs for eyes, vaguely resembling a human being. In science fiction, robots are super-competent at mechanical tasks but unable to master the more difficult problems people have to deal with—good versus evil, honesty versus deception, the nature of self-awareness. For the time being, however, the reality of robots is quite different, because our first assumption is a fallacy: Robots are *not* super-competent mechanically. In fact, it is outrageously difficult to make a robot that can perform some of the simplest motions that we take for granted.

For an example, draw a square on your back halfway between your shoulders. Even though you cannot see your hand, you instinctively sense where it is going. Not so a robot. It can sense only the angles formed by its joints; if it needs to know where its hand is, it has to figure it out mathematically. It can solve the "forward kinematic problem"—if I make an angle θ_1 with my shoulder and an angle θ_2 with my elbow, where is my hand?—reasonably easily, with a little bit of trigonometry. But the more important *inverse kinematic problem*—if I want to put my hand in this spot, what angles should I make with my shoulder and elbow?—turns out to be a lot more difficult. The robot's brain (or computer chip) has to solve a system of four equations in four variables (the cosines and sines of θ_1 and θ_2 respectively). These equations can usually be reduced algebraically to a single equation with one unknown. Suppose, for example, that the robot arm has two one-meter segments. If a controller directs it to move its hand to a spot a meters above and b meters to the right of its shoulder, the computer will have to solve the following equation:

$$4(a^2 + b^2)x^2 - 4(a^2b + b^3)x \\ + (a^2 + b^2)^2 - 4a^2 = 0$$

In this equation, which we could call the "shoulder-angle equation," the unknown variable x is the sine of the angle formed by the shoulder (θ_1). After finding the appropriate value for x, the robot can use similar formulas to solve for the other variables and deduce the appropriate angle for its shoulder and elbow.

The complicated expression at the beginning of the shoulder-angle equation can be broken down into simple operations. A single unknown (x) is raised to various powers: x^2, x^1 (or x) and x^0 (or 1). These powers are multiplied by various known constants, such as $4(a^2 + b^2)$. Finally the resulting products are added together. Any such combination is called a *polynomial*. To solve the inverse kinematic problem, the computer must find a *zero*, or *root*, of the polynomial—that is, a value of x that makes the polynomial equal to zero. (For example, the number 1 is a zero of the polynomial $8x^2 - 8x$, because $8(1^2) - 8(1) = 0$. But 2 is not a zero, because $8(2^2) - 8(2) \neq 0$.) The problem of finding zeros of a polynomial vastly predates robots: Although there are no records of robots in ancient Sumer and Babylon, the Sumerians did know of problems of this sort, and the Babylonians knew how to solve specific instances.

One may say that a major goal of mathematics is to search for a few keys that can open numerous locks. If so, then the ability to solve polynomial equations is such a key. Not only can polynomial equations guide a robot's hand to the proper location, but such equations also turn up in a host of other technological applications: computer vision, modeling, graphics and the analysis of speech, to name four. Moreover, polynomials have always been a key to the development of mathematics itself. Over a period of more than two millennia, most of the significant extensions of our understanding of numbers and algebra have been provoked by questions about the roots of polynomials. The problem of finding *exact* solutions to polynomials was definitively—if negatively—resolved in the 19th century: There can be no universal polynomial-solving formula in terms of the standard algebraic operations (addition, subtraction, multiplication, division and radicals). With the advent of computers, 20th-century mathematicians faced a whole new challenge: how to find an *approximate* root of a polynomial in the most efficient way. This problem calls for just as much ingenuity as the old problem of finding exact roots, and mathematicians are just starting to get to the bottom of it.

From Real to Imaginary

In both the ancient and modern approaches to solving polynomials, the crucial factor controlling the difficulty of the

Victor Y. Pan is a professor in the Department of Mathematics and Computer Science at Lehman College of the City University of New York and in the CUNY Graduate Center Ph.D. Program in Computer Science. He holds a Ph.D. in mathematics from Moscow University and has held visiting research and teaching posts with IBM, the State University of New York at Albany, the Institute for Advanced Study, the International Computer Science Institute, Stanford and Columbia universities, Institute National de Recherche en Informatique et en Automatique and the University of Pisa. His research interests include the design and analysis of algorithms, algebraic computing, numerical methods, mathematical programming and operations research. Address: Department of Mathematics and Computer Science, Lehman College, CUNY, Bronx, NY 10468. Internet: vpan@lcvax.lehman.cuny.edu.

