Contemporary Mathematical Approaches to Computability Theory

Luis Guilherme Mazzali

Undergraduate Student Research Internship

Western University

August 22, 2021

In this paper, I present an introduction to computability theory and adopt contemporary mathematical definitions of computable numbers and functions to prove important theorems in computability theory. I start by outlining the history of computability theory, followed by a brief introduction to Turing Machines. These discussions will hopefully equip the reader with the intuition associated with the concepts of computable numbers and computable functions. I then present the partial recursive functions, which constitute a mathematical framework for the computable functions. This will permit a discussion on computable numbers. Finally, I prove important theorems in computable theory using the techniques and definitions presented in the first three sections. This is intended to provide a motivation for computable analysis – a branch of mathematical analysis based on computability theory that focuses exclusively on computable objects.

The Entscheidungsproblem

In the 1920's, mathematician David Hilbert was interested in finding an effective procedure (an algorithm) through which the validity – truth or falsehood – of any mathematical statement could be determined. That is, he believed that mathematics could be reduced to a finite list of axioms and that it was possible to find a procedure that, within a finite number of steps, could decide whether any given mathematical statement was a logical consequence of the axioms; that is, provable within the axiomatized mathematical language. He named this hypothesis the *Entscheidungsproblem* (decision problem). According to the Stanford Encyclopedia of Philosophy,

> A positive answer to the *Entscheidungsproblem* could be interpreted as showing that it is possible to mechanize the search for proofs in mathematics in the sense of allowing us to algorithmically determine if a formula expressing an open question (e.g. the Riemann Hypothesis) is a logical consequence of a suitably powerful finitely axiomatized theory (e.g., Gödel-Bernays set theory)[1].

The hypothesis suffered its first blow in 1931, when mathematician Kurt Gödel posed his incompleteness theorems. In summary, he concluded that a complete axiomatization of mathematics was impossible; that is, that "there is no reasonable list of axioms from which we can prove exactly all true statements of number theory"[2]. A few years later, mathematicians Alonzo Church and Alan Turing, following Gödel's results, independently developed theories whose methods could be used to show that the Entscheidungsproblem was unsolvable.

[1] Stanford Encyclopedia of Philosophy, "Recursive Functions", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/recursive-functions/.
[2] Stanford Encyclopedia of Philosophy, "Computability and Complexity", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/computability/.

Their methods – the Turing Machine by Turing and λ−calculus by Church –, like many that emerged during their era[3], defined philosophical and mathematically rigorous notions of calculability that encapsulated the full power of the human capacity to solve mathematical problems by hand without the use of ingenuity[4]. Following the "Church-Turing Thesis"[5], such notions are now known as *effective* methods in logic, computer science, and mathematics. The thesis implies that they are equivalent and can be used to define the same set of *effectively calculable*, or *computable*, objects. Formally, a method is so-called effective if:

1. "it is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);
2. it will, if carried out without error, produce the desired result in a finite number of steps;
3. it can (in practice or in principle) be carried out by a human being unaided by any machinery except paper and pencil;
4. it demands no insight, intuition, or ingenuity, on the part of the human being carrying out the method."[6]

Turing used his conceptual device, the Turing Machine, to show that the Entscheidungsproblem was not solvable by an effective method. This means that there is no procedure that, relying solely on the rules of logic and within a finite number of steps, can determine whether any mathematical statement is a logical consequence of a finite list of axioms; thus, Hilbert's conquest to prove the completeness of mathematics was proved impossible, and a new notion of *undecidable* (which cannot be solved by an effective method) problems emerged.

Turing Machines and Undecidability

In this section, the most basic version of the Turing Machine, which is sufficient for our purposes, is outlined. The purpose is for the reader to understand how mechanical calculations done by humans can be replicated by automatic devices; that is, how algorithms can be constructed to solve certain mathematical problems. A key takeaway is that, as mentioned in the

[3] Such as Gödel's Recursive Functions, Kleene's Formal Systems, Markov's Markov Algorithms, and Post's Post Machines. Obtained from Stanford Encyclopedia of Philosophy, "Computability and Complexity", Stanford University, accessed August 5, 2021, https://plato.stanford.edu/entries/computability/.
[4] Before the emergence of modern computers, the word "computer" referred to people, mostly women, who carried out laborious mathematical calculations by hand. Their work, often glossed over in accounts of the advent of the digital era, was essential to the early days of astronomy and spaceflight, for example.
[5] The thesis is based on both Turing's Thesis and Church's Thesis. The former states that "[Turing Machines] can do anything that could be described as "rule of thumb" or "purely mathematical";" the latter, that "A function of positive integers is effectively calculable only if λ- definable (or, equivalently, recursive)." Church proved that the set of functions computable by Turing Machines corresponds exactly to the set of λ- definable functions. Therefore, a method is deemed effective by the Church-Turing Thesis if it can be shown, for example, that it has the full computational power of a universal Turing Machine.
[6] Stanford Encyclopedia of Philosophy, "The Church-Turing Thesis", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/recursive-functions/.

previous section, *any* problem whose solution can be found by an algorithm (an effective procedure) can be solved by an appropriately defined Turing Machine[7].

A basic Turing Machine has three main parts: a finite control, a tape, and an input. The tape is an infinite list of cells, each of which holds a single symbol. The input is a string of symbols $\{X_1, X_2, \ldots, X_n\}$ of finite length that is initially placed on the tape. All other cells of the tape that do not contain an input symbol hold a blank symbol, say "B". Finally, the finite control is placed above the tape and is always in any of a finite set of states $\{q_1, q_2, \ldots, q_m\}$. The finite control is linked to the tape by the tape head, which is initially placed on top of the first input symbol. The tape head is said to be "scanning" the cell on which it is placed, and always scans only one cell at a time.

