DOI:10.1145/2500890

How to test the usefulness of computation for understanding and predicting continuous phenomena.

BY MARK BRAVERMAN

Computing with Real Numbers, from Archimedes to Turing and Beyond

REAL NUMBERS ARE at the center of our mathematical reasoning about the world around us. Computational problems, from computing the number π to predicting an asteroid's trajectory, all deal with real numbers. Despite the abundance of inherently continuous problems, computers are discrete, finite-precision devices. The need to reason about computing with real numbers gives rise to the kind of fascinating challenges explored here.

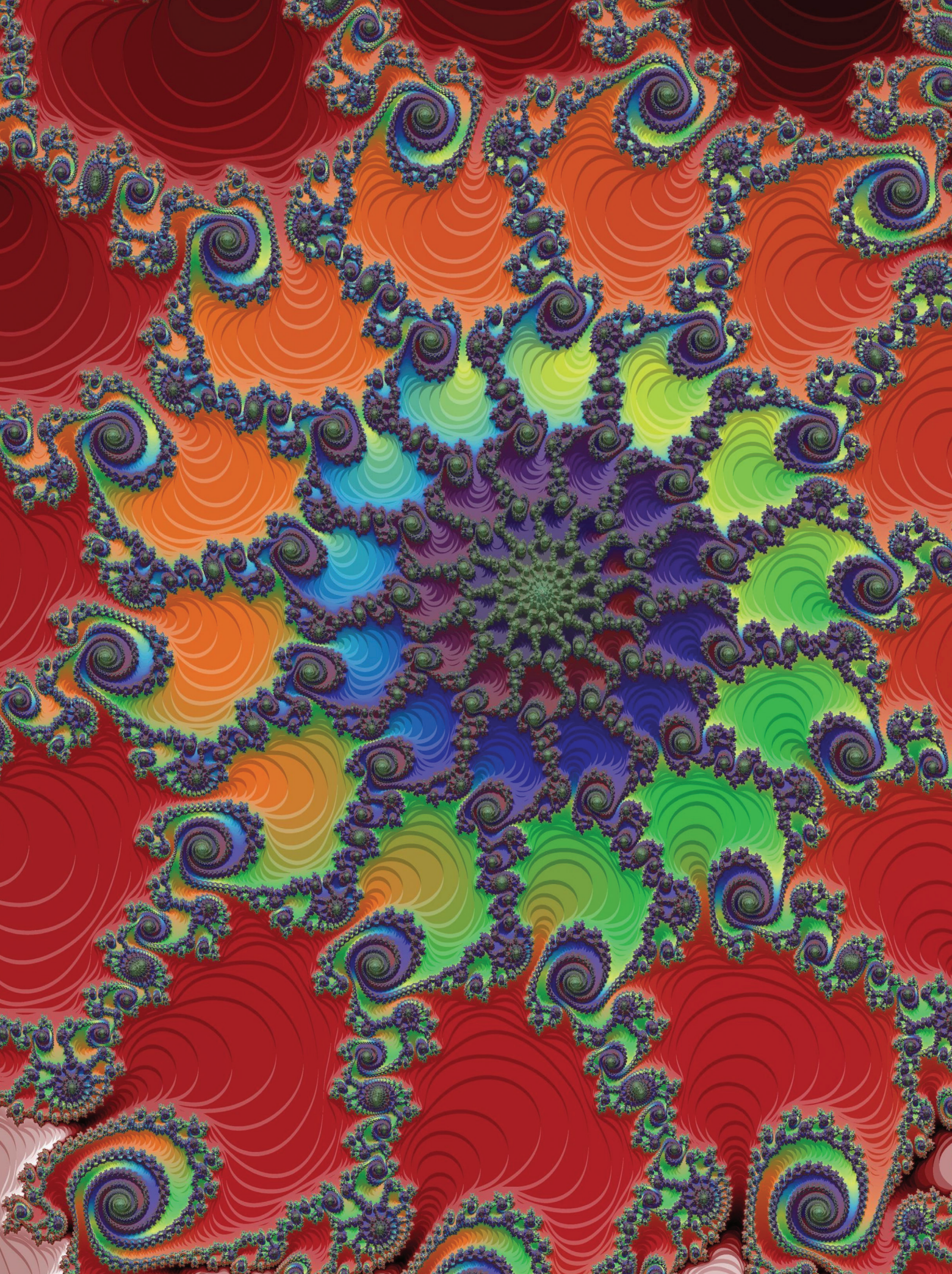
We are so immersed in numbers in our daily lives it is difficult to imagine humans once got by without them. When numbers were finally introduced in ancient times, they were used to represent specific quantities (such as commodities, land, and time); for example "four apples" is just a convenient way to rephrase "an apple and an apple and an apple and an apple"; that is, numbers had algorithmic meaning millennia before computers and algorithmic thinking became as pervasive as it is today. The natural numbers 1, 2, 3, . . . are the easiest to define and "algorithmize." Given enough time (and apples), one can easily produce a pile with any natural number of apples.

Fractions are not as easy to produce as whole natural numbers, yet the algorithm for them is fairly straightforward. To produce 2/3 of an apple, one can slice an apple into three equal parts, then take two of them. If one considers positive rational numbers, there is little divergence between the symbolic representation of the number and the algorithm one needs to "construct" this number out of apples; the number practically shouts a way to construct it. These numbers were the ones that populated the world of the ancient Greeks (in Archimedes's time) who often viewed numbers and fractions through the lens of geometry, identifying them with geometric quantities. In the geometric language, natural numbers are just integer multiples of the unit inter-

» key insights

- **The study of algorithms dealing with real numbers and functions over the reals requires extending the reach of traditional computability theory but is a notable challenge for mathematicians and computer scientists.**
- **The theory of computation over the reals can be applied to the study of computational hardness of dynamical systems involving a range of natural and artificial phenomena.**
- **Predicting a system's long-term properties is easy in some cases; in others it can be as hard as trying to solve the undecidable Halting Problem.**

IMAGE BY PATRICK CATUDAL



val, and positive rational numbers are integer fractions of these intervals.

