Computing the Uncomputable

On non-Turing models of computation

BY JOHN L. CASTI

THE REAL AND THE COMPUTABLE

In everyday life, it's common to talk about doing a "computation." For example, we can speak informally about balancing a checkbook, computing the miles per gallon of gasoline obtained by our car, or converting US dollars into Japanese yen. Each of these operations involves processing numbers in a particular way to obtain the desired result. The procedure by which we process the numbers is an example of what's more formally termed an *algorithm*. The problem faced by theoreticians is how to convert this very common, but informal, notion of carrying out a computation into a precise, formal mathematical structure within which we can actually prove things about the properties of algorithms. Such a formal mathematical system for computation, like the Turing machine, is technically called a *model* of the computational process. Since this use of the term "model" as a kind of formal interpretation of the informal is a bit different from the notion of a model in everyday language, here is an example showing what I mean by the term here.

Suppose we have two sets of elements, call them *V* and *B*. For the moment, we say nothing about the structure of these sets; they are pure abstract objects. Now let's assume that these two sets satisfy the following postulates:

1. Any two members of *V* are contained in just one element of *E*.
2. No member of *V* is contained in more than two members of *E*.
3. The members of *V* are not all contained in a single member of *E*.
4. Any two members of *E* contain just one member of *V*.
5. No member of *E* contains more than two members of *V*.

At first glance, this looks like a pretty abstract, arid, and basically uninteresting set of assumptions. Nevertheless, it is possible to use the traditional tools of logical inference to prove various properties about the sets *V* and *E* from these postulates. For instance, it is fairly straightforward to prove that the set *V* contains exactly three members. It's of crucial importance to understand whether a given set of axioms, like the assumptions 1–5 above, are consistent; that is, can never lead to mutually contradictory theorems. A model for the postulates 1–5 helps answer this question.

Suppose we let *V* be the set of points constituting the vertices of a triangle, and regard the elements of *E* as the lines making up the triangle's edges. Furthermore, we shall interpret the phrase "a member of *V* is contained in a member of *E*" to mean that a point that is a vertex of the triangle lies on a line that is an edge of the triangle. With these interpretations, each of the postulates 1–5 above translates into a true statement about triangles. For example, the first postulate asserts simply that any two points that are vertices of the triangle lie on just one line that is an edge. So in this fashion the set of

John Casti received his Ph.D. in mathematics under Richard Bellman at the University of Southern California in 1970. He worked at the RAND Corporation in Santa Monica, CA, and served on the faculties of the University of Arizona, NYU, and Princeton before becoming one of the first members of the research staff at the International Institute for Applied Systems Analysis (IIASA) in Vienna, Austria. In 1986 he left IIASA to take up his current post as a Professor of Operations Research and System Theory at the Technical University of Vienna. He is also a resident member of the Santa Fe Institute in Santa Fe, New Mexico, USA. In addition to numerous technical articles and a number of research monographs, Professor Casti is the author of several volumes of trade science, including Paradigms Lost, Searching for Certainty, Complexification, *and* Would-Be Worlds.

© 1997 John Wiley & Sons, Inc., Vol. 2, No. 3
CCC 1076-2787/97/03007-06

postulates can be seen to be consistent. A geometrical view of this use of a triangle as a *model* of the postulates 1–5 is shown in Figure 1.

Just as a triangle models the abstract postulates above, so does the Turing machine serve as a model for what we mean in formal mathematical terms by a "computation." And this model dictates exactly what is and is not computable. But there are other possible models for a computation, quite different in form than the Turing-machine model. These alternatives, in turn, determine their own sets of computable quantities. What's at issue is whether any of these models lead to a set of computable quantities that is larger than that obtained with the Turing model. At present, no one really knows. But as a bit of speculation it's worth considering what some of these other models look like.

A lot of computational problems that scientists encounter are expressed in terms of real numbers, not integers. For example, finding the solution to a differential equation or calculating the inverse of a real matrix. The Turing-machine model of computation is not very well-suited for giving us information about such computational processes, defined as it is in terms of binary strings of integer inputs and outputs. To fit the real numbers into this setup can be done, but it's a bit of a stretch and involves some technical delicacies that cloud the issue of computability as much as clarifying it.