The Turing Machine operates through "moves" that are determined by the current state of the finite control and the symbol in the cell that the tape head is scanning. In one move, the machine will: change or maintain the current state, write a tape symbol $\in \{X_1, X_2, \ldots, X_n, B\}$ in the cell scanned, and move the tape head left or right. Formally, then, a Turing Machine (TM) M can be described by the 7-tuple:

$$M = (Q, \Sigma, T, \delta, q_0, B, F) \tag{1}$$

where

$$Q = \{q_0, q_1, \ldots, q_m\} \tag{2}$$

is the finite set of states of the finite control,

$$\Sigma = \{X_1, X_2, \ldots, X_n\} \tag{3}$$

is the finite set of input symbols,

$$T = \Sigma \cup \{B\} \tag{4}$$

is the complete set of tape symbols, and

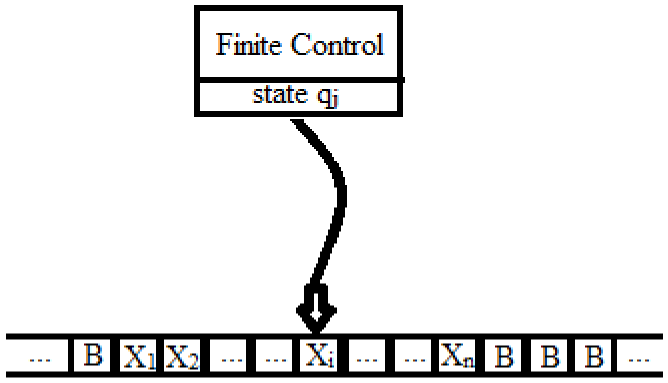
$$\delta(q, X) = (p, Y, D) \tag{5}$$

is the transition function of the current state q and scanned symbol X at a given point in time, which describes a move of M. In the 3-tuple returned by the transition function, p is the new state of the finite control, $Y \in T$ is the tape symbol written in the scanned cell, and D is the direction, left (L) or right (R), in which the tape head moves. Finally, q_0 is the start state in which the finite control is found initially, B is the blank symbol, and $F \subseteq Q$ is the set of accepting states, which cause the machine to halt (accept) upon achieving.[8]

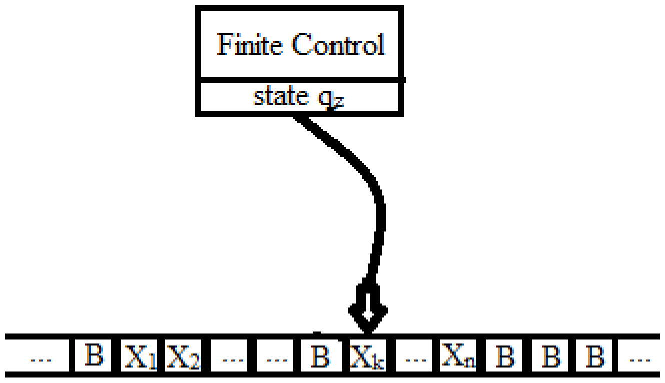
[7] Note, however, that not all decidable problems can be solved by the type of Turing Machine presented here. This model is a deterministic one with a single input tape; variations include, for example, nondeterministic and multi-tape Turing Machines. A more precise statement would be that any decidable problem can be solved by a universal Turing Machine – one that simulates an arbitrary Turing Machine on arbitrary input.

[8] Jeffrey D. Ullman, John E. Hopcroft, Rajeev Motwani, *Introduction to Automata Theory, Languages, and Computation,* 2nd ed. (Boston, MA: Addison-Wesley, 2001), 307-327.

At a given point in time, the machine M can be described graphically in the following way:



Here, the tape head is scanning the cell holding the symbol $X_i \in T$ and the finite control is in state $q_j \in Q$. The next image illustrates a possible move of M:



The move is represented by the transition function $\delta(q_j, X_i) = (q_z, B, R)$. That is, the tape head replaced X_i by B in the cell being scanned, moved one cell to the right, and the finite control changed to state q_z. Note that the state of the finite control and the symbol in a cell need not necessarily change after a move.

After a move, one of three things will happen: the machine will perform another move given the (possibly) new state and new symbol being scanned; the machine will reach an accepting state and halt; or the machine will have no possible moves given the new state and symbol (the transition function is undefined on that 2-tuple) and halt without accepting.

Here is an outline of how addition of positive integers can be performed by this type of TM. Let M_A be the TM that performs positive integer addition. Denote each positive integer by a string of zeroes containing as many zeroes as the value of that integer (e.g., 1 by 0, 2 by 00, 3 by 000, etc.). Denote the addition symbol "+" by a letter, say "a". Let q_a be the accepting state only reachable after replacing a by a blank symbol B.

Let us compute $2 + 3$. The sum $2 + 3$ is represented by $00a000$. Place this string on $M_A's$ input tape and the tape head above the first zero. Replace this zero by a symbol, say "Y", and move the tape head to the right until it finds a blank, then replace that blank by a zero. Then move the tape head leftward until it reaches the first Y, and move one cell to the right, finding a zero. Repeat the previous step. Now, after returning leftward and reaching the first Y, the tape head moves right and finds a. Replace a by a blank symbol, thus reaching q_a, and accept. The tape now contains the string *YYB00000B.* The number of zeroes correspond to the result of the sum: 5.[9]

Note that every move described above can be replicated by a transition function and that for any sum of positive integers, M_A will eventually accept and halt. Moreover, check that the algorithm described above requires no use of ingenuity, but simply the following of a well-defined finite list of rules. Thus, integer addition is clearly effectively calculable, that is, decidable.

Formally, a TM that eventually halts (regardless of whether it accepts) on a given input may be called an *algorithm.* Also, if there exists a TM (algorithm) to solve a given mathematical (determine truth or falsehood[10]) problem, then the problem is called *decidable*. Accordingly, a statement whose truth or falsehood cannot be determined by any TM within a finite number of moves is called *undecidable.*[11]

The Partial Recursive Functions

Turing's abstract device is the most widely used tool to define computability. In fact, all procedures calculable by modern computers are, in principle, also calculable by appropriately defined TMs. The main advantage of Turing Machines is that they are very intuitive: they allow for the easy visualization of computable objects. Therefore, they are often useful for analysis when referring to the computational properties of TMs, as opposed to that of other mathematically rigorous methods, more easily demonstrates certain examples.

However, in general, other methods are preferred for the analysis of computable objects. These attempt to provide a mathematically rigorous foundation for computable functions, and, by extension, computable numbers. The method adopted in this paper is the partial recursive functions[12].

[9] This algorithm was obtained from the website GeeksforGeeks, "*Turing Machine for Addition*", accessed August 6, 2021, https://www.geeksforgeeks.org/turing-machine-addition/.