It was tempting at the time to believe that all numbers, or all possible interval lengths, are rational and can be constructed in this manner. However, it turns out not to be the case. The simplest example of an irrational number is $\sqrt{2}$. The number $\sqrt{2}$ is easily constructed geometrically (such as by using a ruler and compass) and is the length of the diagonal of a 1×1 square. On the other hand, a simple elegant proof, first given by the Pythagorean philosopher Hippasus, shows one cannot write $\sqrt{2}$ as m/n for integers m and n. Hippasus's result was controversial at the time since it violated the belief that all mathematical quantities are rational. Legend has it Hippasus was drowned for his theorem. History offers many examples of scientists and philosophers suffering for their discoveries, but we are not aware of another example of a mathematician being punished for *proving* a theorem. In modern terms, the conflict can be framed through a question: What family of algorithms suffices if one wants to compute all real numbers representing plottable lengths?; Hippasus's opponents supposed integer division would suffice.

Even more intriguing is the number π, which represents the circumference of a circle of diameter 1. Perhaps the most prominent mathematical constant, π can also be shown to be irrational, although the proof is not as simple as for $\sqrt{2}$ and was not known in ancient times. We cannot represent either $\sqrt{2}$ or π as rational fractions. We can "construct" them from mesh wire using their geometrical interpretations, but can we also figure out their numerical values? Unlike the names "4" and "2/3" the names "$\sqrt{2}$" and "π" are not helpful for actually evaluating the numbers. We can calculate approximations of these numbers; for example, for π, we can write

$$3.1415926 < \pi < 3.1415927$$

or perhaps we can follow Archimedes, who carried out the earliest theoretical calculations of π, and write

$$\frac{223}{71} < \pi < \frac{22}{7}.$$

One reason numbers and mathematics was developed in the first place was to understand and control natural systems.

Both representations are correct, giving us a good handle on the value of π, but both have limited precision, thus losing some information about the true value of π. All real numbers can be written using their infinite binary (or decimal) expansion, which can be used to name the number, specifying it unambiguously.

The infinite representation π = 3.1415926 . . . unambiguously specifies the number π. Alas, however, we cannot use it to write π in a finite amount of space. An ultimate representation would take a finite amount of space but also allow us to compute π with any desired precision. In modern language, such a representation should be algorithmic. As there are many formulas for π, there are likewise many ways to represent π this way; for example in approximately the year 1400, Madhava of Sangamagrama gave this formula

$$\pi = 4\,\Sigma_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots \quad (1)$$

It allows us to compute π with any precision, although the convergence is painfully slow; to compute the first n digits of π one needs to take approximately 10^n terms of this sum. Many elegant formulas for computing π have been devised since, some allowing us to compute π in time polynomial in the number of digits. One such formula, known as the Bailey-Borwein-Plouffe formula, is given by

$$\pi = 4\,\Sigma_{k=0}^{\infty}$$
$$\left[\frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)\right]\ldots \quad (2)$$

The fact that the terms in formula (2) decrease exponentially fast in k causes the sum to converge rapidly. Mathematically speaking, formulas (1) and (2) are both valid "names" for π, although the latter is better because it corresponds to a much more efficient algorithm.

Can all numbers be given names in a way that allows us to compute them? No, as it turns out. Surprisingly, it took until Alan Turing's seminal paper[4] in 1936 to properly pose and answer the question. Turing had to overcome a profound philosophical difficulty. When showing a real num-

ber is computable, we would need only to describe an algorithm able to compute it with any prescribed precision, as we did with the number π. In showing that a number $x \in \mathbb{R}$ is not computable, we need to rule out all potential ways of computing x. The first major step in any such proof is formalizing what "computing" means by devising a model of computation. This is exactly what Turing did, defining his famous Turing Machine as an abstract device capable of performing all mechanical computations. Turing's paper started the modern field of computability theory. Remarkably, it happened about a dozen years before the first computers (in the modern sense of the word) were built. Turing used his new theory to define the notion of computable numbers. Not surprisingly, a modern reinterpretation of Turing's definition says a number x is computable if we can write a C++ or Java program that (given sufficient time and memory) can produce arbitrarily precise approximations of x. One of Turing's key insights was the Halting Problem $\mathcal{H}$ (which takes an integer n and outputs $\mathcal{H}(n) = 1$ if and only if $n = [P]$ is an encoding of a valid program P and P terminates) is "undecidable"; no algorithm exists that, given a program P, is capable of deciding whether or not P terminates.

The Halting Problem allows us to give a specific example of a noncomputable number. Write down the values of the function $\mathcal{H}(\bullet)$; the number

$$X_{\mathcal{H}} = 0.\mathcal{H}(1)\mathcal{H}(2)\mathcal{H}(3)\ldots = \sum_{n=1}^{\infty} 10^{-n}\mathcal{H}(n).$$

is not computable, since computing it is equivalent to solving the Halting Problem. Fortunately, "interesting" mathematical constants (such as π and e) are usually computable.

One reason numbers and mathematics was developed in the first place was to understand and control natural systems. Using the computational lens, we can rephrase this goal as reverse-engineering nature's algorithms. Which natural processes can be computationally predicted? Much of this article is motivated by this question. Note, unlike digital computers, many natural systems are best modeled using continuous quantities; that is, to discuss the computability of natural systems we have to extend the discrete model of computation to functions and sets over the real numbers.

Real Functions and Computation

We have established that a number $x \in \mathbb{R}$ is computable if there is an algorithm that can compute x with any prescribed precision. To be more concrete, we say an algorithm $\mathcal{A}_x$ computes x if on an integer input $n \in \mathbb{N}$, $\mathcal{A}_x(n)$ outputs a rational number x_n such that $|x_n - x| < 2^{-n}$. The algorithm $\mathcal{A}_x$ can be viewed as a "name" for x, in that it specifies the number x unambiguously. The infinite-digit representation of x is also its "name," albeit not compactly presented.