A few years ago, the well-known Berkeley mathematician Steven Smale began to take an interest in this question. As part of his effort to understand the theoretical basis of computation over things like real and complex numbers, Smale started buttonholing colleagues in the hallway and coffee room, asking them the question: What do you mean by an algorithm? One reply was, a Fortran program. Together with fellow mathematician Michael Shub and computer scientist Lenore Blum, Smale developed this notion of a Fortran program as an algorithm into a new model of computation, one that easily accommodates computations that operates on either integers or real numbers. This model is sometimes called a *flowchart machine*, or more simply, a *BBS machine*, after the initials of its creators.

Abstractly, a BBS machine is a finite set of nodes connected by arcs that can be traversed in only a single direction (technically, a finite directed graph). These nodes come in five different flavors: input, output, computation, branching, and memory access. To see what a particular BBS machine looks like, let's recall Newton's method for finding the square root of a number, say, 2.

Suppose we are given the equation $x^2 - 2 = 0$. We would like to find a root of this equation, that is, a real number x^* such that $x^{*2} = 2$. More than two centuries ago, Newton discovered a procedure for successively approximating to such a root. He used the approximation scheme

$$x_{k+1} = \frac{1}{2x_k} + \frac{1}{x_k}, \qquad k = 0,1,2,\ldots$$

If we want to find a real number that approximates $\sqrt{2}$ to an accuracy ε, then we would start by guessing a starting number x_0. Newton's scheme is then used to generate approximations x_1, x_2, and so forth, until we arrive at a number x, such that $\|x^2 - 2\| < \varepsilon$.

Newton's method for finding the root of an equation is an example of an algorithm, a mindless, mechanical procedure that is followed step-by-step from a given input (the starting number x_0) to a desired output (the number x, such that $x^2 - 2$ is no larger than the specified accuracy ε). The BBS machine shown in Figure 2 formalizes the steps of this procedure.

One of the primary motivations for development of the BBS machine is to have the power of mathematical analysis available for problems defined over the real or complex numbers, while at the same time having at our disposal the well-developed theory of Turing computability over the integers.

The diagram in Figure 2 can be mathematically formalized so as to give a model for computing the square root of 2. And, in fact, the general idea extends to a more general BBS machine for calculating the root of any equation $f(x)=0$, where x may take its values in any ring of numbers, such as the real or complex numbers, as well as the integers. Of course, one must realize that this is a *theoretical* model of computation, and its implementation in actual hardware depends on the ability of the hardware to do perfect (i.e., error-free) arithmetic operations over the number field involved. In the above example of calculating the square root of 2 over the real numbers, this means that the operations of addition and division called for by Newton's procedure have to be carried out with no round-off error, an impossibility with existing digital computers having a fixed word length. But this does not mean that it's necessarily impossible to develop such "real-number machines." In fact, there are those who think that analogue computers may be produced to carry out just such calculations.

One of the primary motivations for development of the BBS machine is to have the power of mathematical analysis available for problems defined over the real or complex numbers, while at the same time having at our disposal the well-developed theory of Turing computability over the integers. One of the first big results of BSS using their machine involved the issue of decidability. Given a subset S of inputs, one says that a machine decides S if upon given *any* input x, the machine outputs YES if x is in S and NO if x is not in the subset S. This notion was used by BSS to address the question of whether a given starting value for Newton's method would lead to a set of approximations $x_1, x_2, \ldots$ that would converge to an actual root of an equation. In particular, BSS discovered that if the equation is a polynomial $p(z)$ with complex coefficients, and if the polynomial has at least three distinct complex roots, then there is no BSS machine that can decide whether a given starting value z_0 will converge to a number z such that $p(z) = 0$.

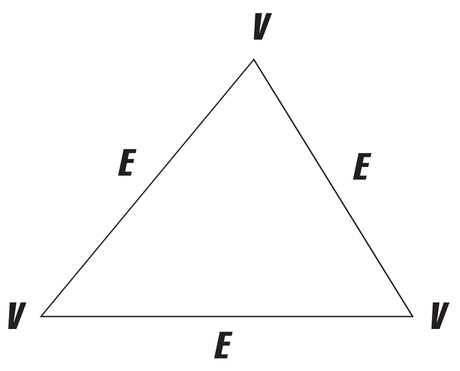
Using the BSS machine, it's possible to extend the idea of *NP*-completeness to problems involving real and complex

© 1997 John Wiley & Sons, Inc.

numbers. For instance, one famous *NP*-complete problem for the integers is Hilbert's 10th Problem, which asks for a general procedure (read: an algorithm) that will tell if a given polynomial equation with integer coefficients has a solution (read: root) in integers. The famous Fermat Problem, solved recently by Andrew Wiles, is exactly a question of this sort for the equation $x^n + y^n - z^n = 0$, where n is an integer greater than 2. Here the question is whether for any such n, there are integers x, y, and z satisfying this equation. Wiles's proof shows that there are not, a result confirming the famous conjecture to this effect made by Fermat over two centuries ago.