[10] Note that any mathematical problem can be reframed in terms of a true-or-false question, even if this is not immediately obvious. For example, in the case of integer addition, the value of 2 + 3 can be determined by using M_A to compute it and then comparing the resulting number of zeroes on the tape to each positive integer and asking whether the numbers are the same. The only case in which the answer will be "yes" or "true" is when comparing the number of zeroes to 5. In this sense, M_A can be used to determine truth or falsehood of integer addition.

[11] Jeffrey D. Ullman, John E. Hopcroft, Rajeev Motwani, *Introduction to Automata Theory, Languages, and Computation,* 2nd ed. (Boston, MA: Addison-Wesley, 2001), 307-327.

[12] Also known as the general recursive functions or μ-recursive functions.

As the name suggests, the recursive functions are those functions defined by the repeated application of certain rules. They can be constructed from a finite list of basic functions and composition rules (functionals). This intuitively shows that these functions are calculable by an effective process; that is, they resemble an algorithm. The term *partial*, as opposed to *total*, is used to include functions which are effectively calculable but not necessarily defined on all points of their domains. The Church-Turing Thesis confirms that the set of partial recursive functions correspond exactly to the set of functions computable by TMs; thus, as will become clear, the partial recursive functions can simply be called the computable functions. Then, informally, a function is computable if there is an effective process (e.g., a Turing Machine that eventually halts or a definition in terms of partial recursive functions) that, given input $n \in \mathbb{N}$, returns $f(n)$[13].

The partial recursive functions are an extension of the *primitive* recursive functions. The latter are based on three basic functions:

1. the successor function $succ\colon \mathbb{N} \to \mathbb{N}$ defined by $succ(x) = x + 1$;
2. the zero function $zero^n\colon \mathbb{N}^n \to \{0\}$ defined by $zero^n(x_1, \dots, x_n) = 0$;
3. and the projection function $proj_i^n\colon \mathbb{N}^n \to \mathbb{N}$ defined by $proj_i^n(x_1, \dots, x_i, \dots, x_n) = x_i$.

And two functionals:

1. Composition, defined in the following sense: let $f\colon \mathbb{N}^n \to \mathbb{N}$ and $g_i\colon \mathbb{N}^m \to \mathbb{N}$, $i = 1, \dots, n$. Then the functional $Comp_n^m[f, g_1, \dots, g_n]$ denotes the function

$$f\big(g_1(x_1, \dots, x_m), \dots, g_n(x_1, \dots, x_m)\big)$$

   of type $\mathbb{N}^m \to \mathbb{N}$.[14]

2. And primitive recursion, defined in the following sense: fix a base case function $f\colon \mathbb{N}^n \to \mathbb{N}$ and a recursive case function $g\colon \mathbb{N}^{n+2} \to \mathbb{N}$. Then the function $h = \rho^n(f, g)\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is the primitive recursion with base case function f and recursive function g. The function h is defined by the scheme:

$$h(x_1, \dots, x_n, 0) = f(x_1, \dots, x_n)$$

[13] This is more precisely stated as follows: the claim is true if there is an effective process that given a *representation* of input $n \in \mathbb{N}$, returns a *representation* of $f(n)$. As it is shown later in the paper, there exist computable real numbers which have infinite decimal representations; thus, they cannot be completely described by a computing device (e.g., a TM), if not by means of some alternative finite representation.

[14] Stanford Encyclopedia of Philosophy, "Recursive Functions", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/recursive-functions/.

$$h(x_1, \ldots, x_n, y+1) \ = \ g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y))$$[15]

Primitive recursion allows for the definition of functions whose value on input $y \in \mathbb{N}$ (the recursion variable) is determined by their values on $[0, \ldots, y \ - \ 1] \cap \mathbb{N}$. Since the base case (input y = 0) is completely determined by the function *f*, the value of *h* on any $y \in \mathbb{N}$ can be computed by primitive recursion. The *x* inputs are called parameters and depend on the function specified. For example, let *add* be the function that adds two natural numbers. Then *add* can be defined by *add(x,y)*, where *x* is the only parameter and *y* is the recursion variable allowed to vary.

Now, let us examine how the operations of addition, subtraction, multiplication, and exponentiation can be constructed from the building blocks defined above. Starting with addition, define add: $\mathbb{N}^2 \rightarrow \mathbb{N}$ by $add(x, y) \ = \ x \ + \ y$. As mentioned above, add can be defined through primitive recursion. Define the base case function f: $\mathbb{N}^1 \rightarrow \mathbb{N}$ by $f(x) \ = \ add(x, 0)$. Clearly, we want $add(x, 0) \ = \ x \ + \ 0 \ = \ x$, and so $f(x) \ = \ x$. Using the projection function, it follows that $f \ = \ proj_1^1$. Define the recursive function g: $\mathbb{N}^3 \rightarrow \mathbb{N}$ by $g(x, y, add(x, y)) \ = \ add(x, y \ + \ 1)$. Clearly, we want $add(x, y \ + \ 1) \ = \ x \ + \ y \ + \ 1 \ = \ add(x, y) \ + \ 1$. Using the successor function, $g(x, y, add(x, y)) \ = \ succ(add(x, y))$. But $add(x, y) \ = \ proj_3^3(x, y, add(x, y))$ and so $g \ = \ Comp[succ, proj_3^3]$. Finally, we have that $add \ = \ \rho^1(proj_1^1, Comp[succ, proj_3^3])$, and so addition is primitive recursive.[16]

Next, let us define subtraction recursively. Define sub: $\mathbb{N}^2 \rightarrow \mathbb{N}^+$ by $sub(x, y) = \max(x - y, 0)$. Define the base case f: $\mathbb{N}^1 \rightarrow \mathbb{N}$ by $f(x) = sub(x, 0)$. We want $sub(x, 0) = x - 0 = x$ and so $f(x) = proj_1^1(x)$. Define the recursive function g: $\mathbb{N}^3 \rightarrow \mathbb{N}$ by $g\big(x, y, sub(x, y)\big) = sub(x, y + 1)$. We want $sub(x, y + 1) = x - y - 1 = sub(x, y) - 1 = pred(sub(x, y))$, where $pred = \rho^0(zero^0, proj_1^2)$ is the predecessor function defined by $pred(x) = x - 1$; $pred(0) = 0$. Then $g\big(x, y, sub(x, y)\big) = Comp[pred, proj_3^3]$. Finally, we have that $sub = \rho^1(proj_1^1, Comp[pred, proj_3^3])$, and so subtraction is primitive recursive.[17]