What does it mean for a function $f:\mathbb{R}\to\mathbb{R}$ to be computable? This question was first posed by Banach, Mazur, and colleagues in the Polish school of mathematics shortly after Turing published his original paper, starting the branch of computability theory known today as "computable analysis." Now step back to consider discrete Boolean functions. A Boolean function $F:\{0,1\}^*\to\{0,1\}^*$ is computable if there is a program $\mathcal{A}_F$ that given a binary string $s\in\{0,1\}^*$ outputs $\mathcal{A}_F(s)=F(s)$. By analogy, an algorithm computing a real-valued function f would take a real number x as an input and produce $f(x)$ as an output. Unlike the Boolean case, "input" and "output" must be qualified in this context. What we would like to say is given a name for the value $x \in \mathbb{R}$ we should be able to produce a name for the output $f(x)$; that is, we want $\mathcal{A}_f$ to be a program that, given access to arbitrarily good approximations of x, produces arbitrarily good approximations of $f(x)$.

A function $f:(a,b) \to \mathbb{R}$ is computable if there is a discrete algorithm $\mathcal{A}_f$ that, given a precision parameter n and access to arbitrarily good rational approximations of an arbitrary input $x \in (a,b)$, outputs a rational y_n such that

$$|y_n - f(x)| < 2^{-n}.$$

This definition easily extends to functions that take more than one input (such as the arithmetic operations $+: \mathbb{R}\times\mathbb{R}\to\mathbb{R}$ and $\times: \mathbb{R}\times\mathbb{R}\to\mathbb{R}$). As with numbers, all "nice" functions, including those usually found on a scientific calculator, are generally computable. Consider the simple example of the function $f(x) = x^2$ on the interval (0,1). Our algorithm for squaring numbers should be able to produce a 2^{-n} approximation of x^2 from approximations of x. And consider this simple algorithm

SimpleSquare(x,n)

1. Request $q = x_{n+1}$, a rational $2^{-(n+1)}$-approximation of the input x.
2. Output q^2.

Note the algorithm SimpleSquare operates only with rational numbers. To see that the algorithm works, we need to show for all $x \in (0,1)$ the output q^2 satisfies $|x^2 - q^2| < 2^{-n}$. Since x is in the interval (0,1), without loss of generality we may assume q is also in (0,1). Therefore, $|x+q| < |x| + |q| < 2$, and we have

$$|x^2 - q^2| = |x+q|\cdot|x-q| \le 2\,|x-q| < 2\cdot 2^{-(n+1)} = 2^{-n}.$$

This shows the SimpleSquare algorithm indeed produces a 2^{-n} approximation of x^2. Although the function $f(x) = x^2$ is computable on the entire real line $\mathbb{R} = (-\infty,\infty)$, in this case, the algorithm would have to be modified slightly to work.

A more interesting example is the function $g(x) = e^x$, which is defined on the entire real line. Indeed, for any x, we can compute e^x with any precision by requesting a sufficiently good rational approximation q of x and then using finitely many terms from the series

$$e^q = \sum_{n=0}^{\infty} \frac{q^n}{n!} = 1 + \frac{q}{1} + \frac{q^2}{2} + \frac{q^3}{6} + \frac{q^4}{24} + \ldots$$

Throughout the discussion of the computability of these functions, we did not have to assume the input x to a computable function is itself computable. As long as the Request command gives us good approximations of x we do not care whether these approximations were obtained algorithmically. Now, if the number x is itself computable, then the Request commands may be replaced with a subroutine that computes x with the desired precision. Thus if f is computable on (a,b) and $x \in (a,b)$ is a computable number, then $f(x)$ is also a computable number. In particular, since e^x is a computable function and π is a computable num-

ber, e^{π} and $e^{e^{\pi}}$ are computable numbers as well.

One technical limitation of the Request-based definition is we can never be sure about the exact value of the input x; for example, we are unable to decide whether the real-valued input x is equal to, say, 42 or not. Thus the function

$$f(x) = \begin{cases} 1 & \text{if } x = 42.0 \\ 0 & \text{otherwise} \end{cases}$$

is not computable. The reason for this inability is while we can Request x with any desired precision, no finite-precision approximation of x will ever allow us to be sure x is exactly 42.0. If we take the requested precision high enough, we may learn $x = 42.0 \pm 10^{-1,000}$. This still does not mean $f(x) = 1$, as it is possible the first disagreement between x and 42.0 occurs after the 1,000th decimal place (such as if $x = 42 + 2^{-2000} \neq 42.0$). The Request function can be viewed as a physical experiment measuring x. By measuring x we can narrow down its value to a very small interval but can never be sure of its exact value. We refer to this difficulty as the "impossibility of exact computation." More generally, similar reasoning shows only continuous functions may be computable in this model.

On the other hand, and not too surprisingly, all functions that can be computed on a calculator are computable under this definition of function computability. But, as with Turing's original work, the main goal of having a model of computation dealing with real functions is to tell us what *cannot* be done, or proving fundamental bounds on our ability to computationally tackle continuous systems. First we need to explore the theory of computation over the reals a little further.

Computability of subsets in $\mathbb{R}^d$. In addition to numbers and functions we are also interested in computing sets of real numbers; see Figure 1 for example subsets of $\mathbb{R}^2$. Sets we might be interested in include simple geometric shapes (such as a circle), graphs of functions, and the more complicated ones, like the Koch snowflake and the Mandelbrot set. When is a set S in, say, the plane $\mathbb{R}^2$, computable? It is tempting to mimic the discrete case

Figure 1. Examples of subsets of $\mathbb{R}^2$: the graph of $y = c^x$, the Koch snowflake, and the Mandelbrot set.