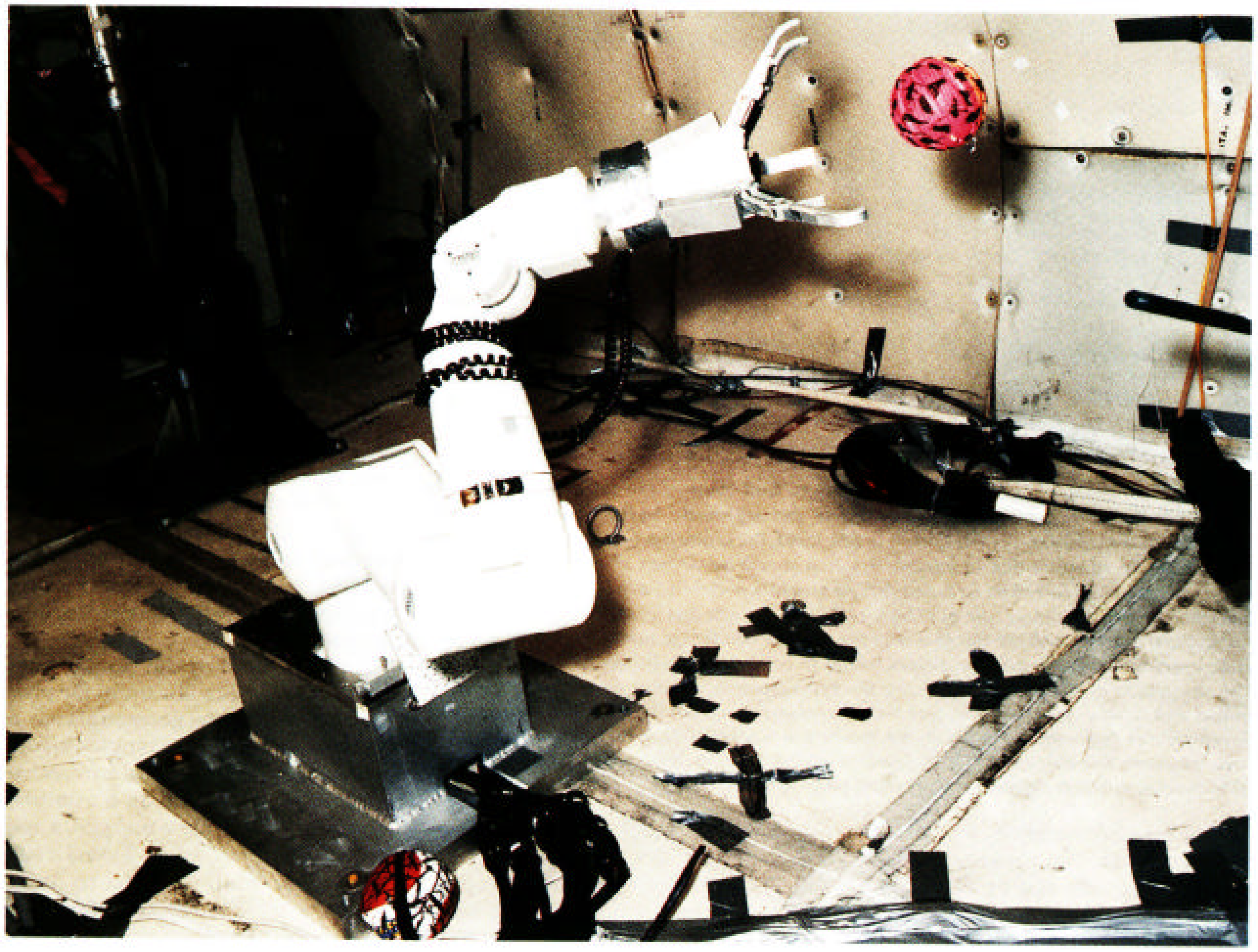

Figure 1.For a robot arm such as this one being developed at NASA's Johnson Space Center, finding the proper set of "elbow" angles for reaching a given spot is not a trivial computational challenge. This "inverse kinematic problem" involves a system of equations in which one must find the roots, or zeros, of several polynomials. Finding the zeros of polynomials is an ancient problem in mathematics that now is of great practical importance. The author and other mathematicians have developed efficient computer algorithms that rapidly approximate these solutions. In this photograph a prototype Extravehicular Activity Helper/Retriever on board a NASA reduced-gravity aircraft demonstrates how a vision-equipped robotic arm can manipulate and grasp moving objects in microgravity. (Photograph courtesy of NASA.)

problem is the highest power of the variable x that appears in the equation. This number is called the *degree* of the polynomial. The ancient Egyptians, Babylonians and Greeks were all able to solve *linear* (or degree-one) equations, such as $3x - 6 = 0$. In essence, any culture that knew how to divide could solve this equation, because the single solution $x = 2$ is obtained by dividing one of the known coefficients (6) by the other (3).

To solve *quadratic* (or degree-two) equations, $ax^2 + bx + c = 0$, however, required tremendous efforts of many people over thousands of years—culminating in a formula that has been drilled into generations of high-school students:

$$x = \left(-b \pm \sqrt{b^2 - 4ac}\right)/2a$$

To develop this *quadratic formula*, mathematicians had to develop a higher level of abstract thinking, manifested first of all by the use of abstract symbols such as a, b and c to represent the coefficients of the equation. A more specific, but related, difficulty was the necessity of coming to grips with the concepts of negative, irrational and imaginary numbers, all of which were quite odd to the ancient and even medieval ways of thinking.

By 2000 B.C., the Babylonians had come close to recognizing both irrational and negative numbers, but in solving an equation such as $x^2 - x - 2 = 0$ they would acknowledge only the solution $x = 2$, ignoring the solution $x = -1$. The Pythagoreans, around 500 B.C., were the first to discover that the solution of the equation $x^2 - 2 = 0$ (which we would write today as $\sqrt{2}$) could not be expressed as a ratio of whole numbers. The term *irrational number* that we still use for such numbers is a relic of the intellectual crisis this event set off. The Pythagoreans had believed that "[whole] numbers rule the universe," but now it seemed they did not even rule all of mathematics. Although they were the first to encounter the concept of irrational numbers, the Greeks still could not handle negative numbers. In this respect, the Chinese (3rd–1st century B.C.), Indians (7th–8th century A.D.) and Arabs (circa 850 A.D.) were more advanced. European mathematicians, however, became comfortable with negative numbers by 1545, when Italian mathematician Girolamo (or Jerome) Cardan manipulated them correctly to solve cubic equations, although he still called them *"numeri ficti."*

Cardan also noticed that the quadratic formula could give rise to *square*

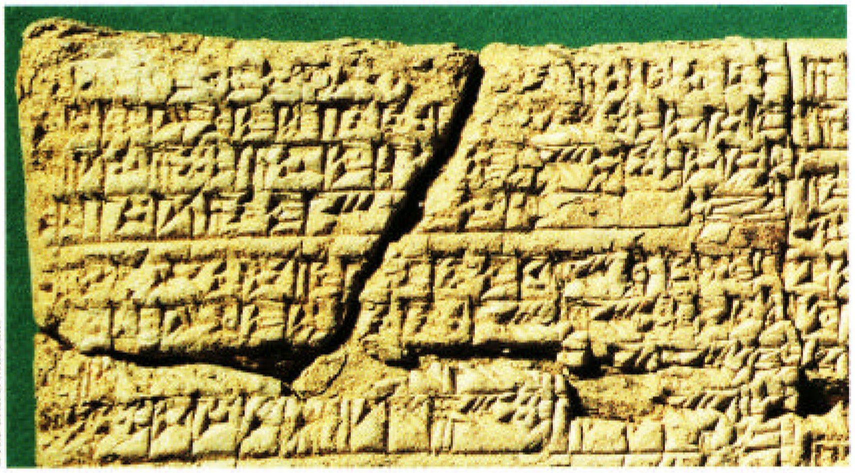

Figure 2. Babylonian mathematicians were able to find some of the roots of polynomials using tools that included reduction, substitution and arithmetic. In the first problem *(top)* described on the cuneiform tablet known as BM 13901, the scribe has found, by factoring, one solution for an equation that in modern notation would be written $x^2 + x - 3/4 = 0$. In the lower section he has solved an equation that we would write $x^2 - x - 870 = 0$, determining that one value for x is the square root of 870-1/4, or 29-1/2, plus 1/2; that is, 30. Except for the use of sexagesimal notation and the fact that only the positive solution is found, the scribe uses the same operations modern mathematicians would in computing x with the quadratic formula. The recognition of irrational numbers (by the Pythagoreans), negative numbers (by Chinese, Indian and Arab mathematicians) and finally complex numbers (in the 18th century) were developments that eventually led to the theoretical understanding that polynomials have roots (or zeros). But the power of 20th-century mathematics and modern computing was needed to develop efficient algorithms for finding polynomial zeros.