Multiplication can be defined similarly. Define $mult$: $\mathbb{N}^2 \rightarrow \mathbb{N}$ by $mult(x, y) = \ x \ \cdot \ y$. Define the base case f: $\mathbb{N}^1 \rightarrow \mathbb{N}$ by $f(x) = \ mult(x, 0)$. We want $mult(x, 0) = \ x \ \cdot \ 0 \ = \ 0$ and so $f(x) = \ zero^1(x)$. Define the recursive function g: $\mathbb{N}^3 \rightarrow \mathbb{N}$ by $g\big(x, y, mult(x, y)\big) = mult(x, y \ + \ 1)$. We want $mult(x, y \ + \ 1) = \ x \ \cdot \ y \ + \ x \ = \ mult(x, y) + \ x$. It follows that $g\big(x, y, mult(x, y)\big) = \ add(mult(x, y), x)$. But $mult(x, y) = \ proj_3^3\big(x, y, mult(x, y)\big)$ and $x \ = \ proj_1^3\big(x, y, mult(x, y)\big)$; thus, we get $g\big(x, y, mult(x, y)\big) = \ Comp[add, (\ proj_1^3, proj_3^3)]$.

[15] Hackers at Cambridge, January 21, 2018, *Partial Recursive Functions 4: Primitive Recursion* [Video]. YouTube. https://www.youtube.com/watch?v=cjq0X-vfvYY&ab_channel=HackersatCambridge.

[16] Hackers at Cambridge, January 21, 2018, *Partial Recursive Functions 4: Primitive Recursion* [Video]. YouTube. https://www.youtube.com/watch?v=cjq0X-vfvYY&ab_channel=HackersatCambridge.

[17] Hackers at Cambridge, January 21, 2018, *Partial Recursive Functions 4: Primitive Recursion* [Video]. YouTube. https://www.youtube.com/watch?v=cjq0X-vfvYY&ab_channel=HackersatCambridge.

Finally, we have that $mult = \rho^1(zero^1, Comp[add, (proj_1^3, proj_3^3)])$, and so multiplication is also primitive recursive.[18]

Lastly, let us define exponentiation recursively. Define exp: $\mathbb{N}^2 \to \mathbb{N}$ by $\exp(x, y) = x^y$. Define the base case function by $f(x) = \exp(x, 0)$. We want $exp(x, 0) = x^0 = 1$ and so $f(x) = 1 = Comp[succ, zero^1]$. Define the recursive function by $g(x, y, \exp(x, y)) = \exp(x, y + 1)$. We want $\exp(x, y + 1) = x^{y+1} = mult(\exp(x, y), x)$ and so $g(x, y, \exp(x, y)) = Comp[mult, (proj_3^3, proj_1^3)]$. It follows that $\exp = \rho^1(Comp[succ, zero^1], Comp[mult, (proj_3^3, proj_1^3)])$ and so exponentiation is primitive recursive.

Other examples of primitive recursive functions include, but are not limited to, maximum and minimum, order and identity, and functions that return positive integers (constant functions).[19] A common property of these functions is that they are *total*, that is, they are defined on all tuples from their domain. However, Turing-computable functions are not limited to total functions. For example, the integer division function $div(x, y) = q$, where $x = q \cdot y + r$, which returns the integer part of a division of two integers is computable by a TM, but it is clearly not defined on all 2-tuples of positive integers, such as (10,0); thus, there must be recursive functions that are not primitive recursive. To define them, we introduce a third functional: *minimisation*. The set of functions definable by means of the three basic functions and the three functionals is called the *partial* recursive functions.

The minimisation operator, or μ-operator, applied to a primitive recursive function f:$\mathbb{N}^k \to \mathbb{N}$, returns the first (in the order of the natural numbers) $y \in \mathbb{N}$ such that $f(x_1, \ldots, x_k, y) = 0$. This property can be manipulated to simulate the search for an input that satisfies some relevant condition for a recursive function *f*. For example, fix the division function defined previously. The *q* that satisfies $div(x, y) = q$ is the greatest *q* such that $q \cdot y \leq x$. We know that $(q + 1) \cdot y = q \cdot y + y > q \cdot y + r = x$, and so $(q + 1) \cdot y > x$. The problem is now one of finding the *lowest* *q* such that $(q + 1) \cdot y > x$. Integer division is therefore a minimisation problem. In this way, by a process involving primitive recursion and minimisation, which we omit, integer division can be defined as:

$$div = \mu^2(Comp[LessThanEqual, (Comp[mult, (Comp[succ, proj_3^3], proj_2^3), proj_1^3))$$

where LessThanEqual is a primitive recursive function that takes two arguments and outputs 1 if the first argument is less than or equal to the second, and 0 otherwise. It follows that integer division is a partial recursive function.[20]

[18] Hackers at Cambridge, January 21, 2018, *Partial Recursive Functions 4: Primitive Recursion* [Video]. YouTube. https://www.youtube.com/watch?v=cjq0X-vfvYY&ab_channel=HackersatCambridge.

[19] Stanford Encyclopedia of Philosophy, "Recursive Functions", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/recursive-functions/.

[20] This discussion on integer division was completely obtained from Hackers at Cambridge, February 17, 2018, *Partial Recursive Functions 5: Minimisation* [Video]. YouTube. https://www.youtube.com/watch?v=bFkU-qV2Ioo&ab_channel=HackersatCambridge.

It might seem odd that the partial recursive functions, or simply the *computable functions*, are exclusively of type $\mathbb{N} \to \mathbb{N}$. However, as will be shown in the next section, every computable object can be described in terms of a list of positive integers, either directly or indirectly by means of some bijection. For example, the decidable sets and the computable *real* functions of type $\mathbb{R} \to \mathbb{R}$, which are defined below, are based on the computable functions. This is also true for the computable real numbers – the subject of the next section.

Definition 1: A set $A \subseteq \mathbb{N}$ is *decidable* (or computable) if there exists a computable function *f* such that for any $x \in \mathbb{N}$, $f(x) = 1$ if $x \in A$ or $f(x) = 0$ if $x \notin A$. That is, the characteristic function of *A* is computable.