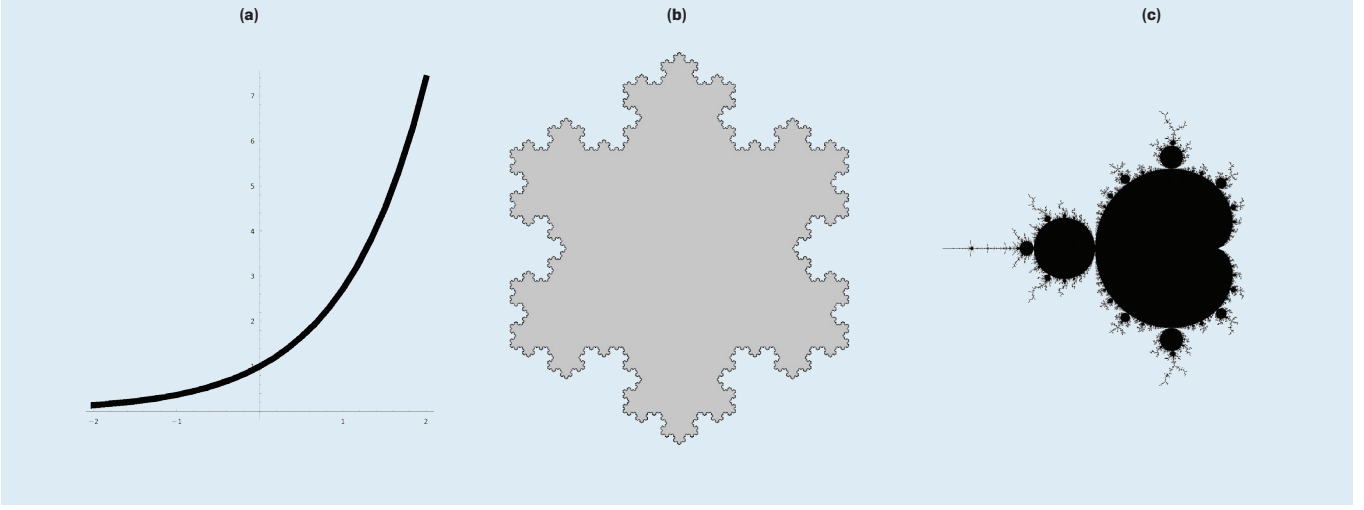
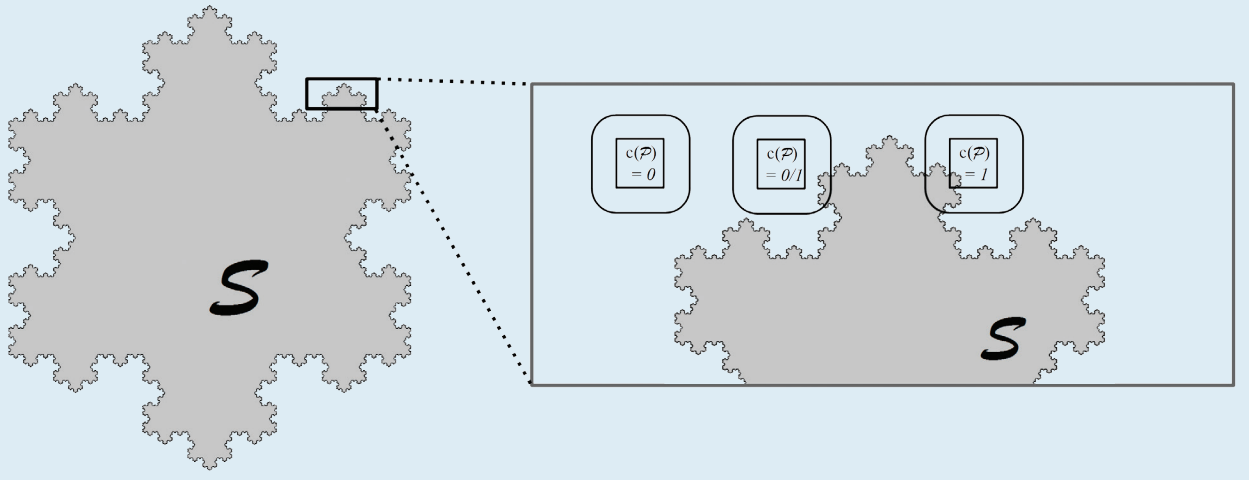


Figure 2. The process of deciding the color $c(\mathcal{P})$ for an individual pixel $\mathcal{P}$.



and say $\mathcal{S}$ is computable whenever the membership function

$$x_{\mathcal{S}}(x) = \begin{cases} 1 & \text{if } x \in \mathcal{S} \\ 0 & \text{otherwise} \end{cases}$$

is decidable. However, this definition involves a serious technical problem, the same impossibility-of-exact-computation problem present in the $x \stackrel{?}{=} 42.0$ example. If x happens to lie on the boundary of $\mathcal{S}$, we will never be able to decide whether $x \in \mathcal{S}$ through a finite number of Request queries. To address this problem we proceed by analogy with the computability of numbers. Rather than try to compute $\mathcal{S}$, we should try to approximate it with any prescribed precision 2^{-n}.

What does it mean to "approximate" a set? There are many ways to address this question, and many "reasonable" definitions are equivalent. First take a "graphical" approach. Consider the process of producing a picture of the set $\mathcal{S}$. The process would consist of many individual decisions concerning whether to color a pixel black or white; for example, we need to make 600 × 600 such decisions per square inch if $\mathcal{S}$ is being printed on a 600dpi printer. Thus a discussion about drawing $\mathcal{S}$ with precision 2^{-n} can be reduced to a discussion about deciding on the color of individual pixels, bringing us back to the more familiar realm of 0/1-output algorithms.

To be concrete, let $\mathcal{S}$ be a subset of the plane $\mathbb{R}^2$ and let

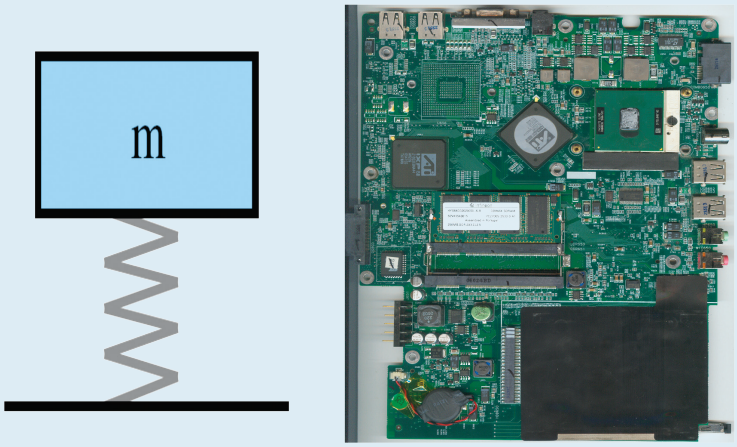
$$P = [x - 2^{-n-1}, x + 2^{-n-1}] \times [y - 2^{-n-1}, y + 2^{-n-1}]$$

be a square pixel of dimensions $2^{-n} \times 2^{-n}$. The coloring of pixels should satisfy the following conditions:

1. If $\mathcal{P}$ intersects with $\mathcal{S}$, we must color it `black`;
2. If $\mathcal{P}$ is 2^{-n}-far from $\mathcal{S}$ we must color it `white`. It is natural to ask why not simply require $\mathcal{P}$ to be colored white if it does not intersect $\mathcal{S}$. The reason is, if we did, we would again run into the impossibility of exact computation. Thus we allow for a gray area in the image; and
3. If $\mathcal{P}$ does not intersect $\mathcal{S}$ but is 2^{-n} close to it, we do not care whether it is colored black or white.

