A POINT TO SET PRINCIPLE FROM A COMPUTATIONAL COLD START

EMET HIRSCH

Abstract. In this paper we present an exposition on a 2016 result [LL18] connecting geometric definitions of dimension for sets in $\mathbb{R}^d$ to algorithmic definitions of dimension for points within those sets. To do so we assume zero familiarity with the theory of computation, giving an overview of all the preliminaries, to make the result accessible to those with a background purely in geometry.

Contents

1. Introduction 1
2. Turing Machines 2
3. Universal Turing Machines, Undecidability, and Oracles 7
4. K-Complexity and The Real Numbers 12
5. The Point-to-Set Principles 16
6. Appendix 21
References 29

1. Introduction

This is an expository paper of [LL18], which presents a result connecting geometric measure theory and the theory of computation. To make the result accessible to those with no background in the theory of computation, we give an expedited presentation of the necessary material to understand the result and its proof, using a little rigor and describing common intuitions.

- The first section is intended to convey what it means mathematically to "compute" something, and giving an intuitive picture of the decidable problems.
- The second section goes over the general structure of decision problems, going over binary Turing machines, the halting problem, and oracles.
- The third section gives the combinatorial properties of K-complexity necessary for the main result, constructs the appropriate notions for real numbers, and points out the connection between real numbers and oracles.

Then we present the motivating result. It is recommended to skip sections if one is already familiar with their content. Certain proofs, definitions, and examples, which are not useful for conveying intuition, but may be helpful to skeptical readers, are presented in the appendix. The content of the appendix is otherwise superfluous.

Finally, given a set of symbols Σ, we call a finite sequence of elements of Σ a *string*. For example, every English word is a string of the English alphabet symbols. The set of

Date: September 13, 2023.

all strings over a particular set Σ is denoted Σ^*. Note that the string of length zero, or the *null string*, is a member of Σ^*. This definition is a practical necessity to shorten other definitions.

2. Turing Machines

It is natural to ask what it means on a rigorous level to *compute* something, or for a result to be *computable*. On an intuitive level, a set is "computable" if one can find a completely deterministic process that takes as input a potential element, and outputs "yes" if that element is a member, and "no" otherwise. We then might attempt to formalize "deterministic process" by considering how humans perform reasoning, especially when that reasoning is precise and explicit. In general, one needs fixed starting and ending conditions, a place to record information (such as a piece of paper), and a list of rules by which to manipulate existing information.

This exactly corresponds to the formal definition of a Turing machine, which we present following Sipser [Sip13]:

Definition 2.1. A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$ where Q is a finite set of *states*, Σ is a finite set of symbols which are used to encode inputs and do not contain the special symbol $\sqcup$, Γ is a finite set of symbols which contains both Σ and $\sqcup$, δ is a function $Q \times \Gamma \to Q \times \Gamma \times \{left,\ right\}$ called the *transition function*, q_{start} is a special initial state in Q, and q_{accept}, q_{reject} are special end states, distinct from one another, which accept and reject the input respectively, also of course in Q.

To understand this definition, it is important to remember that in defining a Turing machine, we are attempting to make rigorous the idea of "algorithm," and algorithms have steps; a Turing machine specifies the rules of an iterative process. What actually happens in this process?

We start with an infinite (shaped like $\mathbb{N}$) set of discrete "cells." A cell is simply a place to store an element of Γ; it is useful to imagine an infinite line of squares, on which one has inscribed numbers or letters. This line as a whole is referred to as the *tape*. Initially, nearly all of the cells are empty, which we formally denote as having $\sqcup$, the "blank symbol." The only nonblank symbols are consecutive, the first of them starts at position 1, and they are all elements of Σ. These symbols are the input; this is why we mandate that Σ does not contain $\sqcup$, so that there is no ambiguity to the machine about where the input ends.

Given the starting tape, at position 1 is the *head* of the Turing machine, which begins, naturally, in the state q_{start}. Then the iterative process begins: at each step, we apply the transition function δ, taking as Q-input the current state of the machine head, and Γ-input the symbol at its position. Its Q-output becomes the new state of the machine, its Γ-output is written on its current position, replacing the previous symbol, and the output in $\{left$, *right*$\}$ causes the head to move one cell in the respective position. To understand this process, it is useful to imagine the Turing machine head as animate: one can imagine a floating mouth, or claw, reading off a case-based list of instructions and passing across a line of squares remembering only its current state.

If the machine head ever enters the state q_{accept} or q_{reject} when given a particular input, we say it respectively *accepts* or *rejects* the input. This is quite clearly well-defined, since on a given input, each state is determined from the previous one by a function, and is thus uniquely defined. It is worthwhile to consider how this definition corresponds to the familiar process of computation; the transition function and states are the algorithm itself, while the tape is used to record information.

Given a set A, we call a subset of A a *language*. While this term seems strange for a subset of $\mathbb{N}$, we often want to consider subsets of Σ^*, with Σ including mathematical symbols other than numbers, for which the term is natural. We say a Turing machine *recognizes a language* L if the Turing machine accepts an input if and only if that input is a member of the language. If, further, it rejects on every other input (rather than ever looping infinitely), we say it *decides* L.

It is worth noting that deciding a language is preferable to recognizing it for practical purposes; otherwise, knowing a machine has not yet accepted on input x, it is not clear whether it will continue forever, excluding x from L, or will eventually accept x.

If one prefers, a more rigorous formulation of the computation process can be found in the appendix. It is often convenient to think of Turing machines as "solving a problem." For this reason languages are often called *decision problems* which the Turing machine *solves*.

To better understand this definition, let us consider an example. Consider the Turing machine:

(1) $Q =\{q_{start},\, q_1,\, q_2,\, q_{accept},\, q_{reject}\}$
(2) $\Sigma = \{0,\, 1\}$
(3) $\Gamma = \{0,\, 1,\, \sqcup\}$
(4) δ takes
- $(q_{start},\, \sqcup) \to (q_2,\, 1,\, right)$
- $(q_{start},\, 1) \to (q_{accept},\, 0,\, right)$
- $(q_{start},\, 0) \to (q_1,\, 0,\, right)$
- $(q_1,\, \sqcup) \to (q_2,\, 1,\, right)$
- $(q_1,\, 1) \to (q_{accept},\, 0,\, right)$
- $(q_1,\, 0) \to (q_2,\, 0,\, right)$
- $(q_2,\, 0) \to (q_{accept},\, 0,\, right)$
- $(q_2,\, 1) \to (q_2,\, 0,\, right)$
- $(q_2,\, \sqcup) \to (q_2,\, 0,\, right)$

What does this Turing machine actually do? That is to say, what language does it recognize?

Given an input starting with the digit 1, it accepts. On an input beginning with 0, the Turing machine head moves to the next portion of the tape, at which point it accepts if there is a 1 and moves right again if there is a 0. At this point, it continues to move right, staying in the same state, when it reads a 1, and accepts if it finds a 0. The way we defined Turing machines implies that if a Turing machine head reads a blank cell that it did not itself write, and moves to the right, it will read another blank cell. This means that this Turing machine, since $(q_2,\, \sqcup)$ goes to (q_2), will loop infinitely, and therefore never accept, if it reads a blank cell.