roots of negative numbers, a possibility he discounted: "So progresses arithmetic subtlety the end of which, as is said, is as refined as it is useless." But in time, 18th-century mathematicians including Roger Cotes, Abraham de Moivre and Leonhard Euler began to realize that numbers of the form $a + b\sqrt{-1}$ (known today as *complex numbers*) were not only useful but for many problems the most natural number system. Finally, Caspar Wessel (1797) and Carl Friedrich Gauss (1832), by following John Wallis (1673), pointed out that a complex number $a + b\sqrt{-1}$ could be thought of as an ordered pair of numbers (a, b)—in other words, as the coordinates of a point in the Euclidean plane. Moreover, every algebraic operation that could be done to complex numbers—addition, multiplication, even radicals—had a simple geometric interpretation. The respectability of plane geometry, as venerable as any branch of mathematics, rubbed off on the complex numbers. Although a number such as 2 is still called "real" and a number such as $2\sqrt{-1}$ is still called "imaginary," since Gauss's time mathematicians have considered both of them to be equally legitimate mathematical concepts. (In fact, any real number is also a complex number; for example, 2 can be written as $2 + 0\sqrt{-1}$.)

Simplicity Through Complexity

In this magazine, William Menasco and Lee Rudolph wrote (January–February 1995) that complex numbers "are harder to visualize than real ones, but most mathematics is actually much easier when one works with complex numbers." Behind this hidden power of complex numbers lies the *fundamental theorem of algebra*, the subject of Gauss's doctoral dissertation in 1799 as well as the work of many other famous mathematicians of the 17th through 20th centuries. This remarkable theorem states that every polynomial, no matter how high its degree, and no matter whether its coefficients are real or complex numbers, has a zero in the complex plane. In fact, Gauss did even better than that: He showed that the number of complex zeros is always identical to the degree of the polynomial. (There is one important caveat: Some zeros may need to be counted more than once. For the polynomial $x^2 - 2x + 1$, which can be factored as $(x - 1)^2$, the zero $x = 1$ is said to occur with *multiplicity* 2, as the factor $(x - 1)$ occurs twice.) This uniformity of behavior—the fact that the number of zeros depends on the degree, and nothing else—contrasts with the unpredictability of the same problem if only real zeros are allowed.

For example, a quadratic polynomial (a polynomial of degree two) may not have any real zeros, it may have one, or it may have two. The inconsistency can be illustrated vividly by the shoulder-angle equation, which is quadratic. Unless you have arthritis, there are some spots on your back that you can touch in two different ways: with the elbow up, or with the elbow down. For such positions, the shoulder-angle equation has two solutions. But there are some spots you cannot touch at all in a normal standing position: for example, your ankle or a high ceiling. These are cases where the shoulder-angle equation has no real-number solutions. It seems incredible that rotating your shoulder and elbow through an *imaginary* or *complex* angle, if such a thing were possible, would enable you to touch your ankle. Yet that is exactly what the fundamental theorem of algebra assures us!

Gauss's proof of the fundamental theorem of algebra illustrates the power and pitfalls of the geometric view of complex numbers. A complex-valued polynomial has both a real part and an imaginary part (just as a complex number, $a + b\sqrt{-1}$, has a real part, a, and an imaginary part, $b\sqrt{-1}$). To find a zero of a polynomial, we need to find a point $x + y\sqrt{-1}$ in the complex plane where both the real and imaginary parts of the polynomial become zero. Gauss asserted that the points in the plane that make the real-part zero form a curve, and those that make the imaginary-part zero also form a curve. Moreover, he proved that the ends of these two curves are equally spaced, like the hour marks on the face of a watch, and they alternate (real-imaginary-real-imaginary). Then he assumed it was obvious that the curves had to cross over each other in order to get from one "hour mark" to another. At any such crossroads, both the real and imaginary parts of the polynomial would equal zero, as desired.

Later mathematicians did not consider the existence of the crossings to be so obvious; the first complete proof of the theorem is dated to the 19th century. Actually, even when Gauss revised his proof 50 years later, he was unable to justify this step convincingly. In fact it was not until 1920, over a century after Gauss wrote his thesis, that the Russian mathematician Alexandre M. Ostrowski (who spent much of his life in Switzerland) was able to fill in the gap, using the modern theory of algebraic curves.

Note that Gauss's proof does not allow us to "write down" a solution to a polynomial, in the same way that the quadratic formula does. It merely asserts that the solution exists, on the grounds that a road from point A to point B must cross a road from point C to point D. By Gauss's time, mathematicians had made some progress on finding explicit solutions of equations of degree higher than two. In 1545 Cardan, despite his reluctance to allow complex numbers, published a formula for finding the zeros of cubics (degree-3 polynomials)—a baroque concoction, involving square roots inside cube roots, that is still known as *Cardan's formula* even though Cardan himself acknowledged obtaining the main idea of the formula from a rival, Nicolo Tartaglia.

In the same paper, Cardan presented the work of his student Ludovico Ferrari, who achieved a similar result for quartics (degree-four polynomials). Attempts to find similar formulas for larger degrees continued until the Italian mathematician Paolo Ruffini, in 1813, and the Norwegian mathematician Niels Henrik Abel, in 1824, proved that quintic (degree-five) polynomials cannot be solved by any expression involving only arithmetic operations and radicals. By 1832, the French mathematician Evariste Galois developed a celebrated theory of equations (still called *Galois theory*), which produces simple specific equations that cannot be solved by radicals—for example, the quintic equation $x^5 - 4x - 2 = 0$. Galois theory brought an elegant close to an old problem, but it also opened up new areas of mathematics: The concepts of fields, groups and extension fields he introduced have motivated numerous discoveries in abstract algebra, up to the present day, and have had a major impact on the applied subject of computer algebra.