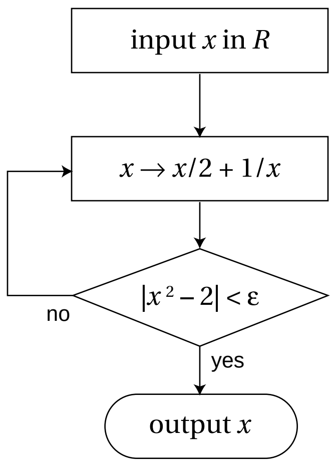
FIGURE 1

THE BLUM-SHUB-SMALE MODEL



A triangle as a model for the set of postulates about the sets *V* and *E*.

FIGURE 2



A BBS machine for finding the square root of 2.

An analogous question involving real numbers rather than integers is: Given a polynomial $p(x)$ of degree 4 with real coefficients, is there a real root of the equation? That is, does there exist a procedure for deciding whether there exists a real number x, such that $p(x) = 0$. BSS showed that this problem is *NP*-complete in exactly the same sense that the Hilbert's 10th Problem is *NP*-complete for integers. Many other results along these same lines are reported in the papers cited in the references for this chapter. The main point to bear in mind is that the BSS machine shows that Turing's way of defining what we mean by a computation is just one of many ways this notion can be formalized, albeit a very convenient one both mathematically and for physical implementation in actual hardware. But there are others. Here's a model based on the processes going on in every cell in a living organism, a so-called *DNA computer*.

DNA COMPUTERS

In 1994, Leonard Adleman of the University of Southern California published an article in *Science* that threatens a revolution in our view of computation. In this research, Adleman coaxed the strands of DNA to link-up in just the right way to solve a "hard" computational problem, one very closely related to the famed Traveling Salesman Problem. The puzzle, known as the *directed Hamiltonian path problem (DHPP),* involves finding a path through a group of towns that are connected in various ways by one-way roads. A Hamiltonian path in such a network of towns is one that passes through each town in the network once and only once. The DHPP asks if a given network has such a path, as well as to produce such a path if one exists.

The DHPP is an example of an *NP*-complete problem; it is easy to check any particular path to see if it is indeed a Hamiltonian path, but it's very difficult to find such a path in the first place. The difficulty, of course, is that all known algorithms for finding a path that passes through each town once and only once have a worst-case complexity that is exponential in the site of the problem. This means that there are networks of towns of modest size, say N towns, that require an amount of computing time that grows exponentially in N. Of course, this would be no difficulty at all if we could check all possible paths at once. This is exactly what Adleman's "DNA computer" allowed him to do. Here's how he used the physical properties of DNA to accomplish this seemingly herculean task.

DNA is nature's way of encoding vast amounts of information in a very compact fashion. For instance, a single strand of DNA contains all the information to generate an elephant, a human, or for that matter, a pterodactyl. Adleman used this information storage capacity of DNA to encode the towns and roads of a network and the paths through the graph in strands of DNA. He then constructed a laboratory setup that enabled him to carry out the four steps in the following algorithm:

1. Generate a multitude of paths through the network.
2. Remove all paths except those that begin with the designated starting town and end with the specified destination town.
3. Remove all paths except those that visit exactly N towns, where N has been given as the number of towns to be visited in a Hamiltonian path.
4. Remove all paths except those that visit every town.

Any strands of DNA that remain in the "soup" after these steps have been performed must correspond to Hamiltonian paths through the network of towns.

The problem Adleman actually solved was that of finding a specific path through a network of towns. For example, consider the four cities New York, Paris, Vienna, and Tokyo. Nonstop flights are scheduled only from New York to Paris, Paris to Vienna, Paris to Tokyo, and Vienna to Tokyo. A question one might ask is, By which route from New York to Tokyo can a traveler visit all the cities, tak-

© 1997 John Wiley & Sons, Inc.

ing only three journeys? In this case the answer is obvious: Fly from New York to Paris, then to Vienna, then on to Tokyo. But if the problem involved all the major cities in the world, and all the connecting flights, the number of possible itineraries would become astronomically large.