So this Turing machine recognizes the language "the set of binary strings which have a 1 in their first two digits, or a 0 thereafter," but it does not decide this language, or any language. A useful exercise is to construct a Turing machine that recognizes the same language as this one, but also decides it.

It is worth noting that the Turing machine we constructed never moves left on the tape, only right. It can be proven that Turing machines restricted in this way compute a much more restricted class of languages than those computed by Turing machines in general. Intuitively, this is because such a Turing machine never goes back and reads the information it wrote, so its "memory" is limited to that of its states.

This machine, while quite simple, and uninteresting in the aforementioned sense, is already tedious to describe. For this reason, Turing machines are usually described in the form of *psuedocode*, that is, a list of unambiguous instructions written in natural language. We present and analyze one more Turing machine as a formal object in the appendix for the purpose of showing off algorithm construction techniques persuasive of the power and generality of Turing machines. If one is already familiar with this, or convinced of it for less mathematical reasons[1], it is reasonable to skip this construction. Frankly, reasoning with the explicit definition of Turing machines is unpleasant.

Specifically, in the appendix we construct a Turing machine that, given an element of Σ^* where $\Sigma = \{0, 1, \circ\}$, accepts it if and only if it is of the form "$\circ$ X $\circ$ Y $\circ$ Z" where $Y + Z = X$, all of which are binary numbers, and rejects otherwise.

This tedious construction give smotivation and justification for a variety of capacities of Turing machines, of which we must become convinced to be able to rigorously read psuedocode.

Turing machines can *remember arbitrarily large finite quantities*. Despite only have a finite number of states, Turing machines can assign special symbols to the tape, which it then reacts to differently later, allowing it perform a particular process an arbitrarily large but finite number of times.

Turing machines can *copy the behavior of other Turing machines*. The machine we constructed performs its computation in a series of steps: first checking the form of the input, then adding leading zeros, then adding Z to Y, then checking the equality of the modified Y and X. But we constructed δ as to separate each stage of this process from the others, giving them separate states and appropriately cleaning the tape of marks between them. The equality-checking process, with its start and end behavior modified slightly, would have sufficed to decide the following language: "$\circ$ X $\circ$ Y" where X and Y are integers represented in binary and $X = Y$.

Let's call this machine E. If while building another Turing machine T, we needed to check such an equality at some point in the process, we can simply place the entire set of states and transition function of E within the set of states Q and transition function δ of T. When T needs to check equality, we have it sends its current state to the relabelled start state of E, which upon reaching its relabelled accept or reject state, goes to a corresponding state in T which remembers whether the integers are equal, cleans the tape, then continues whatever process required checking equality. This is actually a rigorous proof! The skeptical reader should note that machines constructed so would visibly be composed of connected pieces, each for a specific task. The machine we construct is one such, and it is exactly because of this modular construction that we are able to explain the behavior of its transition function step-by-step.

Turing machines can *use what we know about the relevant problem*. "Addition" and "equality" are arithmetical facts, but the Turing machine we constructed simply manipulates meaningless symbols according to specified rules. How do we know our machine's computation had anything to do with addition? Our machine takes a digit from Z and adds

[1] Such as, for example, the fact that these techniques are analogous to those necessary to perform computations using logic gates, without which electronics would not exist.

it to the corresponding digit of Y, remembering to carry a 1 if necessary. The fact that this suffices to add numbers, while obvious, would still require a proof if we were starting from nothing, and such a proof would require many important facts about addition, such as its associativity and distributivity with multiplication. Thus, the Turing machine we constructed "uses" the properties of addition we know in its computation. When we construct more interesting Turing machines, their recognition of the desired language often requires a proof, and such a proof is often nontrivial. This exactly matches the notion of "algorithm" which we used to originally motivate the idea of Turing machines.[2]

Turing machines can *solve problems other than decision problems*. As we defined it, a Turing machine can only determine whether its input is a member of some set by accepting or rejecting it. But the Turing machine we presented, in the process of determining whether to accept or reject a triplet of integers, takes two of them and obtains their sum. Intuitively, this is harder than merely evaluating whether they are a member of some set: it takes the input in Σ^* to an element of $\mathbb{N}$ (represented as a sequence of binary digits), rather than merely taking the sequence to an element of $\{accept, reject\}$.

To formalize Turing machines solving problems that are harder than decision problems, we might try the following definition:

A Turing machine T *computes a function* f, $f : X \to Y$ with X and $Y \in \Sigma^*$, if T accepts on inputs of the form "$x \circ y$" where $x \in X$ and $y \in Y$ with $y = f(x)$ and rejects on all other inputs.

This definition works partially, but is ill-advised. We could imagine some problem of arithmetic that is difficult to solve, but given a candidate solution to it, it is easy to disprove or verify the correctness of that candidate solution. This definition would say such a problem is easy to solve, because it defines "computation" as the ability to disprove or verify candidate solutions! A better definition is as follows:

Definition 2.2. A Turing machine T is said to *compute a function* f, $f : X \to Y$ with $X \subseteq \Sigma^*$ and $Y \subseteq \Gamma^*$, if when given an input $x \in X$, T eventually accepts the input, and when it does so, has $f(x)$ written at the start of it and is blank everywhere else. Further, the machine rejects on inputs not in X.

In this definition, the machine must come up with the output from the input entirely "on its own."[3] Once it does so, it must clean the tape, write the output at the start of the tape, and accept. Such a Turing machine works exactly like a calculator: it takes in symbolic inputs, transforms them according to specific rules, then writes the correct output in a convenient place to be read. It is worth noting that our original definition of Turing machines deciding a language L can be seen as equivalent to a special case of this definition with $X = \Sigma^*$. Simply let the function f be an indicator function for the set L.

[2] This can seem like a trivial or pedantic point. But when one is attempting to understand a correctness proof of an algorithm in a seemingly unrelated field of mathematics, one is doing something exactly analogous to this.

[3] We probably do not have to "imagine" problems for which these definitions are different, given a reasonable notion of "difficult to solve." It is quite troublesome and time-consuming to find the prime factorization of a number, but it is easy to disprove or verify the correctness of a given factorization. This definitional subtlety touches on the famous and central "$\mathcal{P}$ and $\mathcal{NP}$ problem." Specifically, problems troublesome in the aforementioned way exist iff "$\mathcal{P} \neq (\mathcal{NP} \cap co-\mathcal{NP})$." This would be an even stronger result than a resolution to the famous problem.

The Turing machine we presented earlier can easily be adapted to a machine that computes addition, rather than merely deciding a related language. The notion of Turing machines computing functions rather than merely recognizing languages allows us to see more clearly the capabilities of Turing machines. For example, imagine we wanted to construct a Turing machine that given "$\circ\ X \circ Y$" with X and Y numbers in binary digits, computes the product of X and Y in binary. We could use the following machine:

(1) Check that the input is in the appropriate form. If it is not, reject. Otherwise, place a special mark $\bullet$ after the end of Y, and a 0 after it.
(2) Go to Y. If Y consists only of 0s, erase everything to the left of the $\bullet$ mark, then copy everything to the right of it to the start of the tape, delete the mark, then accept the input. Otherwise, perform the following process:
(3) Decrement Y by one. Then go to the first blank space to the right and place a different special mark $\diamond$. Then copy X to after $\diamond$.
(4) Go to the $\bullet$, then behave like the Turing machine which computes the function taking "$\bullet\ X \diamond Y$" to "$\bullet\ Z$" where Z is the sum of X and Y, all binary digits. Then go to step (2).