Definition 2: Let $f: \mathbb{R} \to \mathbb{R}$ be a function defined on a closed interval $[a, b]$. Then *f* is called a *computable real function* if there exists a computable sequence[21] of rational polygons[22] (pg_n) which converges to *f* in the sense that $|pg_n(x) - f(x)| < 2^{-n}$ holds for all $n \in \mathbb{N}$ and $x \in [a, b]$.[23]

The Computable Numbers

Like computable functions, computable numbers can be defined in a variety of ways. Trivially, a real number *a* is computable if there exists a TM *M* and a finite string of symbols *w* such that *M*, when running on input *w*, eventually halts and outputs *a*. That is, *a* can be computed by a finite, terminating algorithm. This already implies that numbers that can be defined by means of the partial recursive functions are computable. If the latter is true for *a*, then *a* can be said to be *recursively definable.*

In this section, I first prove an important theorem about the computable numbers that shows that most real numbers are in fact non-computable. I then proceed to show how the natural numbers and the integers can be defined recursively, followed by three general and equivalent definitions of computability for the real numbers. I finish by showing that the rational numbers are computable.

Recall that a set *H* is countable if: (1) it is finite; or (2) it is countably infinite, that is, *H* is infinite and there exists a bijective map $f: \mathbb{N}^+ \to H$. Then:

Theorem 1: If Σ is a finite alphabet (a set containing finitely many symbols), then Σ^*, the set of finite strings that can be written with the symbols from Σ, is countable.

Proof:

[21] A computable sequence is a sequence whose elements can be defined recursively; that is, if (a_n) is a computable sequence, then for all *n*, a_n can be described in terms of its index *n* and some natural number *m*.
[22] A rational polygon is a piecewise linear function which connects a finite set of rational turning points on a closed interval. That is, it is a function whose graph is a polygon with rational vertices. Obtained from Bauer, M. S., & Zheng, X. (2010). On the weak computability of continuous real functions. *arXiv preprint arXiv:1006.0394*.
[23] This definition was given by Bauer, M. S., and Zheng, X. in Bauer, M. S., & Zheng, X. (2010). On the weak computability of continuous real functions. *arXiv preprint arXiv:1006.0394*.

Let Σ be a finite alphabet. For each $n \in \mathbb{N}$, let $\Sigma_n \subseteq \Sigma^*$ be the set of strings of length n. Clearly, $|\Sigma_n| = |\Sigma|^n$ and since Σ is finite, Σ_n is also finite for each n, and thus countable.

It is trivial that $\cup_{n\in\mathbb{N}} \Sigma_n = \Sigma^*$. Also, $\{\Sigma_n \mid n \in \mathbb{N}\}$ is countable since $f: \mathbb{N}^+ \to \{\Sigma_n \mid n \in \mathbb{N}\}$ is bijective. It follows that Σ^* is a union of countably many countable sets, and thus countable[24].

■

Corollary 1: The computable numbers form a countable subset of the real numbers.

Proof:

Define a "Turing program" to be the collection of all the moves performed by a TM after some finite input is placed on its tape. Then, any Turing program that describes a TM *M* which eventually halts on some input *w* can be described as a finite string of symbols. This follows from a TM's nature as a finite algorithm: it has finitely many states, tape symbols, and actions it can perform. To see this, choose a TM *M* and an input string *w* such that *M* eventually halts when *w* is initially placed on its tape. Assign a unique symbol to each of *M*'s states, tape symbols and actions (move left or right, and rewrite symbol), and let β be the set containing exactly those symbols. Each move of *M* can thus be described as a finite string of symbols from β. Since *M* eventually halts, it only performs finitely many moves; thus, the program *M(w)*, which lists all the moves of *M* in order, is a finite string of symbols.

Next, assign a unique finite binary string to each symbol in β, and let $\Sigma = \{0,1\}$. Replace each symbol in *M(w)* by its corresponding binary string. Then $M(w) \in \Sigma^*$. But *M* and *w* were arbitrary and so the collection of all Turing programs that describe a TM that eventually halts, $\{M(w) | M\ eventually\ halts\ on\ some\ input\ w\}$, is contained in Σ^*. But Σ^* is countable by Theorem 1, and so $\{M(w) | M\ eventually\ halts\ on\ some\ input\ w\}$ is also countable.

Therefore, there are at most countably many Turing Machines that eventually halt. It follows that there are at most countably many real numbers that can be generated by a Turing program; that is, the computable numbers form a countable subset of the real numbers.

■

Recall that $\mathbb{R}$ is uncountable by Cantor's diagonal argument[25]; thus, it is a direct consequence of Corollary 1 that there exist real numbers that are not computable, and the set containing all non-computable real numbers is uncountable.

But what are the computable numbers? Informally, they are the numbers that can be represented in a finite way; either directly, such as the integers and the rational numbers, whose representations in the decimal numeral system are already finite strings of symbols (e.g., -1, 2,

[24] This last statement is proved in Miklós Laczkovich, Vera T. Sós, *Real Analysis: Foundations and Functions of One Variable,* (Berlin, Germany: Springer, 2014), 99.

[25] Miklós Laczkovich, Vera T. Sós, *Real Analysis: Foundations and Functions of One Variable,* (Berlin, Germany: Springer, 2014), 99-100.

5/4), or indirectly, such as by means of nested intervals which contain that number. In fact, any number whose complete description can be thought of is computable (e.g., $\sqrt{2}$ and π), which implies that prior to studying computability theory, one is very unlikely to have faced any non-computable number at all. As it turns out, we have not discovered many non-computable numbers. A few examples of this mysterious class of real numbers include Chaitin's constant[26], the limit of Specker sequences[27], and the solution of the Busy Beaver Problem BB(n)[28].

I now give a proof for the computability of natural numbers. This is easily done using the partial recursive functions.

Theorem 2: The natural numbers are computable.

Proof:

The proof involves showing that natural computers can be computed by a finite, terminating algorithm. Choose $n \in \mathbb{N}$ and define a function $f: \mathbb{N} \to \mathbb{N}$ by $f(n) = n$. Using the methods of the previous section, it can be easily shown that $f = \rho^0(zero^1, Comp[succ, proj_1^2])$; thus, *n* can be defined recursively. But *n* was arbitrary and so the natural numbers are computable.

■

Corollary 2: The negative integers are computable.

Proof:

This corollary is not given a formal proof, but an intuitive one: if the natural numbers are computable, then, for all $n \in \mathbb{N}$, there exists a Turing Machine *M* that on some finite input *w* writes *n* on its output tape after finitely many steps. But *n* is a finite string of symbols, and so -*n* is also a finite string of symbols. Modify *M* to write the symbol "-" in front of the first digit of *n* after computing *n*. Then -*n* is computable.