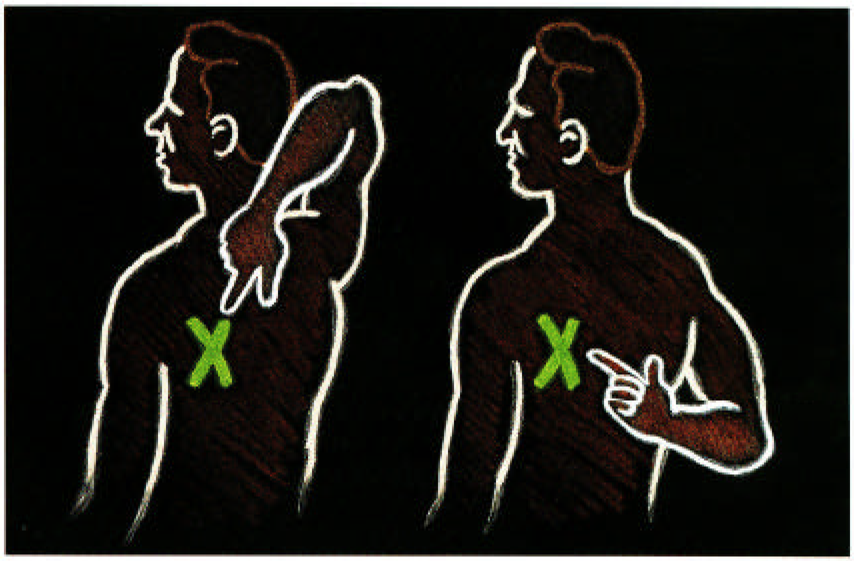
Figure 3. Most people can touch certain spots on their back in two different ways. A mathematician would say that the angle between the shoulder and the torso is governed by a quadratic (degree-two) equation with two feasible solutions. But the fundamental theorem of algebra offers additional ways to solve polynomial equations by searching for zeros, or roots, outside the realm of real numbers. Such a solution can only be imagined in the world of reality: In this case it would involve rotating your shoulder and elbow through an *imaginary* or *complex* angle.

Changing the Paradigm

For a modern-day engineer, trying to program a robot arm to move where he wants it to, the upshot of this beautiful mathematical theory, constructed over 4,000 years, must be rather disappointing. True, the motion of a simple arm with only two joints can be solved exactly. But the degree of the polynomials goes up exponentially with the

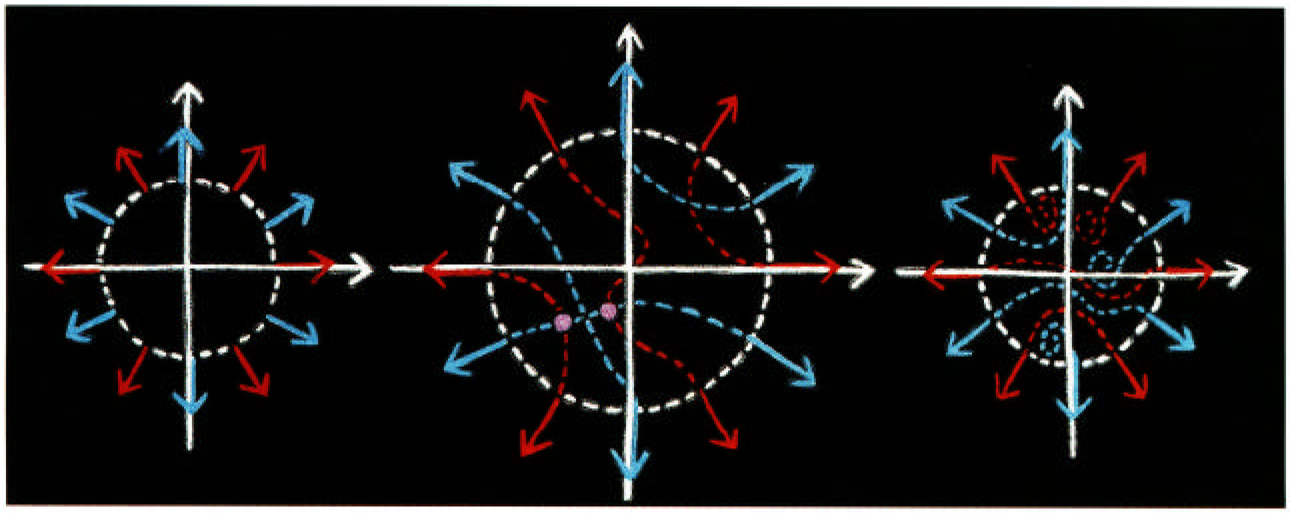
Figure 4. Carl Friedrich Gauss's proof that all polynomials have zeros in the plane of complex numbers was revolutionary, if flawed, in its use of geometry to do algebra. In this illustration curves in red pass through all points where the imaginary part of the polynomial is zero; curves in blue show where the real part is zero. Zeros of the polynomial are found where the curves cross. Outside a very large circle, the red and blue curves are spaced evenly, like the hour marks on a watch, with alternating colors *(left)*. Gauss argued that joining the ends together forces red and blue curves to cross *(middle)*. But he failed to rule out the possibility shown at right: The curves could spiral around without escaping the large circle, and thus avoid each other. Later mathematicians rescued his proof by showing that curves defined by polynomial equations must have both ends outside the circle.

number of joints involved. Thanks to Gauss, the engineer knows that the polynomials have complex zeros (although they may not all be physically meaningful). But the results of Ruffini, Abel and Galois mean that, barring great good fortune, the engineer has no hope of finding an exact zero of the high-degree polynomials. What an engineer needs, though, is not an *exact* solution, but a quick and dependable *approximate* solution.

Contemporary research efforts have focused on devising *iterative algorithms* to solve polynomial equations—methods that zero in on the zeros to any desired degree of precision. The first iterative algorithm for approximating the zero of a polynomial can be traced all the way back to ancient Egypt. This algorithm was brought by Arab mathematicians to medieval Europe, under the name of the *regula falsi* or "false position" method. Together with its modifications, the bisection and secant methods, it remains a major tool for solving equations on computers.