This machine repeatedly add X to itself until it has done it Y many times, which is to say it multiplies X and Y. Then it moves the resulting number to the start of the tape, cleans the tape, and accepts. Using Turing machines' ability to perform computations, to remember arbitrarily large values, to embed other machines' processes, and to decrement numbers, we have constructed a recipe for a Turing machine to perform a task "x many times" with x an arbitrarily large number.

How does one intuitively characterize the capacities of Turing machines? Recall we defined Turing machines to formalize the notion of "algorithm." Everything a Turing machine does follows directly from its input, set of states, and transition function. Given infinite time and an infinite chalkboard, a human being could compute anything a Turing machine could compute simply by drawing the start of the tape, writing the input on it, and repeatedly applying the transition function, adding new blank symbols to the tape if they ever become necessary. Thus, asymptotically, humans can compute everything Turing machines can compute. This is a natural upper bound to their abilities. What about a lower bound?

Given a finite set of symbols Σ, we can assign an ordering to the countably infinite set Σ^* as follows. First, to Σ an arbitrary ordering. Then $x < y$ if x has fewer symbols than y. If they have the same number of symbols, either they are the same element or they differ in some position. Considering the first position where they differ, $x < y$ if the symbol in that position in $x <$ the one in y. This is known as the *lexicographical ordering*, because it is exactly the same ordering that a household dictionary uses, and it is easy for an algorithm to read for the exact same reason that it is easy to locate a desired section of a household dictionary. Given an element of Σ^*, one can directly compute its position in the lexicographical ordering, and vice versa. This means that, for any property of a sequence of symbols that can be decided by a Turing machine, that is to say any language L decided by some Turing machine, one can construct another Turing machine that:

(1) Writes down the number 1 on the tape, surrounded by special marks.
(2) Reads the number surrounded by special marks, writes down the corresponding sequence of symbols in Σ according to the lexicographical ordering, then checks if it satisfies the property. If it does, it accepts. If it doesn't, it cleans the tape, increments the number by 1, and restarts this step.

This Turing machine accepts if and only if there is some sequence of symbols that satisfies the property, and goes on infinitely otherwise. This is a complete formal proof that given any decidable language L, there is a Turing machine that, if the language is nonempty, computes its lexicographically first member, and otherwise goes on forever.

One should note that for any mathematical claim, given a sequence of mathematical symbols, evaluating whether it is a proof of the claim is a decidable problem. One simply needs to check that each line follows from the previous lines according to the recursively defined rules, and that the input ends in the claim. Thus by the previous result, a Turing machine can prove any mathematical result a human can![4]

For this reason, many believe that Turing machines and human cognition have the same level of generality in doing mathematics. This is known as the *Church-Turing thesis*. This is a philosophical position, not a mathematical assertion, but understanding it is useful for developing intuition regarding Turing machines, most obviously because it justifies us in asking, "Could I do this and be guaranteed to eventually finish, if I had infinite time?" as a way to tell at a first glance whether a Turing machine decides a particular problem.

3. Universal Turing Machines, Undecidability, and Oracles

In the previous section, we made heavy use of marked symbols to record information. It turns out, surprisingly, to be the case that for every function computable by some Turing machine, a basically equivalent function can be computed by a Turing machine using only binary digits and the blank symbol.

First, let us make clear the notion of "equivalent." Given a Turing machine T, it has a set of input symbols Σ and a set of symbols for processing and outputs, Γ, both of which are finite with $\Sigma \subseteq \Gamma$. Let us count the elements of Γ, excluding $\sqcup$, assigning them numbers from 1 to $|\Gamma|$. Now, given a number for each element, consider the function γ which takes each element to that many 0s, followed by a 1. For example, given a Turing machine with $\Sigma = \{0, 1, \circ\}$ and $\Gamma = \{0, 1, \circ, 2, \dot{0}, \dot{1}, \sqcup\}$, we could let γ take $0 \to 01$, $1 \to 001$, $2 \to 0001$, $\dot{0} \to 00001$, $\dot{1} \to 000001$, and $\circ \to 0000001$.

We can apply γ to elements of Γ^* simply by applying it to each symbol in order. Taking $x \in \Gamma^*$, note that $\gamma(x)$ is longer than x by at most a constant factor equal to $|\Gamma| + 1$, the length of the longest sequence of binary digits we created, and is easily converted back to its original form. This means that if we can decide a language or compute a function in the image of Γ under γ, we can also do it in the original form. The two computational problems convey the same information about the underlying objects we care about; in effect, we have simply renamed some symbols.

Now we want to see that for any language L decidable by a Turing machine, its image $\gamma(L)$ (that is the subset of binary strings $\gamma(x)$ for some $x \in L$), in the set of sequences of binary digits, is decidable by a Turing machine for which $\Gamma = \{0, 1, \sqcup\}$. Note that there are many sequences of binary digits that do not correspond to $\gamma(x)$ for any x, and our new machine necessarily must reject on them.

[4] This is non-rigorous. We are omitting some relevant and deep proof-theoretic technicalities related to the meaning of "verification" which are beyond scope and would only serve to confuse.

We prove this statement in the appendix. This is done by, for a given Turing machine, constructing another machine which breaks the tape into ”blocks” of consecutive 0s and 1s, and uses the blocks as single characters for the emulation of the original machine.[5]

Given this result, it is clear that Turing machines are highly redundant. Not only are there infinitely many distinct Turing machines which compute the same function (this can be trivially seen by adding states that are impossible to reach), there are infinitely many ”seemingly distinct” Turing machines which compute functions which are basically identical. When we study Turing machines, we rarely care about their formal specifications, instead only considering what they compute and how. Thus we have learned that, speaking non-rigorously, the set of Turing machines is highly self-similar: we can take the set of all Turing machines and computably, injectively map it to the set of all binary Turing machines, preserving the structure of their inputs and outputs.

This is already quite surprising. Can it be strengthened? For example, can I embed every Turing machine, into only *one* Turing machine? It is worth giving a bit of time and effort to attempt to answer this question for oneself, even before one hears a formal phrasing of it.

More formally: is there some Turing machine T taking inputs of the form ”$X \circ Y$” such that for every Turing machine T' with symbols in Γ, there is some x such that T on input ”$x \circ y$” accepts, rejects, or continues forever if and only if T' on input y accepts, rejects, or continues forever respectively? Further, to generalize this to computing functions, we mandate that if T accepts or rejects on $x \circ y$, its tape must be identical to the tape when T' accepts or rejects. Again, it is worth attempting to resolve this for oneself. *(Hint.)*[6]

Consider the set of Turing machines taking inputs in $\{0, 1\}$ and with $\Gamma = \{0, 1, \sqcup\}$. We just proved that these are ”just as strong” as the entire set of all Turing machines. By the definition of Turing machine, all of these are of the form $(Q, \{0, 1\}, \{0, 1, \sqcup\}, \delta, q_{start}, q_{accept}, q_{reject})$. Q can be written in set notation, with all its elements written in the form q_n for $n \in \mathbb{N}$, written in binary. δ can be specified using the formal definition of a function as a set of ordered pairs, thus using the parentheses symbols, the comma, and the symbols used to represent Q.