This gray area allows us to avoid the impossibility of exact computation problem while still producing a faithful image of $\mathcal{S}$ (see Figure 2).

Figure 3. One of the "easiest" (left) and one of the "hardest" (right) dynamical systems.



The definition of set computability presented here may seem ad hoc, appearing to be tied in to the way we choose to render the set $\mathcal{S}$. Somewhat surprisingly, the definition is robust—equivalent to the "mathematical" definition of $\mathcal{S}$ being "approximable" in the Hausdorff metric, a natural metric one can define on subsets of $\mathbb{R}^d$. The definition is also equivalent to the distance function $d_{\mathcal{S}}(x)$ that measures how far the point x is from the set $\mathcal{S}$ being a computable function.

Just as "nice" calculator functions are computable, "nice" sets are likewise computable; for example, a circle $C(o,r)$ with center $o = (x, y)$ and radius r is computable if and only if the numbers x, y, and r are computable. Graphs of computable functions are computable sets. Thus the graph of the function $x \mapsto e^x$ is computable. For a more interesting example, consider the Koch snowflake K (see Figure 1b). This fractal set has dimension $\log_3 4$ and lacks a nice analytic description. However, it is computable and is, in fact, the limit set of a sequence of finite snowflakes. Each finite snowflake K_n is just a polygon and thus easily computable. To approximate K we need to draw only the appropriate finite snowflake K_n, exactly how the Koch snowflake is drawn in practice, as in Figure 1b.

Now that we have the notion of computable real functions and real sets we can turn to formulating the computational hardness of natural and artificial systems, as studied in the area of dynamical systems.

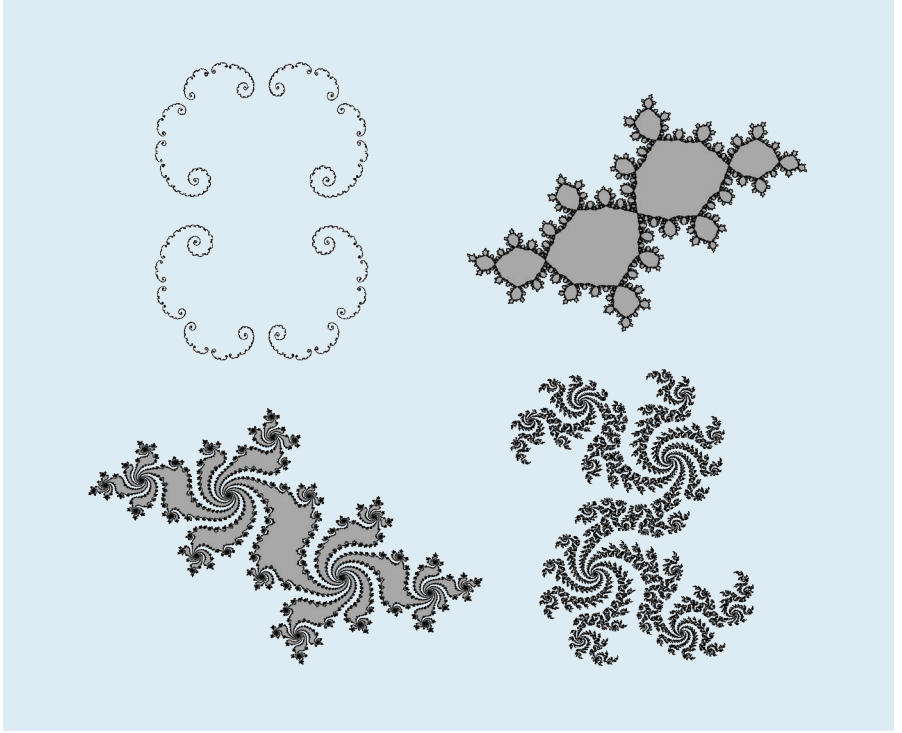
Computing Nature: Dynamical Systems

At a high level, the area of dynamical systems studies all systems that evolve over time. Systems ranging from an electron in a hydrogen atom to the movement of galaxies to brain activity can thus be framed in the context of dynamical systems. A dynamical system consists of a set of states $\mathcal{X}$ and a set of evolution rules $\mathcal{R}$. Evolution of the system occurs over time. The state of the system at time t is denoted by $X_t \in \mathcal{X}$. The time may move either discretely or continuously. If time is discrete, the evolution of the system is given by the sequence $X_1, X_2, X_3, \ldots$, and the rule $\mathcal{R}$ specifies the dependence of X_{t+1} on X_t. If time is continuous, the evolution $\mathcal{R}$ of the system $X_t = X(t)$ is usually given by a set of differential equations specifying the rates of change in $X(t)$ depending on its current state.