■

I now present three equivalent definitions that can be used to determine whether an arbitrary real number is computable. The proof of their equivalence is omitted.

---

[26] That is, the probability that a randomly constructed program will halt. Obtained from WolframMathWorld, "Chaitin's Constant", WolframMathWorld, accessed August 6, 2021. https://mathworld.wolfram.com/ChaitinsConstant.html.

[27] These are increasing and bounded computable sequences of rational numbers. A proof that the limits of Specker sequences are non-computable is given in Klaus Weihrauch, *Computable Analysis: An Introduction*, (Berlin, Germany: Springer, 2000), 5.

[28] This problem was proposed by Tibor Radó in 1962 and it consists in "finding the largest finite number of 1s that can be produced on blank tape using a Turing Machine with n states." Obtained from Jorgen Veisdal, "Uncomputable Numbers", Jorgen Veisdal, accessed August 7, 2021. https://jorgenveisdal.medium.com/uncomputable-numbers-ee528830d295.

Definition 3(a): A real number *a* is computable if there exists some computable function $f\colon \mathbb{N} \to \mathbb{N}$ such that for any $n \in \mathbb{N}$, *f* produces a natural number *f(n)* such that:

$$\frac{f(n)-1}{n} \leq a \leq \frac{f(n)+1}{n}$$

The function *f* is said to approximate *a* in this way.[29]

This definition is instrumental in showing that *a* is computable if it is possible to get an arbitrarily precise approximation for *a* without reference to itself. This intuitively defines the computable numbers: *a* is computable if it can be computed (or approximated within arbitrary precision) from a list of positive integers by means of an effective method. A natural consequence of this is that through an effective method, a *finite* portion (approximation) of *a* can be computed from a *finite* list of positive integers. Moreover, note that for each natural number *n*, the above definition gives two rational numbers, one greater than *a* and one smaller than *a*. In other words, *a* is computable if it has arbitrarily tight lower and upper rational bounds[30] that can be computed by an effective method. This is formalized in the following definitions:

Definition 3(b): A real number *a* is computable if there exists a computable real function $f\colon \mathbb{N} \to \mathbb{Q}$ such that for all rational numbers $\varepsilon > 0$, $|f(n) - a| \leq \varepsilon$ for some $n \in \mathbb{N}$. [31]

This definition can be reformulated in terms of nested intervals, yielding a third equivalent definition of computable numbers that is very useful for analysis:

Definition 3(c): Choose $a \in \mathbb{R}$ such that $\{a\} = \cap_{n\in\mathbb{N}} I_n$ where $(I_0, I_1, \dots)$ is a sequence of *closed* intervals with rational endpoints and $I_{n+1} \subseteq I_n$ for all $n \in \mathbb{N}$. Then $(I_0, I_1, \dots)$ is a *name* for *a* and *a* is computable if and only if for each $n \in \mathbb{N}$, the endpoints of I_n are completely determined by *n*; that is, the endpoints of I_n are computable real functions of type $\mathbb{N} \to \mathbb{Q}$.[32]

The fact that the rational numbers are computable follows directly from Definition 3(c):

Theorem 4: The rational numbers are computable.

Proof:

Choose $q \in \mathbb{Q}$ and set $I_n = [q - 2^{-n}, q + 2^{-n}]$. Then $(I_0, I_1, \dots)$ is a sequence of nested intervals and $\{q\} = \cap_{n\in\mathbb{N}} I_n$; thus, q is computable by Definition 3(c).[33]

[29] Wikipedia, "Computable number", Wikipedia, accessed July 1, 2021, https://en.wikipedia.org/wiki/Computable_number

[30] This statement was obtained from Klaus Weihrauch, *Computable Analysis: An Introduction*, (Berlin, Germany: Springer, 2000), 86.

[31] Wikipedia, "Computable number", Wikipedia, accessed July 29, 2021, https://en.wikipedia.org/wiki/Computable_number

[32] This is the same definition given in Klaus Weihrauch, *Computable Analysis: An Introduction*, (Berlin, Germany: Springer, 2000), 4.

[33] This proof was obtained directly from Klaus Weihrauch, *Computable Analysis: An Introduction*, (Berlin, Germany: Springer, 2000), 4.

■

Properties of Computable Functions and Computable Numbers:

I now turn to important theorems about the computable functions and computable numbers that follow from the theorems and definitions presented in the last three sections. Some of them are fundamental to computable analysis, such that the computable real functions are continuous and that the computable numbers form a field.

However, computable analysis is scarcely practiced using the methods presented in this paper. In fact, mathematicians working in this field make use of various representations of real numbers[34] and topology, which have not been explored here.

Theorem 5: If $c \in \mathbb{R}$ is a computable number, then $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = c$ is a computable real function.

Proof:

Let $c \in \mathbb{R}$ be a computable number and define $f: \mathbb{R} \to \mathbb{R}$ by $f(x) = c$. Choose $n \in \mathbb{N}$ and define the rational polygon pg_n piecewise in the following way: for each $x \in \mathbb{R}$, choose some $\delta > 0$ and two intervals with rational endpoints, $[r_1, r_2] \subseteq (x - \delta, x + \delta)$ and $[r_3, r_4] \subseteq (x - \delta, x + \delta)$, such that $r_2 > r_4 > r_1 > r_3$. Define the following functions:

g_1 by $g_1(x) = c + 2^{-n}$ for all $x \in [r_1, r_2]$;

g_2 by $g_2(x) = c - 2^{-n}$ for all $x \in [r_3, r_4]$;

g_3 to be the line connecting r_1 and r_2; and

g_4 to be the line connecting r_3 and r_4.

Let $pg_n = \cup_{i=1}^{4} g_i$. Construct a sequence (pg_n) in this way. Then $|pg_n(x) - c| = |pg_n(x) - f(x)| < 2^{-n}$ for all $n \in \mathbb{N}$ and $x \in \mathbb{R}$, and the theorem is proved.

■

Recall that a function *f* is continuous if it is continuous at every point on which it is defined. A function *f* is continuous at a point a if for all $\varepsilon > 0$ there exists a $\delta > 0$ such that if $|x - a| < \delta$ then $|f(x) - f(a)| < \varepsilon$. Informally, this means that if *x* is arbitrarily close to *a*, then *f(x)* is arbitrarily close to *f(a)*. Then:

Theorem 6: Every computable real function is continuous.