Consider a Turing machine T with $\Sigma = \{q, \dot{q}, 0, 1, \dot{0}, \dot{1}, \circ,$ open bracket, close bracket, comma, open parenthesis, close parenthesis$\}$ and $\Gamma = \Sigma \cup \{\sqcup, h\}$.

This is going to be our solution to the problem we posed. Let us say that this machine first reads the input to make sure it is of the form ”$T \circ X$” where T is a Turing machine specified in the previous notation and symbols and X is a binary integer or the empty string, which is the sequence with no symbols. If it isn't, it rejects. If it is, it writes down h followed by the start state of T after the $\circ$ but before the input, moving the input to keep it intact. It also places a $\circ$ between them to differentiate the index of the state from the input, which is necessary because they both consist of binary digits. Then it performs the following process:

(1) Look at the index of the state following the symbol h, and at the symbol on the tape following the $\circ$ after that index, then return to the beginning of the tape. If it matches the accept or reject state, clean T, the $\circ$, and the h and the state written down next to it from the tape, then move the remaining nonblank symbols to the

[5] One might also be inclined to take the theorem statement as self-evident, since in our daily lives we frequently interact with machines that encode a variety of stored types of data to perform different processes using only binary. Specifically, of course, digital electronics.

[6] We have already presented an informal description of such a machine in the form of an algorithm or iterative process.

left to fill the empty space this created, then accept or reject respectively. Otherwise move to step (2).

(2) Look at the transition function of T. There is a unique ordered pair in it in $Q \times \Gamma \to Q \times \Gamma \times \{left,\ right\}$ with the input state equal to the state next to h and input symbol equal to the symbol next to that. Use marking to check the equality of the indices and find it, cleaning the marks when done. Mark the correct state with $\dot{q}$, remember its output in $\{left,\ right\}$, and travel back beyond the $\circ$. Move the h and the written-down state next to it one cell in the corresponding direction, taking the cell it just moved into, and move it to the other side of the h and state. Then find the marked element of δ, changing the symbol just moved to the output in Γ, and finally changing the state next to h to the state output of the marked element. Then delete the mark in δ and return to step (1).

The h refers to Turing machine "h"ead, because this machine mechanically applies the definition of Turing machine computation to its input! It creates a separate portion of the tape on which to do computation, and places a marker on it representing the input Turing machine's head and its state, and at each step consults its Turing machine input about how to transform the tape and change states, using its various marks to track unboundedly long quantities as it moves from the "fake tape" to the machine it takes as input. When it accepts or rejects, it simply copies the "fake tape" to the front of the real tape and cleans up all the symbols it used, so it computes the same function.

We have constructed a single Turing machine that is capable of emulating all Turing machines that use only binary digits. This is called a *universal Turing machine*, or *UTM*. We could use the same general strategy for any fixed Γ. Further, we could even convert our strategy for converting arbitrary Γ to binary sequences into a formal algorithm and construct a Turing machine that performs it to the input, then behaves like the previously defined UTM. This could emulate all Turing machines up to an easy change of symbols.

We also proved earlier that for every Turing machine, there is another Turing machine which computes an equivalent function. However, given a Turing machine which gives outputs already in binary digits, the machine we constructed does not give exactly the same outputs, instead lengthening them in a way that seems unnecessary. This is a purely technical detail which can be fixed with little effort. We omit this fix.

We can apply this to the UTM we constructed: we can encode parentheses and brackets just as well as any other symbol, and thus construct a UTM with $\Gamma = \{0,\ 1,\ \sqcup\}$ that emulates all Turing machines of the same Γ. Now we are justified in being able to discuss UTMs with $\Sigma = \{0,\ 1\}$, $\Gamma = \{0,\ 1,\ \sqcup\}$ without having to be careful about technicalities. We call this a *binary UTM*. Often when studying Turing machines, one does so in the background of a fixed arbitrary UTM.

Now it is natural to ask whether there are any languages that a Turing machine cannot decide. This is another question that is worth attempting to answer for oneself. *Hint.*[7]

Fix an arbitrary binary UTM. Consider the language $L = \{X : X$ is a sequence of binary digits that, when interpreted by the UTM, eventually accepts or rejects, rather than going on forever.$\}$ This language is called the *halting problem*. Assume for the sake of contradiction that there is some Turing machine T which decides L. Consider the Turing machine T', which given an input x, runs T on the ordered pair $(x,\ x)$. If T says that it halts, i.e. it

[7] The proof can be done in three lines by a diagonalization. But how does one apply diagonalization in this context?

accepts or rejects, then T' goes into an infinite loop. If T says that it runs forever, then T' halts. Now, consider $T'(T')$. If it halts, then it necessarily runs forever. If it runs forever, then it necessary halts. This is absurd. $\square$

This proof is so important, wide-ranging in implication, and exemplary of useful techniques that it is usually taught to students even before it can be made rigorous using UTMs. We gave the "mathematician's proof." To a computer science student the existence of UTMs is obvious: it is a near synonym for "programming language." Here is a three line "computer scientist's proof" referenced in the hint:

(1) Let T take ordered pair (T', x), decides whether Turing machine T', input x, halts.
(2) We can construct *explodeT*, which on input x runs T on (x, x) and does the opposite.
(3) $explodeT(explodeT) \to \perp$. $\square$

This proof is rigorous and contains the actual important insight. It merely fails to define how to interpret the input as a Turing machine which can interpret other components of the input, which is why the mathematician's proof first formalizes the notion of UTM.

We have successfully proven that the language L is not decidable. But in fact, it is recognizable. Simply run the UTM on the input and accept if the emulated Turing machine ever accepts or rejects. Then it accepts if and only if the input halts, and it fails to halt otherwise.

An implication of this is that the complement of L, that is the set of ordered pairs which do not halt plus those which do not represent a Turing machine (an uninteresting component which can be decidably filtered out) is not even recognizable! If it were, one could construct a Turing machine that separated the tape into two components, emulated the machine that recognizes L on one component and emulated the machine that recognizes its complement in the other. Since every pair clearly does or does not halt, such an algorithm would necessarily decide the problem, and thus cannot exist.

Two useful questions: first, why does this argument $fail$ to prove that L is not recognizable? If it could prove this it would prove a false statement, and then we certainly could not trust arguments of this form. Second, how is this proof like other proofs by diagonalization one may have encountered in other contexts?

We also have another proof, this one merely showing existence, that there exist undecidable languages.

(1) Each Turing machine corresponds to a sequence of symbols from a finite set according to our formal definition, of which there are countably many.
(2) Each Turing machine decides at most one language.
(3) A language is a subset of the set of finite sequences of symbols from a finite set, of which there are uncountably many. $\square$

.

This proof, while much less satisfying, generalizes in a different direction: one can replace the word "decide" with "recognize" in it and it will still be valid. Thus the vast majority of languages are not only undecidable, but not even recognizable.

Nonetheless, one might be motivated to study such languages, and the analysis of Turing machines is a powerful tool. Can one construct similar objects which are more powerful and general?

Before the definition, let us first describe the intuition for *Turing machines equipped with an oracle*, or *oracle machines*.

Consider the question, "If the halting problem were decidable by a Turing machine, what other languages would be decidable?" Trivially, the answer is, "all of them and also none of them because we can derive a contradiction."