As an example, consider a simple harmonic oscillator. A mass in Figure 3 is attached to a spring and is moving in a periodical fashion. Assuming no friction, the system X_t^{osc} evolves in continuous time, and its state at any time is described fully by two numbers: the location of the mass on the line and its velocity. Thus the state X_t^{osc} can be represented by the vector $X_t^{osc} = (\ell(t), \upsilon(t))$, where $\ell(t)$ represents the location of the mass, and $\upsilon(t)$ represents its velocity. The evolution rule of the system is given in this case by high-school physics

$$\begin{cases} \ell'(t) = \upsilon(t) \\ \upsilon'(t) = -\alpha \cdot \ell(t) \end{cases} \quad (3)$$

Figure 4. Example Julia sets.



where α is a parameter that depends on the spring and the mass of the weight. X^{osc} is a very simple system, and we can answer pretty much any question about it; for example, we can solve this system of equations to obtain the full description of the system's behavior

$$X_t^{osc} = (A\sin(\sqrt{\alpha}\cdot t+\phi),\; A\sqrt{\alpha}\cos(\sqrt{\alpha}\cdot t+\phi)) \quad (4)$$

where the parameters A and ϕ depend on the initial condition $X_0^{osc} = (\ell(0),\upsilon(0))$ of the system at time 0; that is, if we know the exact state of the system at time 0, we can compute the state of the system at any time in the future. It is not just the prediction problem that is easy for this system. Using the analytic solution (4) we can answer almost any question imaginable about it; for example, we can describe the set of all possible states the system being released from state X_0^{osc} will reach.

At the other extreme, predicting some dynamical systems in the long run is incredibly difficult. One important set of examples of "hard" dynamical systems comes from computer science itself. Consider the Turing Machine or its modern counterpart, a RAM computer with unlimited RAM, as a dynamical system. The state-space X^{comp} is the (infinite) state space of the computer. The system X^{comp} evolves in discrete time, with X_t^{comp} representing the state of the computer at step t of the execution. The evolution rule $\mathcal{R}$ is the rule according to which the computation proceeds; that is, $\mathcal{R}(X)$ is the state of the computer in the next time step if its current state is X. Call this system the "computer dynamical system."

The computer dynamical system is easy to simulate computationally; all we must do is simulate the execution of the computation. On the other hand, unlike the oscillator example, answering long-term questions about the system is difficult; for example, given an initial condition X_0^{comp}, there is no computational procedure that can tell us whether the system will ever reach a given state Y; determining whether the system will reach a terminating state is equivalent to solving the Halting Problem for programs, that, as discussed, is computationally undecidable. It can likewise be shown that almost any nontrivial question about the long-term behavior of X^{comp} is noncomputable in the worst case.

These examples exist at the two extremes of computational hardness: X^{osc} is a linear dynamical system and fully computable. X^{comp} is (trivially) computationally universal, capable of simulating a Turing Machine, and reasoning about its long-term properties is as computationally hard as solving the Halting Problem. What kinds of systems are prevalent in nature? For example, can an N-body system evolving according to the laws of Newtonian gravity simulate a computer, in which case predicting it would be as difficult as solving he Halting Problem? Is predicting it computationally easy? Or something in-between?

We do not know the answers for most natural systems. Here, we consider an interesting in-between example that is one of the best-studied dynamical systems. We consider dynamics on the set of complex numbers $\mathbb{C}$, or a number of the form $a + bi$, evolving by a quadratic polynomial. For the rest of this discussion, the set of complex numbers is identified with the 2D complex plane, with the number $a + bi$ corresponding to the point (a,b), allowing us to visualize subsets of $\mathbb{C}$ nicely. Let $c \in \mathbb{C}$ be any complex number. Denote

$$p_c(z) := z^2 + c.$$

Define the discrete-time dynamical system X_t^c by

$$X_{t+1}^c = p_c(X_t^c).$$

The polynomial $p_c(z)$ is arguably the simplest nonlinear transformation one can apply to complex numbers, yet this system is already complicated enough to exhibit a variety of interesting and complicated behaviors. In particular, it is impossible to give a closed-form expression for X_t^c in terms of X_0^c as we did with the oscillator example. Dynamical systems of the form X_t^c are studied by a branch of the theory of dynamical systems known as complex, or holomorphic, dynamics. Within mathematics, one of the main reasons for studying these systems is the rich variety of behaviors they exhibit allows us to learn about the behavior of more general (and much more difficult to study) dynamical systems.

Outside mathematics, complex dynamics is best known for the fascinating fractal images it generates. These images, known as Julia sets (see Figure 4), depict a global picture relevant to the long-term behavior of the system X^c. More specifically, J_c is the subset

of initial conditions in the complex plane on which the long-term behavior of X^c is unstable. To understand what this means, we need to take a slightly closer look at the system X^c. Consider an initial point $X_0^c = x_0$, as mapped by $p_c(z) = z^2 + c$

$$x_0 \mapsto x_0^2 + c \mapsto (x_0^2 + c)^2 + c \mapsto \ldots$$

If we start with an x_0 with a very high absolute value, say, $|x_0| > |c| + 2$, then the absolute value of $p_c(x_0) = x_0^2 + c$ will be larger than $|x_0|$, and $|p_c(p_c(x_0))|$ will be larger still and the state of the system will diverge to ∞. The set of starting points for which the system does not diverge to ∞ is called the filled Julia set of the system X^c and is denoted by K_c. The Julia set[a] J_c is the boundary ∂K_c of the filled Julia set.

The Julia set J_c is the set of points around which the system's long-term behavior is highly unstable. Around each point z in J_c are points (just outside K_c) with trajectories that ultimately escape to ∞. There are also points (just inside K_c) with trajectories that always stay within the bounded region K_c. The Julia set itself is invariant under the mapping $z \mapsto z^2 + c$. This means trajectories of points that start in J_c stay in J_c.

The Julia set J_c provides a description of the long-term properties of the system X_c. Julia sets are therefore valuable for studying and understanding these systems. In addition, as in Figure 4, Julia sets give rise to an amazing variety of beautiful fractal images. Popularized by Benoit Mandelbrot and others, Julia sets are today some of the most drawn objects in mathematics, and hundreds of programs for generating them can be found online.