[34] One of such representations are nested sequences of intervals with rational endpoints, as defined in the previous section. Other equally useful representations make use of Dedekind cuts, Cauchy sequences, and b-adic expansions. More in Chen, Q., Su, K., & Zheng, X. (2007). Primitive recursiveness of real numbers under different representations. *Electronic Notes in Theoretical Computer Science*, *167*, 303-324.

Proof:

Let $f\colon \mathbb{R} \to \mathbb{R}$ be a computable real function defined on some closed interval $[a, b]$. Fix a sequence of rational polygons (pg_n) such that $pg_n \to f$ in the sense of Definition 2. Choose $\varepsilon > 0$ and $c \in [a, b]$. Choose a $n \in \mathbb{N}$ such that pg_n has at least two vertices in $(f(c) - \varepsilon, f(c) + \varepsilon)$, say $pg_n(r_1)$ and $pg_n(r_2)$, such that $r_2 > c > r_1$ and $(pg_n(r_2) - 2^{-n}, pg_n(r_1) + 2^{-n}) \subseteq (f(c) - \varepsilon, f(c) + \varepsilon)$.

Let $g\colon [r_2, r_1] \to [pg_n(r_2), pg_n(r_1)]$ be the line that connects $pg_n(r_2)$ and $pg_n(r_1)$. Then, for any $x \in dom(g) = [r_2, r_1]$, we have $f(x) \in (pg_n(r_2) - 2^{-n}, pg_n(r_1) + 2^{-n}) \subseteq (f(c) - \varepsilon, f(c) + \varepsilon)$. Pick a rational number $\delta > 0$ such that $(c - \delta, c + \delta) \subseteq [r_2, r_1]$. We conclude that if $|x - c| < \delta$, then $|f(x) - f(c)| < \varepsilon$, and the theorem is proved.

■

The following theorems are presented without proof[35]:

Theorem 7: The function $f\colon \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^i$ is a computable real function for all $i \in \mathbb{N}$.

Theorem 8: The function $f\colon \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = x + y$ is a computable real function.

Theorem 9: The function $f\colon \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = x \cdot y$ is a computable real function.

Recall that a number field is a subset of $\mathbb{R}$ closed under addition and multiplication[36] and which conforms with the following additive and multiplicative rules: commutativity, associativity, additive identity, additive inverse, multiplicative identity, and multiplicative inverse. Let F_c be the set of computable numbers. Then:

Theorem 10: The computable numbers form a field.

Proof:

1. Closure under addition and multiplication:

Choose computable functions *f* and *g* that satisfy Definition 3(a) for *x* and *y*, respectively. It follows that

$$\frac{f(n) - 1}{n} + \frac{g(n) - 1}{n} \leq x + y \leq \frac{f(n) + 1}{n} + \frac{g(n) + 1}{n} \tag{6}$$

that is,

$$\frac{\big(f(n) + g(n)\big) - 2}{n} \leq x + y \leq \frac{\big(f(n) + g(n)\big) + 2}{n} \tag{7}$$

[35] See Klaus Weihrauch, *Computable Analysis: An Introduction*, (Berlin, Germany: Springer, 2000), 109-111 for proofs of these theorems.

[36] This means that if F is a number field and $a, b \in F$, then $a + b \in F$ and $a \cdot b \in F$.

but $f(n), g(n) \in \mathbb{N}$ and so the function h defined by $h(n) = add(f(n), g(n))$ is a computable function of type $\mathbb{N} \to \mathbb{N}$. Thus,

$$\frac{h(n) - 2}{n} \leq x + y \leq \frac{h(n) + 2}{n} \tag{8}$$

Note that $\frac{1}{n} \to 0$ and $\frac{2}{n} \to 0$ as $n \to \infty$; therefore, inequality (8) gets very close to

$$\frac{h(n) - 1}{n} \leq x + y \leq \frac{h(n) + 1}{n} \tag{9}$$

as $n \to \infty$ – as the approximation gets more precise; thus, $x + y \in F_c$ by Definition 3(a), and we conclude that F_c is closed under addition.

Now, choose sequences of nested intervals $(I_0, I_1, \dots)$, $(J_0, J_1, \dots)$ that satisfy Definition 3(c) for x and y, respectively. We intend to show that $x \cdot y \in F_c$ according to Definition 3(c). Choose $n \in \mathbb{N}$ and fix $I_n = [f_1(n), g_1(n)]$ and $J_n = [f_2(n), g_2(n)]$, where f_1, g_1, f_2, g_2 are computable real functions of type $\mathbb{N} \to \mathbb{Q}$. Clearly,

$$f_1(n) \cdot f_2(n) \leq x \cdot y \leq g_1(n) \cdot g_2(n) \tag{10}$$

And so $(I_0 \cdot J_0, I_1 \cdot J_1, \dots)$ is a sequence of nested intervals such that $\{x \cdot y\} = \cap_{n \in \mathbb{N}} I_n \cdot J_n$, where $I_n \cdot J_n = [f_1(n) \cdot f_2(n), g_1(n) \cdot g_2(n)]$.

Each of f_1, g_1, f_2, g_2 can be written as a ratio of two computable functions of type $\mathbb{N} \to \mathbb{Z}$. Let $f_1(n) = \frac{u(n)}{v(n)}$ and $f_2(n) = \frac{i(n)}{l(n)}$. Then $f_1(n) \cdot f_2(n) = \frac{u(n) \cdot i(n)}{v(n) \cdot l(n)}$, where $u(n) \cdot i(n)$ and $v(n) \cdot l(n)$ are of type $\mathbb{N} \to \mathbb{Z}$ since $\mathbb{Z}$ is a field. Clearly, then, $f_1 \cdot f_2$ is a computable function of type $\mathbb{N} \to \mathbb{Q}$. An analogous argument can be given for $g_1 \cdot g_2$. We conclude that $x \cdot y \in F_c$ by Definition 3(c) and F_c is closed under multiplication.

2. Commutativity and associativity of addition and multiplication:

Note that $F_c \subseteq \mathbb{R}$ and that $\mathbb{R}$ is a field. Since commutativity and associativity of addition and multiplication hold for the real numbers, they must also hold for the computable numbers.