If we wants to give this question a meaningful answer, we must do something subtler. We equip a Turing machine with a *black box*, a term common outside mathematics which often means "a piece of technology which we use, but do not understand." One also may have heard other mathematicians describing a theorem or lemma they cannot prove, but have used to prove other results, as a black box. The Turing machine can, during its computation, give the black box an input using symbols from Γ, and query it for an output. The black box, or oracle, for a language L will output 1 if the input is a member of the language, and 0 if it is not.

Most people do not know how their electronics work, let alone understand them at the level of mathematical rigor. Similarly the machine does not actually "know" whether an arbitrary machine halts. It can only ask a mysterious oracle, assumed to be trustworthy, individual queries of that form, an unbounded but finite (if the oracle machine halts) number of times.

Definition 3.1. A Turing machine equipped with an oracle A, with A some language with symbols in $\{0,\ 1\}$, is the same as a normal Turing machine, with three additions to the transition function: it has two more output components and one more input component. The first output component lies in $\{left,\ right\}$, the second lies in $\{0,\ 1\}$, and the last input component lies in $\{0,\ 1,\ \emptyset\}$.

The output components describe the behavior of a second head along a second tape. The second head starts at the second cell (which corresponds to 1) of a separate infinite tape indexed by $\mathbb{N} \cup \{0\}$. The second head writes, but does not read, binary digits on the cells of the second tape other than the 0-indexed cell. At the 0 index, it reads, but does not write. At any given step of the computation, the oracle reads all of the digits written on the second tape. If the corresponding sequence is in A, the oracle writes 1 at the 0-indexed cell, and if it is not in A, it writes 0 in the 0-indexed cell. Whenever the second head is not at the 0-index cell, the transition function takes as the value of its new input component $\emptyset$. When the second head is at the 0-index cell, the transition function takes the value at that cell as its new input component.

This formal definition is sufficiently without insight that it is sometimes omitted entirely. It simply states that the Turing machine can write a sequence of symbols, usually copied from its main tape, into a place for the oracle to read: the oracle says whether the sequence is in the language, and the Turing machine can behave differently based on how the oracle answers. We present this definition merely to justify pseudocode. Psuedocode for oracle machines works as follows: we can perform any computations a Turing machine can, but also ask the oracle about the membership of a sequence we have computed in A (referred to as *querying the oracle*), and use its answer in future computations.

One may worry that since we are dealing with uncomputable decision problems, our earlier process for converting all problems into binary problems may fail. This is not the case for the oracle itself: our procedure for converting arbitrary languages to "equivalent" languages using only binary symbols is a computable function even if its input ranges over some uncomputable language. Similarly, given an arbitrary oracle machine, the same process of sectioning the tape into chunks described earlier works to convert it to a binary machine.

Some natural questions to ask regarding oracles: given an oracle for the halting problem, what other languages are decidable?

"Ordered pairs of Turing machine and inputs such that the Turing machine accepts on the input" is decidable with respect to this oracle: take an oracle machine that simply uses the input Turing machine to explicitly construct another Turing machine that does the same thing but, when it would reject the input, instead loops forever, then query the halting oracle about that Turing machine on the given input.

What about "Turing machines that halt on at least one input?" This is also now decidable: given a Turing machine input, let our oracle machine construct from it a Turing machine that emulates it, "weaving" through every possible input in lexicographical order. That is, it performs the first step of the computation on the first input, then the first step of the second input, then the second of the first, then the second of the second, then the first of the third, and so on. Querying the oracle about the constructed machine suffices.

One may think: to make that machine prettier, why can't we let our machine simply explicitly construct an oracle machine that queries the oracle about every input in lexicographical order, and halts when one of them halts, and then query the oracle about that machine? It is worth figuring out for oneself why one cannot do this.

Because the oracle halting problem provides information about whether *Turing machines* halt, not whether *oracle machines* halt! And we cannot possibly use the former to obtain the latter in full generality, because oracle machines with oracle A cannot decide the halting problem for themselves. The exact same proof as before can prove this.[8]

The exact same cardinality proof also works for oracle machines as well, establishing that for any oracle, no matter how much or how powerful the information it contains, a Turing machine equipped with it can only recognize countably many languages. One should also note that every language is trivially decidable by some oracle machine, simply by giving it an oracle for that particular language.

4. K-Complexity and The Real Numbers

Consider an arbitrary sequence of binary digits that is 500 digits long. It seems that most such sequences are in some sense "more complicated" than the sequence of 500 consecutive 0s. Intuitively, one should be able to pick a particular such string and assert it is "more complicated" than the string of 0s of equivalent length. However, most notions of complexity for sequences of symbols, such as those employing probability, require a specified "context" that assigns weights to all elements, and the notion of complexity only holds meaning in that context. Using the theory of computation, one can construct a less contextual notion of sequence complexity, known as *K-complexity* or *Kolmogorov complexity*, which intuitively measures "how hard a particular sequence is to specify."

[8] Of course, oracles can clearly sometimes solve the halting problem for other oracles. Consider the oracle for the empty language: that is, the oracle which always outputs 0, because the language has no members. Any such oracle machine behaves identically to the Turing machine formed by considering the value of the transition function restricted to oracle inputs 0 and $\emptyset$ at appropriate times, those times being computable by a Turing machine that "follows" the oracle machine and checks when the second head enters the read-only section. Thus one can tell whether such a machine halts if one has access to an oracle for the Turing machine halting problem. Thus oracles can be said to be "stronger" or "weaker" than other oracles, and one can study the corresponding ordering on equivalence classes of languages.

Definition 4.1. Fix a binary UTM. The *K-complexity* of a sequence of digits x is the length of the shortest input sequence on which the UTM computes x.

One should first note that this definition does not discriminate between the "Turing machine" part of the input and the "input to that machine" part of the input. This is common in many applications. To interpret this it is useful to remember that a UTM is just a Turing machine, and the form of its input just a human convention for understanding; there is only a fixed deterministic process, and a sequence of binary digits. There is no fundamental distinction between the "Turing machine input" and the rest of the input. More practically, we choose to ignore this because it does not really matter; two Turing machines might compute different functions, but on respective inputs both output the desired x. Specifying x can be done by using many symbols to specify the input component in the UTM, or by using many symbols to specifying the Turing machine component.

Given a way to measure the complexity of something, it is natural to ask what lower bounds one can establish with it. Elegantly, at least half of all strings of length $\leq n$ have K-complexity at least n. Why?

(1) Every input specifies at most one output.
(2) There are $2^{n+1} - 1$ strings of length $\leq n$.
(3) There are $2^n - 1$ inputs of length $\leq n - 1$.
(4) Therefore at least 2^n of the output strings require an input of length at least n.
(5) $2^n/(2^{n+1} - 1) > 1/2$. □

This is an elementary proof by a counting argument. To what degree can one obtain stronger results using more sophisticated methods?

Let us note that the language "$(x,\, n)$ such that n is the K-complexity of x" is not only undecidable, but not recognizable![9] Of course, since we do not care about the time the Turing machine takes, this is equivalent rather than weaker to computing n from x simply by brute-forcing every possibility.