To demonstrate a major idea behind these methods, imagine that a river separates two villages. No matter which way you use to go from one village to the other, you must cross the river, because your path is a continuous curve on the ground. Analogously, any polynomial p can be viewed as a continuous function, whose graph is an unbroken curve in a plane with coordinates x and y. You may think of this curve as your path between the villages. Suppose the river runs along the line of points with y-coordinate equal to 0. If the village at $x = 0$ is south of the river (that is, its y-coordinate is negative) but the village at $x = 1$ is north of the river (its y-coordinate is positive), then there must be some number x between 0 and 1 where your path (with y equal to the value of p at x, or $y = p(x)$) and the river (with $y = 0$) cross. At this point, $p(x) = 0$—that is, x is a zero of the polynomial *(see Figure 5)*.

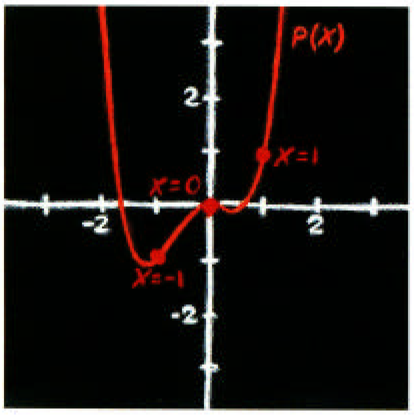

Figure 5. Algorithms for finding roots of polynomials rely on the idea that polynomial functions are unbroken curves that cross the x-axis; each root lies at the point along the graph of the function where $y = 0$, and for this reason it is called the zero of the polynomial. For instance, in a graph of a function where $y = p(x)$, if y is (even barely) negative when $x = 0$, and positive when $x = 1$, then there must be a number between 0 and 1 where the curve crosses the x-axis. In the realm of real numbers, a quartic polynomial such as the one above may have as many as four zeros.

With these observations in mind, perhaps the simplest version, the bisection method, is nothing more than a mathematical version of the children's game of "Twenty Questions." Suppose you want to find a zero of a polynomial, and you know that the value of the polynomial at $x = 0$ is negative but the value at $x = 1$ is positive. Then at some number z between 0 and 1, the value of the polynomial must equal 0. In Twenty Questions, you would start out by asking, "Is z bigger than 1/2?" But in the bisection method you have to be a little cleverer, since your "opponent," the polynomial, will not give you that information directly. Instead you ask, "Is the value of the polynomial at $x = 1/2$ negative?" If so, then there must be a zero between 1/2 and 1 (since the polynomial changes sign over that interval). If not, there must be a zero between 0 and 1/2. Either way, you get the information you want.

Suppose you find out that z is between 0 and 1/2. Then your next question would be, "Is the value of the polynomial at $x = 1/4$ negative?"—or, in translation, "Is z bigger than 1/4?" Continuing in this fashion, you would narrow down the interval z lies in by a factor of 2 at every step. You might never find the exact value of z, but after 20 questions you would be within $1/2^{20}$ of it—or, to put it another way, you would know z to about six decimal places. For a robot, this might well be close enough. (Of course, you could ask more or fewer questions, depending on which mattered more, accuracy or speed.)

The strengths of the bisection method are its absolute simplicity and reliability. But it cannot be used to find complex zeros, because the notion of betweenness is not well-defined in a plane. A somewhat similar method, the *quadtree method*, was devised by the German mathematician Hermann Weyl in 1924 to overcome this limitation. But this method, like bisection itself and its other variations, zeros in on the solutions fairly slowly.

A much more rapid, but finicky, technique is known as *Newton's method*, even though it originated millennia before Newton and first appeared in its present form in a 1690 work by Joseph Raphson. To use this method, you start out with a single initial estimate, x_0, usually chosen to be fairly close to the unknown zero, z. You find the corresponding point on the graph of the polynomial and draw a tangent line to the graph at that point. The spot where the tangent line intersects the x-axis is your new approximation to z, which you can call x_1. Generally it will be a better approximation to z than x_0 was, because the graph of the polynomial (a curve) will not deviate very far from the tangent line over a short distance. But if x_1 is not a good enough approximation to z, the procedure can be repeated to produce an even better approximation, x_2, and so on.

In Newton's method, each iteration roughly doubles the number of correct decimals. Thus, starting with an estimate that is accurate to one decimal place, in three steps a computer programmed with Newton's method could estimate z to eight decimal places, whereas a computer programmed with the bisection method would still be just beginning its game of Twenty Questions. Moreover, although this is not obvious, Newton's method can be used to find complex zeros as well as real zeros. But there is a catch: Newton's method has been proven to exhibit this fast rate of convergence only if the initial approximation x_0 is close to z, and even then only if there are no other zeros nearby. It may work slowly or fail completely if several zeros of the polynomial form a cluster. This case is not rare in practice, because practical applications frequently deal with polynomials with multiple zeros whose coefficients are actually represented numerically, with small round-off errors. If the original polynomial has a single zero with multiplicity 10, the rounded-off version typically has 10 very closely spaced zeros with multiplicity 1.

By now, hundreds of iterative algorithms are available for approximating polynomial zeros, and most of them are quite effective on average-case, smaller-degree polynomials, although they usually run into problems in treating polynomials with clustered zeros. Several of these algorithms are used successfully in commercially available software. However, a 1994 paper that numerically tested the three most popular algorithms concluded, "None of the methods gives acceptable results for polynomials of degree higher than 50," and, "If roots of high multiplicity exist, any … method has to be used with caution."

Keeping Score

The bisection and Newton's methods locate just one zero of a polynomial at a time. Recent algorithms have a more ambitious goal: to compute all n zeros of a complex polynomial of degree n at once. Surprisingly, by one theoretically important measure, a computer can do this in little more time than it takes to write them down.

Among computer scientists, the customary measure for the time-complexity of an algorithm is the number of *bit operations* involved. Numbers are stored in a computer as a string of 1s and 0s; each time two of these are combined by addition or multiplication, one *bit operation* takes place. A computer, just like a person, may take longer to add two 10-digit numbers than two one-digit numbers, although the time may be measured in billionths of a second rather than in seconds. However, for simplicity we will ignore this difference and describe the complexity in terms of *arithmetic operations*, or "ops." One op is simply a sum or product of two numbers. (Applied mathematicians frequently say "flop," referring to an op performed by using floating-point computer arithmetic.)