3. Additive identity:

We know that $zero^1(n) = 0$ for all $n \in \mathbb{N}$; thus, 0 can be defined recursively and so $0 \in F_c$. Let $x \in F_c$. Clearly, $x + 0 = x$ since $x \in \mathbb{R}$. Since x was arbitrary, 0 is the additive identity of F_c.

4. Additive inverse:

Let $x \in F_c$. It follows from $x \in \mathbb{R}$ that if $x + w = 0$ then $w = -x$; thus, it suffices to show that $-x \in F_c$. Choose a computable function $f: \mathbb{N} \to \mathbb{N}$ that satisfies Definition 3(a) for x. Then

$$\frac{f(n) - 1}{n} \leq x \leq \frac{f(n) + 1}{n} \tag{13}$$

And so

$$\frac{-f(n)+1}{n} \geq -x \geq \frac{-f(n)-1}{n} \tag{14}$$

By Corollary 2, $-f(n)$ is computable since $-f(n) \in \mathbb{Z}$. So, there exists a computable function g such that $g(n) = -f(n)$. It follows that

$$\frac{g(n)+1}{n} \geq -x \geq \frac{g(n)-1}{n} \tag{15}$$

And so -*x* is computable by Definition 3(a). Hence F_c possesses an additive inverse.

5. Multiplicative identity:

Clearly $Comp[succ, zero^n] = 1$ for any n-tuple of natural numbers; thus, 1 can be defined recursively and so $1 \in F_c$. For any $x \in F_c$, $x \cdot 1 = x$ since $x \in \mathbb{R}$. It follows that F_c has a multiplicative identity.

6. Multiplicative inverse:

Let $x \in F_c$. We know that if $x \cdot w = 1$ then $w = x^{-1} = \frac{1}{x}$ because $x \in \mathbb{R}$; so, it suffices to show that $x^{-1} \in F_c$.

Choose a sequence of closed intervals with rational endpoints $(I_0, I_1, \ldots)$ such that $\{x\} = \cap_{n\in\mathbb{N}} I_n$ which satisfies Definition 3(c) for *x*. This sequence is used to construct another which satisfies the relevant conditions for x^{-1}. Choose a $n \in \mathbb{N}$ and pick $I_n = [r_1, r_2], I_{n+1} = [r_3, r_4] \in (I_0, I_1, \ldots)$. We know $I_{n+1} \subseteq I_n$ and so

$$r_1 < r_3 < x < r_4 < r_2 \tag{16}$$

Clearly,

$$\frac{1}{r_1} > \frac{1}{r_3} > \frac{1}{x} > \frac{1}{r_4} > \frac{1}{r_2} \tag{17}$$

Let $I_n^* = \left[\frac{1}{r_2}, \frac{1}{r_1}\right]$ and $I_{n+1}^* = \left[\frac{1}{r_4}, \frac{1}{r_3}\right]$. Then both intervals have rational endpoints; $x^{-1} \in I_n^*, I_{n+1}^*$; and $I_{n+1}^* \subseteq I_n^*$. Construct a new sequence $(I_0^*, I_1^*, \ldots)$ in this way. Since *n* was arbitrary in the previous step, we know $\{x^{-1}\} = \cap_{n\in\mathbb{N}} I_n^*$.

Recall that the endpoints of I_n can be described by computable real functions of type $\mathbb{N} \to \mathbb{Q}$, that is, $I_n = [\frac{f(n)}{g(n)}, \frac{h(n)}{e(n)}]$. Clearly, the endpoints of I_n^* can also be described by computable real functions of type $\mathbb{N} \to \mathbb{Q}$ since $I_n^* = \left[\frac{g(n)}{f(n)}, \frac{e(n)}{h(n)}\right]$; then, x^{-1} is computable by Definition 3(c).

■

Theorem 11: If p is a polynomial of one variable with computable real coefficients, then p is a computable real function.

Proof:

Define a polynomial p as $p(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + \cdots + a_m \cdot x^m$, where a_i is computable for all $i = 0,1, \dots, m$. The fact that p is a computable real function follows directly from theorems 8 and 9.

$\blacksquare$

References

Bauer, M. S., & Zheng, X. (2010). On the weak computability of continuous real functions. *arXiv preprint arXiv:1006.0394*.

Chen, Q., Su, K., & Zheng, X. (2007). Primitive recursiveness of real numbers under different representations. *Electronic Notes in Theoretical Computer Science, 167*.

GeeksforGeeks, "*Turing Machine for Addition*", accessed August 6, 2021, https://www.geeksforgeeks.org/turing-machine-addition/.

Hackers at Cambridge, February 17, 2018, *Partial Recursive Functions 5: Minimisation* [Video]. YouTube. https://www.youtube.com/watch?v=bFkU-qV2Ioo&ab_channel=HackersatCambridge.

Hackers at Cambridge, January 21, 2018, *Partial Recursive Functions 4: Primitive Recursion* [Video]. YouTube. https://www.youtube.com/watch?v=cjq0X-vfvYY&ab_channel=HackersatCambridge.

Hopcroft J., Motwani R., Ullman, J. (2001). *Introduction to Automata Theory, Languages, and Computation,* 2nd ed. Addison-Wesley.

Laczkovich, M., Sós, V. (2014). *Real Analysis: Foundations and Functions of One Variable*. Springer.

Stanford Encyclopedia of Philosophy, "Computability and Complexity", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/computability/.

Stanford Encyclopedia of Philosophy, "Formalism in the Philosophy of Mathematics", Stanford University, accessed August 16, 2021, https://plato.stanford.edu/entries/formalism-mathematics/#ForPos.

Stanford Encyclopedia of Philosophy, "Recursive Functions", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/recursive-functions/

Stanford Encyclopedia of Philosophy, "The Church-Turing Thesis", Stanford University, accessed July 1, 2021, https://plato.stanford.edu/entries/recursive-functions/.

Veisdal, J, "Uncomputable Numbers", Jorgen Veisdal, accessed August 7, 2021. https://jorgenveisdal.medium.com/uncomputable-numbers-ee528830d295.

Weihrauch, K. (2000). *Computable Analysis: An Introduction*. Springer.

Wikipedia, "Computable number", Wikipedia, accessed July 1, 2021, https://en.wikipedia.org/wiki/Computable_number.

WolframMathWorld, "Chaitin's Constant", WolframMathWorld, accessed August 6, 2021. https://mathworld.wolfram.com/ChaitinsConstant.html.