Formally speaking, the problem of computing the Julia set J_c is one of evaluating the function $\mathcal{J}: c \mapsto J_c$. $\mathcal{J}$ is a set-valued function whose computability combines features of function and set computability discussed earlier. The complex input $c \in \mathbb{C}$ is provided to the program through the request command, and the program computing $\mathcal{J}(c) = J_c$ is required to output an image of the Julia set J_c within a prescribed precision 2^{-n}. $\mathcal{J}$ is a fascinating function worth considering in its own right; for example, the famous Mandelbrot $\mathcal{M}$ (see Figure 1c) can be defined as the set of parameters c for which $\mathcal{J}(c)$ is a connected set. It turns out the function $\mathcal{J}(c)$ is discontinuous, at least for some values of c (such as for $c = \frac{1}{4} + 0 \cdot i$). This means for every ε there is a parameter $c' \in C$ that is ε-close to $\frac{1}{4}$ but for which the Julia set $J_{c'}$ is very far from $J_{\frac{1}{4}}$. Due to the impossibility of exact computation, this discontinuity implies there is no hope of producing a single program $P_{\mathcal{J}}$ that computes $\mathcal{J}(c)$ for all parameters c; on inputs close to $c = 1/4$, such a program would need to use request commands to determine whether c is in fact equal to 1/4 or merely very close to it, which is impossible to do.

As with numbers, all "nice" functions, including those usually found on a scientific calculator, are generally computable.

If there is no hope of computing $\mathcal{J}$ by one program, we can at least hope that for each c we can construct a special program P_{J_c} that evaluates J_c. Such a program would still need access to the parameter c through the request command, since only a finite amount of information about the continuous parameter $c \in \mathbb{C}$ can be "hardwired" into the program P_{J_c}. Nonetheless, by requiring P_{J_c} to work correctly on only one input c we manage to sidestep the impossibility of exact computation problem.

Most of the hundreds of online programs that draw Julia sets usually draw the complement $(\overline{K_c}) = \mathbb{C} \setminus K_c$ of the filled Julia set, or the set of points whose trajectories escape to ∞. It turns out that from the computability viewpoint, the computability of $\overline{K_c}$ is equivalent to the computability of J_c, allowing us to discuss these problems interchangeably.[b] The vast majority of the programs follows the same basic logic; to check whether a point z_0 belongs to $\overline{K_c}$, we need to verify whether its trajectory $z_0, p_c(z_0), p_c(p_c(z_0)), \ldots$ escapes to ∞. We pick a (large) number M of iterations. If z_0 does not escape within M steps, we assume it does not escape. To put this approach in a form consistent with the definition of computability of sets in $\mathbb{R}^2$, let $\mathcal{P}$ be a pixel of size 2^{-n}. The naïve decision procedure for determining whether the pix-

a "Julia" is not a first name in this context but rather the last name of the French mathematician Gaston Julia (1893–1978).

b As mentioned here, the computability of $(\overline{K_c})$ and J_c is not equivalent to the computability of the filled Julia set K_c.

el $\mathcal{P}$ overlaps with $\overline{K_c}$ or is 2^{-n}-far from it thus looks roughly like this:

A naïve heuristic for drawing $\overline{K_c}$:

1. Let z_0 be the center of the pixel $\mathcal{P}$;
2. Let $M = M(n)$ be the number of iterations;
3. **for** $i = 0$ **to** M
   3.1. Set $z_{i+1} \leftarrow z_i^2 + c$;
   3.2. **if** $|z_{i+1}| > |c|+2$, **return** "$\mathcal{P}$ intersects $\overline{K_c}$";
4. **if** $|z_{M+1}| \leq |c|+2$ **return** "$\mathcal{P}$ is 2^{-n} far from $\overline{K_c}$"

If the pixel $\mathcal{P}$ is evaluated to intersect with $\overline{K_c}$, it is colored `black`; otherwise it is left `white`. However, there are multiple problems with these heuristics that make the rendered pictures imprecise and sometimes just wrong. The first is we take one point z_0 to be "representative" of the entire pixel $\mathcal{P}$. This approach means even if $\mathcal{P}$ intersects $\overline{K_c}$ or some point $w \in \mathcal{P}$ has its trajectory escape to ∞, we might miss it if the trajectory of z_0 does not escape to ∞. This problem highlights one of the difficulties encountered when developing algorithms for continuous objects. We need to find the answer not just for one point z_0 but for the uncountable set of points located within the pixel $\mathcal{P}$. However, there are computational ways to remedy this problem. Instead of tracing just one point z_0 we can trace the entire geometric shape $\mathcal{P}, p_c(\mathcal{P}), p_c(p_c(\mathcal{P})), \ldots$ and see whether any part of the M^{th} iteration of $\mathcal{P}$ escapes to ∞. This may increase the running time of the algorithm considerably. Nonetheless, we can exploit the peculiarities of complex analytic functions to make the approach work. John Milnor's "Distance Estimator" algorithm does exactly that, at least for a large set of "good" parameters c.

The heuristic also involves a much deeper problem—choosing the parameter $M(n)$. In the thousands of Java applets available online, selection of the number of iterations M is usually left to the user. Suppose we wanted to automate this task or make the program evaluate a "large enough" M such that M iterations are sufficient (see Figure 5 for the effect of selecting an M that is too small); that is, we want to find a parameter M such that if the M^{th} iteration z_M of z_0 did not escape to ∞, then we can be sure no further iterations will escape to ∞, and it is safe to assert z_0 lies within the filled Julia set K_c. Computing such an M is equivalent to establishing termination of the loop

```
Loop(z_0): i ← 0
        while |z_i| ≤ |c|+2
                z_{i+1} ← z_i^2 + c
                i ← i + 1
```

In general, the termination of loops, as with the Halting Problem $\mathcal{H}$ is a computationally undecidable problem. If the loop terminates, we are sure z_0 has escaped. But if the loop keeps running there is no general way of knowing it will not terminate later. There is thus no simple solution to figuring out the appropriate M; the only way to know the loop does not terminate is to understand the system X^c and the set $\overline{K_c}$ well enough. Turning the naïve heuristic into an algorithm necessarily involves a deep understanding of the underlying dynamical system. As with the systems X^{osc} and X^{comp} discussed earlier, it all boils down to understanding the underlying system. Fortunately, complex dynamicists have developed a rich theory around this system since its introduction around 1917 by the French mathematicians Gaston Julia and Pierre Fatou. This knowledge is enough to give precise answers to most questions concerning the computability of K_c, $\overline{K_c}$, and J_c. One can formalize the naïve heuristic discussed here and show that (with slight modifications) it works for the vast majority of values of c.