How many ops does it take to find the n roots of a degree-n polynomial? As previously noted, there are two controlling factors. Most important is n itself, the degree of the polynomial, which is also a rough measure of the amount of input data (a polynomial of degree n usually has about n coefficients). Second is the desired accuracy of the estimate, which can be measured by the number of correct binary digits in the answer, h. As either n or h get larger, the number of ops will increase, and the computer executing the algorithm will slow down.

Since each arithmetic operation has two operands and one output, it will take at least n operations simply to write down (output) all of the n zeros of the polynomial. This can be taken as a benchmark for rapidity, the fastest run time that any algorithm could possibly achieve. Moreover, even to output *one* of the zeros must take at least $n/2$ operations: Since each zero depends on *all* the input data, any algorithm that computes a zero must somehow take into account each of the n coefficients. Even if all it does is add the first two, then add the next two, then add the next two, and so forth, that is already $n/2$ additions.

The way computer scientists "keep score" of the speed of algorithms might seem strange at first. To begin with, they generally ignore constant factors. Thus they would say that both of the hypothetical algorithms mentioned in the last paragraph use "order n" ops. Similarly, if one algorithm used n^2 ops and another algorithm used a million times n^2 ops, both would be considered "order-n^2" and might even be called "equally fast." One reason for this callous indifference toward constants is that over the long haul, as n gets extremely large, any order-n algorithm will outperform any order-n^2 algorithm. All that counts is the exponent on n. When it comes down to writing the code for an algorithm, a computer scientist would certainly start to worry about constants, but until then they are an unnecessary encumbrance. It is hoped that that in practice the constants would not be too large to outweigh the factor n even for moderately large n.

Computer scientists do keep track of one other kind of factor: the logarithm (usually to the base 2) of n. The function $\log n$ does increase as n increases, and so, unlike a constant, it cannot be ignored. But $\log n$ increases very slowly—more slowly than n raised to any power. Hence, computer scientists are willing to accept any number of $\log n$'s to get rid of a single n, as long as n is large. For example, an algorithm that runs in "order $n(\log n)^4$" ops is superior, for large enough values of n, to an algorithm that runs in "order n^2" ops. On the other hand, "order $n(\log n)^4$ is worse than "order n," but not by much.

Weyl's quadtree method, the first algorithmic method for solving polynomials, takes on the order of $n^2h \log n$ steps to find all n zeros to an accuracy of h binary digits. I have modified his method to replace h by $\log h + \log n$—a substan-

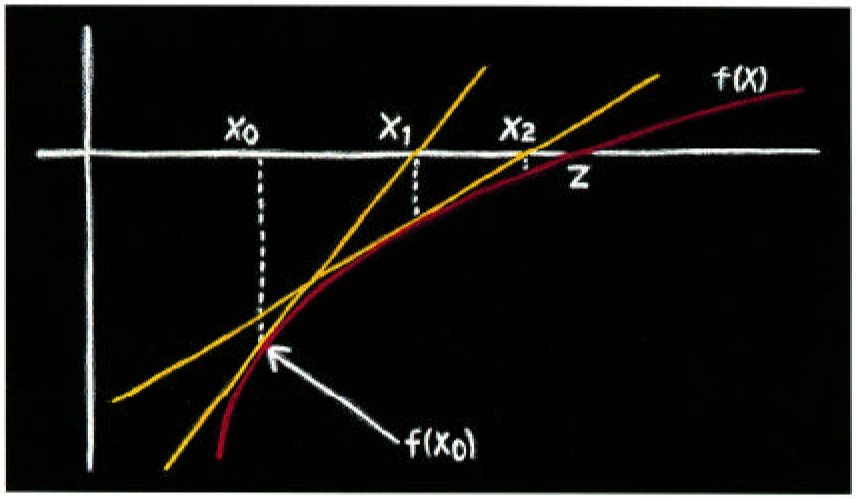

Figure 6. Newton's method is named after Isaac Newton, although its origin can be traced to Babylonian times and it was finalized by Joseph Raphson in 1690. The method zeroes in on a solution to a polynomial faster than the bisection method but only if a good enough initial guess, x_0, is made. Subsequent approximations x_1, x_2 are obtained by approximating the graph of the polynomial by a line and calculating where this tangent intersects the horizontal axis. Typically, the number of correct digits is proportional to the square of the number of steps.

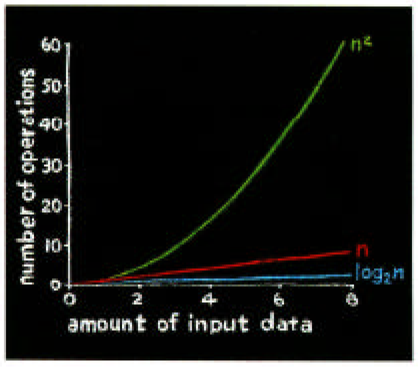

Figure 7. Functions encountered in computing complexity estimates increase as the amount of input data (n) increases. An algorithm that requires n^2 steps is ultimately much less efficient than an algorithm that requires n steps, which in turn is less efficient than an algorithm that requires ($\log_2 n$) steps.

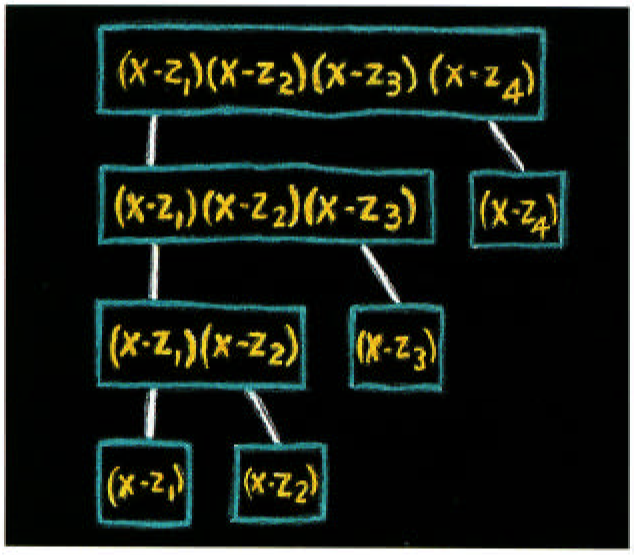

Figure 8. Using factoring, an unbalanced fan-out algorithm offers an approach to finding all the zeros (z_1, z_2, z_3, z_4 in this example) of a polynomial. For a polynomial of degree *n*, the algorithm requires order-*n* steps, peeling off one zero at each step.