Imagine it were recognizable, with some Turing machine T recognizing it. Then there is another machine T' which, on input n, considers every pair $(x,\, i)$ with x in lexicographical order and $i \leq n$ with i written in normal binary notation and applies T to each. Eventually it will enumerate all such x, then let it output the lexicographically smallest y such that the K-complexity of y is at least n. This machine has some K-complexity U under the fixed UTM, so the value of y corresponding to n, which is $T'(n)$, has K-complexity at most $U + \log_2 n$ But by this algorithm $T'(n)$ has K-complexity n, so letting n large gives a contradiction. □

This result does not seem so surprising: after all, "most" problems are uncomputable. But it is worth noting that for any axiomatic system that is sound, whose proofs are recognizable as described earlier, there is some constant U such that no sequence can be proved to have a K-complexity greater than or equal to U.

One may notice that we earlier claimed that K-complexity assigns meaningful values to individual strings in a way that does not depend on a "context" of other strings, but also required the fixing of a UTM. If one wanted to, for any string, one could pick a UTM for which it has K-complexity 0. In what sense is it nonarbitrary?

K-complexity has regularity that crosses UTMs. Specifically, for any two distinct UTMs T_1, T_2 there is a constant U such that for any string x, the K-complexity of x under T_2 is

[9] One can weaken this to bounds on its K-complexity or otherwise alter it and obtain different degrees of uncomputability, but this is outside scope: this result is presented only to build intuition.

less than its K-complexity under T_1 plus U. This is because all UTMs can simulate each other, so T_2 can simulate T_1 by reading some Turing machine input y. Thus if T_2 computes x on input z, then T_1 also computes it on input yz, upper bounding its complexity. Note that switching T_1 and T_2 may give an entirely different U.

This is an upper bound, but one can easily see there is no corresponding lower bound. This is intuitive: K-complexity measures the difficulty of specifying a sequence of digits, and specification relies on previous information, and that "previous information" may already know the desired sequence.

One can also consider "oracle complexity," that is K-complexity where the UTM is equipped with an oracle and interprets its input, which also encodes an oracle machine, to determine when to query its oracle. All of our previous results are still true for this object, by the exact same proofs.

Now when we refer to K-complexity, we are implicitly fixing an arbitrary UTM. Note that by the previous upper bound, and an easy squeeze argument, if one only cares about K-complexity up to some error which is $>> O(1)$, one does not need to worry about individual UTMs. Much of the study of K-complexity takes place in such a context, including the Point-to-Set Principles.

One may note that real numbers require infinitely many digits to completely specify, thus our definition of Turing machines computing functions is not adequate to describe real-valued functions in general. One way to solve this problem is to define computation a different way when considering real numbers.

Definition 4.2. For some $x \in \mathbb{R}^d$, we say a Turing machine *computes* x if for every input $r \in \mathbb{N}$, represented in binary, the machine outputs a number $q \in \mathbb{Q}^d$ with $|q - x| < 2^{-r}$. Note that every input can be interpreted as a natural number when $\Sigma = \{0, 1\}$.[10]

That is, on input n, the Turing machine, or oracle machine, computes x to "n digits of precision." Instead of directly truncating the digits of the vector components we define computation using a bound on the distance: this eliminates some pathological behavior and makes proofs much easier. One can safely use one's intuition for truncation in this context.

By a now familiar counting argument, for a given oracle A, there are only countably many computable numbers in $\mathbb{R}^d$, since clearly each machine can only compute one.

It is worth considering which numbers are computable by Turing machines, or equivalently oracle machines with oracles for decidable languages. The rationals clearly are, since we can explicitly write into the Turing machine's states what to write on every input. Familiar irrational constants such as π, e, and roots of polynomials with integer coefficients are computable by Turing machine versions of appropriate approximation algorithms. Vectors whose components are computable clearly are computable, and given an algorithm computing $x \in \mathbb{R}^d$ we can clearly compute its components, so it suffices to consider $x \in \mathbb{R}$.

Are there explicit constructions of reals that are not computable? Certainly by definition we cannot give processes to approximate them to arbitrary precision, but we would like to be able to give examples that are constructive "in spirit," as the halting problem was.

Choose an arbitrary binary UTM, and order its possible inputs lexicographically. Consider the real number whose binary expansion has a 1 in the n-th position past the zero if the n-th Turing machine halts, and a 0 if it does not. If this number were computable, one

[10] We interpret blank spaces in the output as division signs and commas separating vector components, so that machines with $\Gamma = \{0, 1\}$ can express numbers in $\mathbb{Q}^d$. This is a purely technical detail of little interest.

could solve the halting problem by approximating the number sufficiently closely to obtain the appropriate digit, so this number must not be computable.

More interestingly, we can consider the sequence $\{a_n\}$, where a_n is a rational number whose k-th digit is 1 if the k-th Turing machine halts in the first n steps and 0 otherwise for $k \leq n$, and is 0 for $k > n$. This sequence clearly consists of computable numbers because their values can be directly computed. Further, we can construct a Turing machine that outputs the a_n on input n, so it is not the case that the values are placed in some clever uncomputable way. Nevertheless, the sequence is monotonically increasing and its limit is the previous uncomputable number. Thus, this particular notion of computing real numbers does not "cooperate" with familiar, convenient tools such as completeness and continuity.[11]

One can easily convert any language into a real number by using the lexicographical ordering to encode an indicator function for it in the binary digits of a real number, as we just did for the halting problem. Further, any real number can be converted to a language: simply convert it to binary, and let the digit in position n after the decimal describe the language's inclusion of the n-th sequence of binary digits in the lexicographical ordering. Note that we must use the lexicographical ordering (or a similarly readable ordering), rather than merely writing the natural number in binary, because otherwise inputs with leading 0s would never be meaningfully defined.

This connection naturally motivates the computational study of reals as infinite sequences of digits, each truncation of which is clearly computable because it is finite. We denote the K-complexity of a number $q \in \mathbb{Q}^d$ by $K(q)$.

Definition 4.3. The *K-complexity at precision* r of a number $x \in \mathbb{R}^d$ is $\min\{K(q) : |q-x| < 2^{-r}\}$. We write it as $K_r(x)$. In natural language, it is the K-complexity of the least complex rational vector within 2^{-r} of x. We can similarly define this for arbitrary oracle A, in which case we write $K^A(q)$ and $K_r^A(x)$.

Naturally one might want to study the asymptotic behavior of this as $r \to \infty$, specifically $\lim_{r\to\infty} \frac{K_r^A(x)}{r}$. However, convergence of this is not guaranteed. Non-rigorously, one might imagine some real number that starts with a highly complex sequence of digits such that $\frac{K_r^A(x)}{r} > \frac{3}{4}$, followed by sufficiently many repetitions of 1 such that $\frac{K_r^A(x)}{r} < \frac{1}{4}$, followed by a sufficiently long highly complex string such that $\frac{K_r^A(x)}{r} > \frac{3}{4}$, alternating forever. Thus we instead consider the limit infimum and limit supremum.

Definition 4.4. For a given oracle A, we write

$$\dim^A(x) = \liminf_{r\to\infty} \frac{K_r^A(x)}{r}$$

$$\mathrm{Dim}^A(x) = \limsup_{r\to\infty} \frac{K_r^A(x)}{r}$$

The names of these values are unfortunately not standardized. We use *lower dimension* and *upper dimension* respectively, but one may also encounter lower and upper *effective dimension*, lower and upper *algorithmic dimension*, and, in the original paper, simply *dimension* and *strong dimension*.