Adleman solved a seven-city problem of this type by encoding the details into single strands of DNA. The double helix of DNA is formed of two complementary strands of the four DNA bases, labeled A, T, G, and C. The base A is the complement of T—that is, it can bind only to T. Similarly, G is the complement of C. One strand of the DNA molecule is thus a complementary mirror image of the other. Adleman randomly selected single strand codes to represent each city—say, ATGCGA for New York, TGATCC for Paris, GCTTAG for Vienna. Then the strand representing each flight path might be defined by the last three code letters of the city the flight was leaving, and the first three code letters of the destination. So a flight New York-to-Paris would be coded CGATGA in the above four-city example.

Using genetic engineering, it is possible to manufacture single DNA strands to order. Adleman mixed in his test tube the complementary strands of the DNA coding for the cities (ACTAGG for Paris, say) with the flight path strands, and, as they joined to form double helices, the flight path strings acted as complementary bridges to bind the city strings together. Molecules for all possible combinations of flights formed, but given that billions of molecules were reacting, it was almost certain that a molecule representing the correct flight path would be in the final mixture.

The problem was to identify the molecule that represented a solution to the problem from all the zillions of other molecules floating around in the DNA soup. This magic molecule has the property that it spans the length of all the city codes, beginning with the starting city and ending with the destination, and it contains the codes of all other cities en route once and only once. Knowing this, Adleman was able to isolate the molecule representing the solution using standard techniques of molecular biology. He then checked by hand the order in which its building blocks had been put together, confirming the correct sequence of cities.

The essence of Adleman's approach to computation is to use the peculiar physical properties of DNA to check all possible candidates for the solution to a mathematical problem at one go. This is also the idea underlying an entirely different model of computation. This is the notion of harnessing the weird ways of quantum mechanics to perform computations. Let me take a brief look at this idea as the last stop on this tour of non-Turing models of computation.

The essence of Adleman's approach to computation is to use the peculiar physical properties of DNA to check all possible candidates for the solution to a mathematical problem at one go.

QUANTUM COMPUTERS

In 1982, Richard Feynman published an article titled, "Simulating Physics with Computers." In this speculative piece, Feynman suggested harnessing the weird ambivalence of quantum-mechanical states of particles like electrons to compute at a pace that would far exceed the fastest possible conventional computer.

The phenomenon upon which a quantum computer rests is the ability of a quantum system, say an atom or a single photon, to be in more than one quantum state at the same time, what in the vernacular of the quantum theorist is called *superposition.* So, for instance, the spin of a photon can be in both the up and down states simultaneously. If we represent these two states by 0 and 1, respectively, then calculations on the superposition act on both values at once. Thus, a quantum computer containing n photons or atoms in superposed states could do a calculation on 2^n numbers simultaneously—a degree of parallelism that is inconceivable for everyday, classical computers.

There are a couple of drawbacks to this rosy picture, however. The first is that the process of quantum-mechanical measurement limits (via the Heisenberg Uncertainty Principle) the amount of information that can be extracted from a quantum computer. The second is that quantum superpositions are delicate, fragile things; any contact with the environment sets off a process called *decoherence*, which collapses the superposition to just one of the many possible states the quantum object can occupy. This collapse, of course, eliminates entirely the advantage of using a quantum, rather than classical, computer.

A good question to ask is, Why bother trying to build such a gadget as a quantum computer? Until rather recently, there was no convincing evidence to support the idea that a quantum machine could compute anything that could not be done equally well with a classical computer. But in 1992, Peter Shor of AT&T Bell Laboratories published a quantum-mechanical algorithm for factoring large numbers into their prime components that showed the superiority of using quantum computers over their classical counterparts. Factoring large numbers is a task of not only theoretical, but practical, importance in cryptography, which resulted in Shor's work getting a lot of attention. In all known classical factoring algorithms, the amount of time needed to find the prime factors of a number grows as an exponential function of the size of the number (for those with a bent for figures, this factor is approximately $\exp(L^{1/3})$, where L is the length of the number in question). Shor's quantum algorithm requires a time proportional to L^2, a dramatic reduction over the classical methods when L becomes large. So dramatic, in fact, that it transforms factoring from a computationally "hard" to a computationally "easy" problem.