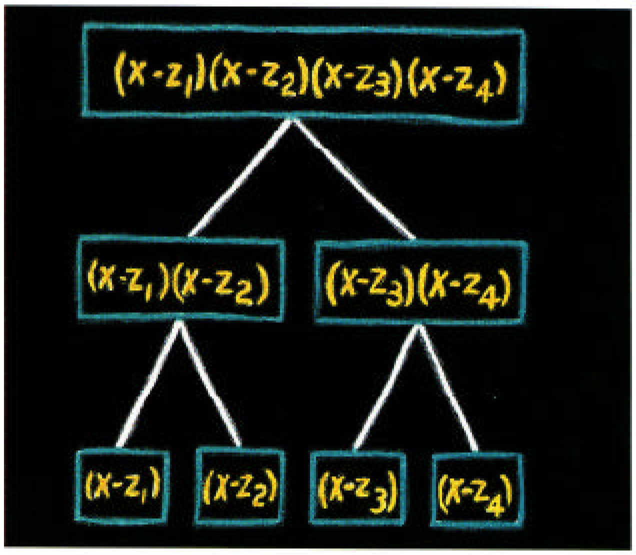

Figure 9. Balanced fan-out algorithm factors a polynomial into two pieces of roughly the same degree. For a polynomial of degree *n*, the balanced algorithm requires only order of (log *n*) steps to determine all the roots.

tial improvement with respect to the accuracy constraint *h*, but no improvement at all with respect to the important constraint *n*. Because of the factor of n^2, even the modified Weyl method is very much worse than the optimal "order-*n*" benchmarks. But in 1995 and 1996, I published the first polynomial-solving algorithms that truly approach the optimal run time: They take on the order of $n(\log n)^4 + n(\log n)^2(\log h)$ ops. These algorithms, based on an idea called "balanced fan-out," are not only the best to date but, if one ignores the logarithmic factors, the best possible. The superior complexity estimates as *n* goes to infinity do not guarantee practical superiority over Weyl's modified algorithm for more modest values of *n*. But neither do they rule it out: For a polynomial of degree 1,000, for instance, Weyl's order $n^2(\log n)^2$ ops become approximately 100 million, versus 10 million for the balanced fan-out approach.

The term "balanced fan-out" approach can best be understood by a picture. Suppose the polynomial $f(x)$ has degree four, and hence four complex roots, z_1 through z_4. Then, up to a constant factor, $f(x)$ can be factored as follows:

$$f(x) = (x - z_1)(x - z_2)(x - z_3)(x - z_4)$$

Perhaps the most obvious way to find all four zeros is to first find z_1, and then divide the polynomial $f(x)$ by $(x - z_1)$. The result will be a new polynomial of degree three, with only three zeros: the same zeros as $f(x)$, except for z_1. This procedure can be repeated until all the roots are found. Schematically, the algorithm proceeds as shown in Figure 8. It takes three steps to find all the roots; if we had started with a polynomial of degree *n*, it would take $(n - 1)$ steps. But remember that each step—finding a zero—itself takes at least order of *n* ops, as we saw before. Thus any program that uses this "unbalanced fan-out" process to find all the zeros must require order of n^2, or $n(n - 1)$, ops.

By contrast, the approach illustrated in Figure 9 separates the polynomial at each step into two factors of (roughly) the same degree. Factoring the degree-eight polynomial takes only three steps, and, in general, factoring a degree-*n* polynomial takes log *n* steps. Because each step itself requires *n* ops, a "balanced fan-out" process could, in principle, run in order of *n* log *n* ops—a vast improvement over n^2. Instead of whittling away at the problem, it would dice it up with a few swift strokes.

Such a dicing tool was found in 1982 by Arnold Schönhage, then at the University of Tübingen and now at the University of Bonn in Germany, in a culmination of the efforts and techniques of several other people. Schönhage's algo-

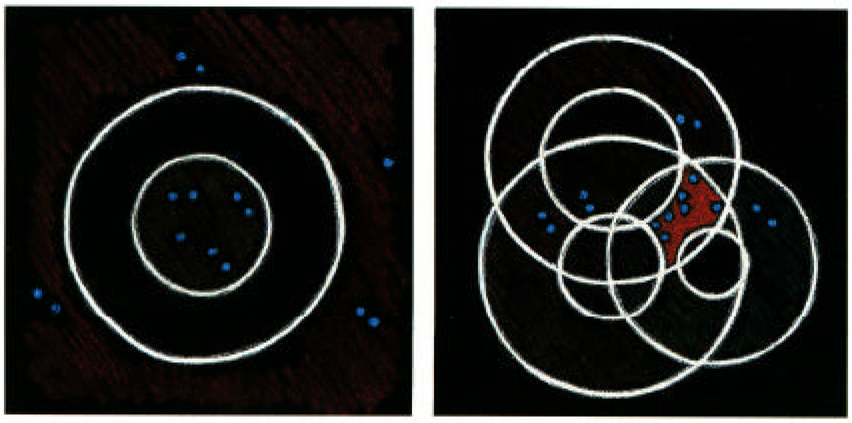

Figure 10. Balanced splitting of a polynomial can be accomplished by algorithms of Arnold Schönhage, Jean-Paul Cardinal, and Dario Bini and the author, provided that the zeros of the two factors of balanced degrees are separated by a wide enough "moat" *(left)*. If no moat wide enough exists, than the opposite occurs: Most of the zeros are concentrated into narrow rings. Intersecting three of these rings gives a zero-rich region *(right, brown area)*. This region, or one derived from it, can be used as the center of a moat suitable for the balanced-splitting algorithm.

rithm cleaves an arbitrary polynomial into two factors of roughly equal degree, as long as there is a large enough zero-free moat (or, to use the technical term, "annulus in the complex plane") separating the zeros of the two factors. (The problem can also be solved effectively by more recent algorithms proposed (in 1996) by Jean-Paul Cardinal of the University of Paul Sabatier in Toulouse, France, and by Dario Bini of Pisa, Italy, and myself.)

There is only one problem with this rosy scenario. Until recently, no one knew an efficient way to find such a moat. All proposed balancing algorithms have failed in some way: They required too many arithmetic operations or worked only when all the zeros were real.