However, it also turns out there are also noncomputable Julia sets:

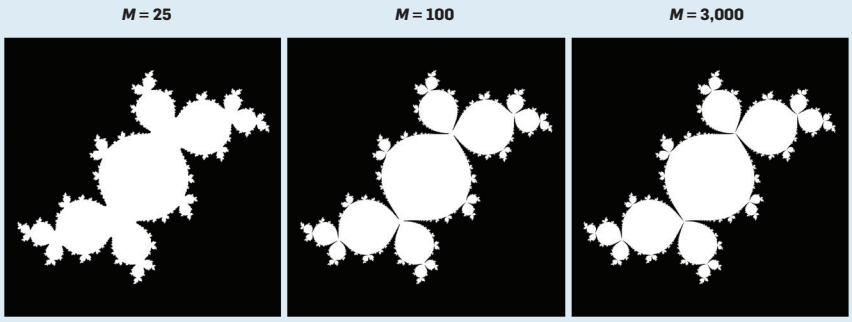
THEOREM 1.[2] *There exist parameters c such that the Julia set J_c is not computable. Moreover, there are such parameters $c \in \mathbb{C}$ that can be produced algorithmically.*

That is, one can produce a parameter c such that drawing the picture of J_c is as computationally hard as solving the Halting Problem. What does such a Julia set look like? All sets in Figure 4 are computable, as they were produced algorithmically for inclusion here. Unfortunately, Theorem 1 means we will most likely never know what these noncomputable Julia sets look like.

The negative result is delicate; in a surprising twist, it does not extend to the filled Julia set K_c:

Systems ranging from an electron in a hydrogen atom to the movement of galaxies to brain activity can be framed in the context of dynamical systems.

Figure 5. Example outcomes of the heuristic algorithm with $c \approx -0.126 + 0.67i$ and $M = 25$, $M = 100$, and $M = 3{,}000$ iteration. Note with the lower values of M the fjords do not reach all the way to the center since the points close to the center do not have time to "escape" in M iterations. The difficulty selecting a "large enough" M is a crucial obstacle in computing the exterior of the filled Julia set K_c.



THEOREM 2.[2] *For all parameters c,* the filled Julia set K_c*:*

THEOREM 2.[2] *For all parameters c, the filled Julia set K_c is computable.*

Is the Universe a Computer?

We have explored three examples of dynamical systems: The first, harmonic oscillator X^{osc}, is simple; its behavior can be calculated in closed form, and we can answer pretty much any question about the long-term behavior of the harmonic oscillator computationally. The second, computer system X^{comp}, is at the opposite extreme; predicting it is computationally hard, and it is (relatively) easy to show it is computationally hard through a reduction to the Halting Problem. The third, complex dynamics, requires an involved approach. For some (in fact, most) parameters c, the long-term behavior of the system X^c is easy in almost any sense imaginable. Showing there are parameters c for which no amount of computational power suffices to compute the Julia set J_c required a full understanding of the underlying dynamical system developed over nearly a century.

Our experience with the computability of Julia sets, as well as the relative success of the field of automated verification at solving undecidable problems in practice,[5] indicates there is likely to be a gap between computability in dynamics in the worst case and in the typical case. This gap means it is possible that while questions surrounding many natural systems (such as the N-body problem and protein assembly) are provably noncomputable in the worst case, a typical case in practice is tractable.

A related interesting possibility is that noncomputable structures in many systems are too delicate to survive the random noise present in all natural systems. Noise is generally viewed as a "prediction destroying" force, in that making predictions in the presence of noise is computationally more difficult. On the other hand, if we are interested in predicting the statistical distribution of possible future states, then noise may actually make the task easier. It is likely there are natural systems that (if implemented with no noise) would be computationally impossible to predict but where the presence of noise makes statistical predictions about the system computationally tractable.

Another lesson from the study of the computational properties of Julia sets is that mapping out which of the Julia sets J_c are and which are not computable requires a nuanced understanding of the underlying dynamical system. It is likely this is the case with other natural dynamical systems; the prerequisite to understanding its computational properties would be understanding its other properties. Indeed, understanding the role (non)computability and computational universality play in natural dynamical systems probably requires significant advances in both real computation and dynamical systems. The role of computational universality—the ability of natural systems to simulate generic computation—in nature is therefore likely to remain one of the most tantalizing open problems in natural philosophy for some time to come.

Bibliographic Notes

The following references include extensive bibliographies for readers interested in computation over the reals; computability of real numbers was first discussed in Turing's seminal paper,[4] which also started the field of computability. There are two main modern visions on computability over the real numbers: computable analysis and the Blum-Shub-Smale (BSS) framework. My presentation here is fully based on the framework of computable analysis, as presented in-depth by Weihrauch.[6] The BSS framework is more closely related to algebraic geometry and presented by Blum et al.[1] I focused on computable analysis, as it appears more appropriate for the study of the computational hardness of natural problems over the reals. The results on the computability and complexity of Julia sets was presented by Braverman and Yampolsky.[2] Computational universality of dynamical systems is discussed in several sources, including Moore[3] and Wolfram,[7] but many basic questions remain open.

Acknowledgments

Work on this article has been supported in part by an Alfred P. Sloan Fellowship, National Science Foundation awards CCF-0832797 and CCF-1149888, and a Turing Centenary Fellowship from the John Templeton Foundation.

References

1. Blum, L., Cucker, F., Shub, M., and Smale, S. *Complexity and Real Computation.* Springer-Verlag, New York, 1998.
2. Braverman, M. and Yampolsky, M. *Computability of Julia Sets.* Springer Verlag, Berlin Heidelberg, 2009.
3. Moore, C. Unpredictability and undecidability in dynamical systems. *Physical Review Letters 64*, 20 (May 1990), 2354–2357.
4. Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 42*, 2 (Nov. 12, 1936), 230–265.
5. Vardi, M. Solving the unsolvable. *Commun. ACM 54*, 7 (July 2011), 5.
6. Weihrauch, K. *Computable Analysis.* Springer-Verlag, Berlin, 2000.
7. Wolfram, S. *A New Kind of Science.* Wolfram Media, Champaign, IL, 2002.

Mark Braverman (mbraverm@cs.princeton.edu) is an assistant professor in the Department of Computer Science at Princeton University, Princeton, NJ.