The question surrounding the feasibility of Shor's method in practice (leaving aside the not-entirely-simple question of actually constructing a quantum machine) is whether the exponentially fast collapse of the superposition of states when exposed to the environment

© 1997 John Wiley & Sons, Inc.

would swamp the exponential increase in computing speed for the factorization scheme. Careful estimates by Shor, Wojciech Zurek, and others have shown that quantum computation can still be useful in factoring as long as a sufficiently low decoherence rate can be maintained. Basically, these results rely on Shor's discovery of quantum analogues of the error-correcting codes used in conventional computers. These standard schemes for error correction cannot be used in the quantum case, however, because they involve actually reading the state of the computer as the computation unfolds and creating redundant states to get any erroneous calculation back on track. The difficulty in the quantum situation is that information disappears in a quantum system as soon as you look at it.

The trick to creating an error-correcting scheme for the quantum case is to encode the state of a single quantum bit as a combination of states in a multiple-bit system. In other words, Shor spreads the information in a single bit across nine such bits. In this way, the original quantum information is preserved even if an error occurs in one of these nine bits. Seth Lloyd of MIT states that the existence of such error-correcting codes came as "quite a surprise. Before Shor came up with this idea, nobody thought it was possible." One problem noted by Lloyd, however, is that error correction, being a computation all its own, runs the same risk of making mistakes as in the original computation. At the moment, nobody has any good idea of how to address this "second-order" difficulty in quantum computation.

While there is no room here to enter into the technical details of various approaches people have presented for actually constructing a quantum computer, there appear to be no physical or logical barriers to be overcome—just massive engineering and technological ones. So while it may be some time before we find ourselves with a quantum machine on our desks, happily buzzing away factoring numbers several hundred digits in length, the possibility for such machines is clearly evident. And if the history of technology teaches us anything, it's that once something is possible, it becomes almost mandatory. With this thought in mind, let me close this discussion of non-Turing models of computation by considering briefly the basis for the exalted position that the Turing model plays in the modern theory of computation.

THE TURING-CHURCH THESIS

In the high-altitude world of theoretical computer science, a principle has sprung up to the effect that we agree to equate the informal notion of "carrying out a computation" with the formal model of the Turing machine. This assumption has been dignified by the label "thesis," and is generally called the *Turing-Church Thesis.* If it is correct, then any operation or process that could be termed a computation must be equivalent to running some program on a Turing machine. In particular, this would mean that if a quantity is not computable on a Turing machine, then it's just plain not computable. Finis! It remains true to this day that no one has yet produced a knockdown argument to show that any model of computation computes a larger class of objects than the Turing-machine model. But this is a far cry from saying that such a model does not exist.

Just to see how extreme the Turing-Church Thesis actually is, a few years ago mathematician Ian Stewart half-jokingly suggested the idea of what he called, The Rapidly-Accelerating Computer (RAC). His goal was to show exactly what it is about computing machines that gives rise to things like the unsolvability of the Halting Problem and uncomputable numbers. Basically, the problem is the assumption that it takes a *fixed,* finite amount of time to carry out a single step in a computation. For his idealized computer, Turing assumed an infinite amount of memory. Stewart, on the other hand, considers the RAC, whose clock accelerates exponentially fast, with pulses separated by intervals of $^1/_2$, $^1/_4$, $^1/_8$, ... seconds. So the RAC can cram an infinite number of computational steps into a single second. Such a machine would be a sight to behold as it would be totally indifferent to the algorithmic complexity of any problem presented to it. On the RAC, everything runs in *bounded* time.

It remains true to this day that no one has yet produced a knockdown argument to show that any model of computation computes a larger class of objects than the Turing-machine model.

The RAC can calculate the incalculable. For instance, it could easily solve the Halting Problem by running a computation in accelerated time and throwing a switch if and only if the program stops. Since the entire procedure could be carried out in no more than one second, we then only have to examine the switch to see if it's been thrown. The RAC could also prove or disprove famous mathematical puzzles like Goldbach's Conjecture (every even number greater than 2 is the sum of two primes). What's even more impressive, the machine could prove all possible theorems by running through every logically valid chain of deduction from the axioms of set theory. And if one believes in classical Newtonian mechanics, there's not even a theoretical obstacle in the path of actually building the RAC.