[11] One may be confused as to why an algorithm for computing $\{a_n\}$ then does not suffice to compute the uncomputable number. This is because, for sufficiently large n, there are $k \leq n$ which the Turing machine cannot know are "supposed" to be 0 or 1, precisely because one cannot solve the halting problem.

It is worthwhile to consider some example cases. A number x computable with respect to an oracle A has $\mathrm{Dim}(x) = 0$, since the oracle machine computing it has some complexity U, so $K_r(x)$ is at most $U + \log_2 r$, and in the limit $\frac{U+\log_2 r}{r}$ goes to 0.

A number $x \in \mathbb{R}^d$ that is in some sense "maximally hard to approximate" for a given oracle A, has $\dim(x) = d$.

Given $x \in \mathbb{R}$, $y = (x, x) \in \mathbb{R}^2$, $\dim(y) = \dim(x)$. Its lower dimension cannot be lower, since any approximation of x can be extracted from an approximation of y with an input only more complex by a constant. And it cannot be higher, since an approximation of y can be similarly extracted from an approximation of x. Similarly, but with the implication directions of the proof reversed, we have $\mathrm{Dim}(y) = \mathrm{Dim}(x)$.

Finally, it is worth noting that one's intuition for the behavior of Turing machines approximating computable numbers should *not* transfer to numbers with nonzero lower dimension. We are inclined to to think of some reasonable iterative process gradually approximating the number, but by the argument three paragraphs before, we know this is not the case, since such a process could be used to construct a Turing machine. Instead we have easy-to-specify rational numbers that approximate the number appearing seemingly "at random," with no computable underlying pattern, with the lower and upper dimensions merely measuring how easy-to-specify such coincidentally close numbers are in the limit.

Now that we have tools to study the asymptotic complexity of real numbers with respect to oracles, it is natural to ask how they interact with the real numbers geometrically and arithmetically.

5. The Point-to-Set Principles

Recalling some definitions: taking a set $E \subseteq \mathbb{R}^d$, the *s-dimensional Hausdorff measure* of E, denoted as $H^s(E)$ is

$$\lim_{\delta \to 0^+} H^s_\delta(E), \text{ where } H^s_\delta(E) = \inf \left\{ \sum_{i \in \mathbb{N}} \mathrm{diam}(U_i)^s : U_i \text{ is a cover of } E \text{ and } \forall i \ \mathrm{diam}(U_i) < \delta. \right\}$$

The *Hausdorff dimension* of a set E, denoted by $\dim_H(E)$, is the unique s such that

$$s = \sup_{d \in \mathbb{R}} \{d : H^d(E) = \infty\}, \text{ equivalently } s = \inf_{d \in \mathbb{R}} \{d : H^d(E) = 0\}$$

We also have the *s-dimensional packing measure.* We let $B_i(x_j)$ represent the *closed* ball centered at x_j with *index i*, *not* the diameter i. First construct the appropriate pre-measure

$$P^s_0(S) = \limsup_{\delta \to 0^+} \left\{ \sum_{i \in \mathbb{N}} \mathrm{diam}(B_i(x_i))^s : \text{all } B_i \text{ are pairwise disjoint and } \forall i \ x_i \in S \text{ and } \mathrm{diam}(B_i) < \delta/2 \right\}$$

Then, to turn this into an actual measure, we take

$$P^s(E) = \inf \left\{ \sum_{j \in \mathbb{N}} P^s_0(U_j) : U_j \text{ is a cover of } E \right\}$$

Then we have the *packing dimension* of a set E, denoted by $\dim_P(E)$, is the unique s such that

$$s = \sup_{d \in \mathbb{R}} \{d : P^d(E) = \infty\}, \text{ equivalently } s = \inf_{d \in \mathbb{R}} \{d : P^d(E) = 0\}$$

In different ways these definitions describe the "thickness" of a set E in $\mathbb{R}^d$, and thus we might naturally assume they could be manipulated to give some lower bound on the K-complexity of the elements of E. This relationship is miraculously simple, as proven by [LL18]:

Theorem 5.1 (Point-to-Set Principle)**.**

$$\dim_H(E) = \min_{A \subseteq N} \sup_{x \in E} \dim^A(x)$$

$$\dim_P(E) = \min_{A \subseteq N} \sup_{x \in E} \mathrm{Dim}^A(x)$$

A is an oracle, and we can denote arbitrary oracles for decision problems over elements of Σ^* as subsets of $\mathbb{N}$ as before, specifically by taking the lexicographical ordering and letting the subset's inclusion of $n \in \mathbb{N}$ represent the decision problem's acceptance of the corresponding element of Σ^*.

It is worth noting these are *minima*, not infima. This will emerge naturally from the proof, which constructs such minima explicitly. We prove the part regarding Hausdorff dimension first.

Proof of Hausdorff part. Let us first show

$$\min_{A \subseteq \mathbb{N}} \sup_{x \in E} \dim^A(x) \leq \dim_H(E)$$

Letting $\dim_H(E) = s$, fix $s' > s$. Then by a familiar property of Hausdorff dimension $H^{s'}(E) = 0$, so for sufficiently small δ we have $H^{s'}_\delta(E) < 1$.

Let $\delta_0 = 2^{-m}$, $\delta_0 < \delta$. Define $\{\delta_n\}$ a sequence starting with δ_0 and $\delta_{n+1} = \frac{\delta_n}{2}$. Then let, for each n, $\{U_{i,n}\}$ be a countable cover of E, its elements indexed by i with $\mathrm{diam}(U_{i,n}) < \delta_n$ and

$$\sum_{i \in \mathbb{N}} \mathrm{diam}(U_{i,n})^{s'} < 1.$$

In natural language, we have constructed a countable family of countable covers $\{U_{i,n}\}$ of E with exponentially decreasing rational δ, all of whose measure are bounded by a constant. The positions of these coverings naturally gives us information about the approximate position of the elements of E, so now we are going to construct an appropriate oracle that converts these coverings into a way to "locate" arbitrary elements of E. Before we do so, we must deal with the boring case of singleton sets.

First, consider the set $\{x : \exists i, n \text{ with } \{U_{i,n}\} = \{x\}\}$. In natural language, this is the set of elements of E which are ever covered by a singleton within our family of coverings. This set is clearly countable since it is a countable union of countable sets, let us label it $\{x_j\}$. Now we let our oracle A encode the function: $f(j,l) =$ "the first l digits of x_j") using its even numbers. Then there is some oracle machine T^A which on input (J, L) queries the oracle to obtain the digits of x_j. Then all elements of $\{x_j\}$ are computable with respect to the oracle A, so by previous reasoning for all such elements $\dim^A(x_j) = 0$. Thus without loss of generality we can ignore the elements ever in singleton sets, and look only at the interesting elements.

In the odd numbers of the oracle, we encode a function as follows:

- $f : \mathbb{N}^2 \to \mathbb{Q}^d$, denoted $(j, r) \to q$
- $\exists\ i, n$ such that $\exists u \in U_{i,n}$ with $|f(j,r) - u| < \delta_{r+2}$ and $\mathrm{diam}(U_{i,n}) \in [\delta_{r+2}, \delta_{r+1})$

- For each possible such $U_{i,n}$ there is a j such that $f(j,r)$ maps to a nearby u for the appropriate r. Further, $f(1,r)$, $f(2,r)$, $f(3,r)$... all satisfy the constraint for distinct $U_{i,n}$. Thus we only need as many distinct possible j as there are sets satisfying the second bullet point.