A major step toward balancing was the theorem proved in 1994 by Don Coopersmith and C. Andrew Neff at the IBM T. J. Watson Research Center in New York, that any massive cluster of zeros of a polynomial $f(x)$ must lie close to a zero of a much smaller-degree polynomial $g(x)$, derived in a known way from $f(x)$ (and called a higher-order derivative of $f(x)$). Later I learned that the same help could have come from an earlier theorem proved in 1952 by the late Russian mathematician Alexandre O. Gel'fond of Moscow University. The theorem implies that a sufficiently wide zero-free moat can always be found somewhere in the complex plane. Figure 10 shows the basic idea. If no moat exists, then the opposite behavior must occur: Around every point there must be a ring that contains a fairly high percentage of the zeros. Three such rings are shown in the illustration. The intersection of these three rings is a small region with a large collection of zeros. By adjusting the size of the rings, one can get half of the zeros of $f(x)$ to lie inside the intersection. This is a concentrated enough cluster for Gel'fond's and Coppersmith and Neff's results to take effect. Then, roughly speaking, a sufficiently wide zero-free moat can be found around the cluster—its precise location depending on the location of the zeros of the derived polynomial $g(x)$—and the cited algorithms can be used to obtain a balanced splitting of $f(x)$.

The resulting recursive algorithm for approximation of the zeros of $f(x)$ still falls short of the optimal order-n (up to logarithmic factors) complexity boundary, because the associated fan-out process was ternary rather than binary: It entailed the extra work of finding a zero of the derived polynomial. However, I have developed additional geometric and algebraic techniques to remove this extraneous term. Another practical limitation of the algorithm has not been overcome yet: The geometric constructions involved in the search for a zero-free moat do not seem to be easy to code. Owing to this complication, the balanced fan-out algorithm in its present state does not seem to be very promising in the case of a low-degree input (low n) and a low-precision output (low h). But the substantial advantages the method promises for more taxing problems should motivate further efforts to overcome this difficulty—both on the computer end, by improving the code, and on the mathematical end, by simplifying the geometric algorithms.

Meanwhile, the practical problem of improving the existing software for zero-finding remains quite urgent. A European project originally called POSSO (for Polynomial System Solving) and now continued as FRISCO (Framework for Integrated Symbolic/Numeric Computation) is re-examining, updating and coding the best available algorithms. In addition to the modified Weyl construction, FRISCO uses known extensions of Newton's iteration, which approximate all n zeros of a polynomial simultaneously. These methods are simple to implement and use on the order of n^2 ops per iteration. Usually they converge very rapidly to approximate all the zeros in a few iterations, yielding a solution in the order of $n^2 \log h$ ops. This is a bit faster than the modified Weyl algorithm, $(n^2 \log n)(\log h + \log n)$ ops, although their fast convergence is a heuristic conclusion from many experiments performed for a large class of input polynomials. Unlike Weyl's modified algorithm and fan-out algorithms, they have not been proved to converge this rapidly for all input polynomials, and in some instances they are known to fail. It is unclear yet how frequently such instances occur in practice.

The gap in speed between the fastest known practical algorithms and the emerging fan-out algorithms, along with the uncertainty over the theoretical limitations of the former, indicate that further progress is likely in mathematicians' 4,000-year quest for the best way to solve polynomials. The robots of the future are waiting.

Acknowledgment

The author thanks Dana Mackenzie for his skillful assistance with the preparation of this article.

Bibliography

Atkinson, K. 1978. *Introduction to Numerical Analysis*. New York: Wiley.

Bell, E. T. 1940. *The Development of Mathematics*. New York: McGraw-Hill.

Bini, D., and V. Y. Pan. 1994. *Polynomial and Matrix Computations, v.1: Fundamental Algorithms*. Boston: Birkhäuser.

Bini, D., and V. Y. Pan. 1998. *Polynomial and Matrix Computations, v.2: Selected Topics*. Boston: Birkhäuser.

Bini, D., and V. Y. Pan. 1996. Graeffe's, Chebyshev-like and Cardinal's processes for splitting a polynomial into factors. *Journal of Complexity* 12:492–511.

Boyer, C. A. 1968. *A History of Mathematics*. New York: Wiley.

Cardinal, J.-P. 1996. On two iterative methods for approximating the roots of a polynomial. In *Proceedings of a Workshop: Mathematics of Numerical Analysis: Real Number Algorithms*, ed. J. Renegar, M. Shub and S. Smale. Lectures in Applied Mathematics 32:165–188. Providence: American Mathematical Society.

Cox, D., Little, J., and O'Shea, D. 1996. *Ideals, Varieties and Algorithms* (2nd edition). New York. Springer.

Gauss, G. F. 1973. *Werke*. New York: Band X, George Olms Verlag.

Henrici, P. 1974. *Applied and Computational Complex Analysis*, 1. New York: Wiley.

McNamee, J. M. 1993. A bibliography on roots of polynomials. *Journal of Computational and Applied Mathematics* 47(3):391–394.

Madsen, K. 1973. A root-finding algorithm based on Newton's method. *Bit* 13:71–75.

Neugebauer, O. 1957. *The Exact Science in Antiquity* (2nd edition). Providence, R.I.: Brown University Press.

The Numerical Algorithms Group Ltd. 1996. *FRISCO—A Framework for Integrated Symbolic/Numeric Computation*. http://extweb.nag.co.uk/projects/FRISCO.html. Main NAG site: http://www.nag.co.uk/

Pan, V. Y. 1987. Sequential and parallel complexity of approximate evaluation of polynomial zeros. *Computers and Mathematics (with Applications)* 14(8):591–622.

Pan, V. Y. 1994. New techniques for approximating complex polynomial zeros. *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*. New York: ACM Press; Philadelphia: SIAM Publications. pp. 260–270.

Pan, V. Y. 1995. Optimal (up to polylog factors) sequential and parallel algorithms for approximating complex polynomial zeros. *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*. New York: ACM Press, pp. 741–750.

Pan, V. Y. 1996. Optimal and nearly optimal algorithms for approximating polynomial zeros. *Computers and Mathematics (with Applications)* 31:97–138.

Pan, V. Y. 1997. Solving a polynomial equation: Some history and recent progress. *SIAM Review* 39(2):187–220.

Smale, S. 1981. The fundamental theorem of algebra and complexity theory. *Bulletin of the American Mathematical Society* 4:1–36.