In Newton's world, we could model the RAC by a classical dynamical system involving a collection of interacting particles. One way to do this, suggested by Z. Xia and J. Gerver, is to have the inner workings of the machine carried out by ball bearings that speed up exponentially. Because classical mechanics posits no upper limit on the velocities of such particles, it's possible to accelerate time in the equations of motion by simply reparameterizing it so that infinite subjective time passes within a finite amount of objective time. What we end up with is a system of classical dynamical equations that mimics the operations of the RAC. Thus, such a system can compute the uncomputable and decide the undecidable.

Of course, just like Turing's infinite-memory machine, the RAC is impossible in the real world. The problem is that at the nitty-gritty level of real material objects like logical gates and integrated circuits, there is a theoretical upper bound to the rate of information transfer (i.e., velocities). As Einstein showed, no material object can exceed the velocity of light. Thus, there is no RAC and, hence, no devices to complete the incompletable.

But the fact that we can never construct a RAC in no way precludes the possibility that some analogue device-like a DNA computer or a quantum machine cannot be made that will allow us to transcend the Turing limit, and, hence, compute the uncomputable.

REFERENCES

1. L. Blum, M. Shub, and S. Smale: On a theory of computation and complexity over the real numbers: NP-Completeness, recursive functions, and universal machines. Bulletin of the American Mathematical Society, 21: pp. 1-46, 1989.
2. DNA Based computers. R. Lipton and E. Baum (Eds.) DIMACS Vol. 27, American Mathematical Society, Providence, RI, 1996.
3. L. Adleman: Molecular computation of solutions to combinatorial problems. Science, 266: pp. 1021-1024, 1994.
4. R. Feynman: Simulating physics with computers. International Journal of Theoretical Physics, 21: pp. 467-488, 1982.
5. I. Stewart: The dynamics of impossible devices. Nonlinear Science Today, 1:4 pp. 8-9, 1991.

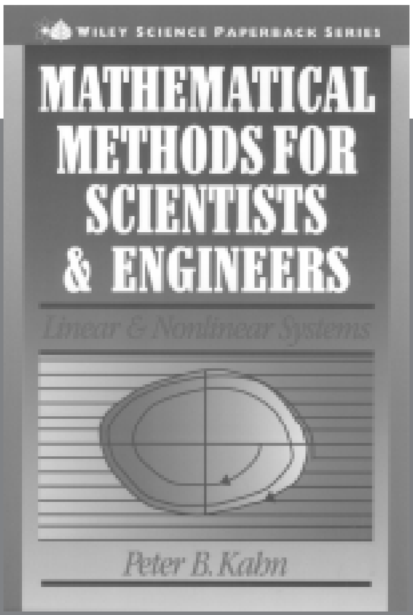
now available in paper . . .

Informal, Accessible Treatments of Linear and Nonlinear Systems in a Broad Range of Science and Engineering

Mathematical Methods for Scientists & Engineers

Linear & Nonlinear Systems

Peter B. Kahn, ***State University of New York at Stony Brook***


WILEY SCIENCE PAPERBACK SERIES
MATHEMATICAL METHODS FOR SCIENTISTS & ENGINEERS
Linear & Nonlinear Systems
Peter B. Kahn

Introduces readers to both linear and nonlinear problems and the associated models that occur in different fields of science and engineering. The first part of the book develops a mastery of linear systems with an emphasis on perturbation or approximation techniques. Special attention is given to asymptotic methods. In the second half, Kahn extends the discussion to nonlinear problems, including weakly nonlinear oscillatory systems and nonlinear difference equations. The two parts merge smoothly: many of the nonlinear techniques arise from the study of linear systems.

This book helps develop tools of analysis that can be used to treat problems that do not admit a simple solution in a closed form, but that have their origin in familiar, solvable problems. The author stresses important topics by permitting them to arise repeatedly in a variety of ways. A few basic examples recur throughout the exposition, guiding students with robust, familiar paradigms. Some of the numerical and graphical solutions have been generated with the computer software packages Lotus 1-2-3® and Math CAD®.

To order call:
1-800-879-4539

1990 • 0-471-16611-1 • 469 pp. • $38.00 • softcover

John Wiley & Sons, Inc.

© 1997 John Wiley & Sons, Inc.