In natural language, this oracle encodes a rational number q for each element in every covering in our family with q exponentially close to that element, indexed by decreasing size-exponent in r and indexed individually among those of similar sizes by j. It must be noted that they are indexed by their diameter, *not* by the δ-constraint of their family, and that they use a *lower* bound as well as an upper bound.[12]

Let us note that given some $(U_{i,n})$ in one of our covers, it can only correspond to $f(i,r)$ for some specific value of r. Let us also note that the set corresponding to $f(j,r)$ can only occur in the first $r+2$ elements of our descending-diameter family, because beyond that point its minimum-diameter requirement contradicts the original maximum-diameter requirement.

Finally, recall that

$$\sum_{i\in\mathbb{N}} \operatorname{diam}(U_{i,n})^{s'} < 1.$$

By arithmetic each family can contain at most

$$\frac{1}{\delta_0}(2^{-r+2})^{-s'}$$

sets $(U_{i,n})$ for a particular value of r. This quantity is of course $O(2^{rs'})$ with respect to r since we already fixed s'. Thus the total number of such sets is at most on the order

$$\sim c_0(r+2)2^{rs'}$$

Let us observe that every element $x \in E$ is contained in some $U_{i,n}$ for infinitely many r. By our definition of q and the diameter constraint on the set, we have $|f(i,r)-x| < 2^{-r}$.

Now, consider an oracle machine T^A which on input (i,r) queries the oracle to obtain $f(i,r)$, and outputs it. This input requires $\log_2 i + \log_2 r + o(\max\{\log_2 i, \log_2 r\})$, the first two to specify the respective natural numbers and third due to a technical detail of computation relating to unambiguously reading the separate numbers.

So for all $x \in E$ we have infinitely many r such that $T^A(i,r)$ gives some q with $|q-x| < 2^{-r}$. T^A has constant K-complexity, so at these values of r $K_r(x)$ is at most

$$c_1 + \log_2(c_0(r+2)2^{rs'}) + \log_2 r + \epsilon \sim rs' + o(r)$$

By simple arithmetic and applying the definition of lower dimension we therefore have, for all $s' > s$, for all $x \in E$

$$\dim^A(x) \le s' \quad \square$$

This completes the first inequality. For the other direction, assume for contradiction there is some oracle A and $s' < s$ such that $\forall x \in E$

$$\dim^A(x) \le s'$$

. Now, let $c \in (s', s)$ and $c' \in (c, s)$. We use $B_r(x)$ to denote the open ball of diameter r centered at x.

[12] If one is skeptical of the construction of such an oracle, it is worthwhile to remember how one would construct it, encoding functions directly by explicitly assigning 1s and 0s to the appropriate ordered pairs, then "weaving" or more generally the lexicographical ordering to encode multiple functions.

Consider the finite set of open balls

$$U_r = \{B_{2^{-r}} : q \in Q, K_r^A(q) < rc\}$$

Note that there are at most 2^{rc} many, since there are only that many UTM inputs to specify real numbers. Note that, by assumption, every element of E lies in U_r for infinitely many r. Letting

$$W_r = \bigcup_{k \geq r} U_k$$

So we have $E \subseteq W_r$ for all r. These form countable coverings of E for arbitrarily small diameters, so let us use them to find an upper bound of $H^{c'}(E)$.

$$H^{c'}(E) \leq \lim_{r \to \infty} \sum_{S \in W_r} \mathrm{diam}(S)^{c'}$$

which is equal to

$$\sum_{r'=r}^{\infty} \sum_{S \in U_{r'}} \mathrm{diam}(S)^{c'} \leq \sum_{r'=r}^{\infty} 2^{r'c}(2^{1-r'})^{c'}$$

The figure on the right rearranges to $2^{c'}(2^{r'(c-c')})$, which goes to 0 in the limit as r increases. So $H^{c'}(E) = 0$, so by the definition of Hausdorff dimension we have $\dim_H(E) \leq c' < s$, a contradiction. □

This direction has a nice corresponding intuition: the infinite intersection $\bigcap_{r \to \infty} W_r$ itself has Hausdorff dimension c,[13] meaning rational numbers q which have $\frac{K_r(q)}{q} < c$, which is rare when $c < d$, are "asymptotically thin" in exactly the same way that a set of Hausdorff dimension c is "asymptotically thin." One also may note W_r's rough similarity in definition to the Cantor set.

The proof of the first direction does not have a nice geometric intuition that the author knows of, and is best understood through the counting argument that it presents. It is worth noting that the counting argument emerges simply from a careful, attentive understanding of Hausdorff dimension: given the Hausdorff dimension's restriction on the "thickness" of a set, there is an bound on the necessary number of indexes to index any "sufficiently not precise" covering elements, and that this can be done for arbitrary δ, giving asymptotic information.

There is also a computational view of this proof. When one sees a "minimum supremum," it brings to mind minimaxes and other similar structures which can be understood adversarially: one comes up with a "strategy" to pick an oracle to minimize the effectiveness of the opponent's "strategy" to pick an element. This analogy is fairly standard, and also commonly used to understand quantifier alternation.

From this perspective, Hausdorff measure, which is a monotone increasing limit of infima of coverings, is a limit of an adversarial process of the form, "how well can I do by picking small δ such that my opponent must cover E with increasingly many diameter δ sets?" Then the Point-to-Set principle is one of the many results allowing one to switch the order of two limit-taking operations, specifically in the context of adversarial process. Measure theoretic definitions of volume, insofar as they rely on outer measures based on "strategies" of coverings or the like, are already "computational" from this standpoint.

[13] This can be checked from above trivially, and checked from below by applying this principle in an argument from contradiction.

Proof of packing part. First let us prove

$$\min_{A\subseteq\mathbb{N}}\sup_{x\in E}\mathrm{Dim}^A(x)\leq\dim_P(E)$$

Let $\dim_P(E)=s$. Then for any $s'>s$, we have $P^{s'}(E)=0$. So we can pick some countable covering $\{E_i\}$ with $E\subseteq\{E_i\}$ and

$$\sum_{i\in\mathbb{N}}P_0^{s'}(E_i)<1$$

Now we can study the packing pre-measure over $\{E_i\}$ and construct the appropriate oracle. For each $r\in\mathbb{N}$, and a fixed i, consider each packing of E_i with exclusively closed balls of radius 2^{-r-2} that is maximal, in the sense that no further balls can be added. Call this packing $\{B_{j,i,r}\}$, indexed by j. For sufficiently large r, we have by our prescription on the $\{E_i\}$ that there are at most $2^{rs'+2s'}$ balls. Further, notice that for each $x\in E_i$, there is necessarily some ball $B_{j,i,r}$ such that $\forall q\in B_{j,i,r}\ |q-x|<2^{-r}$, because the packing is maximal.

Now the proof is quite clear, following the strategy previously used for Hausdorff dimension. Let A be an oracle taking $(r,i,j)\to q$ rational $\in B_{j,i,r}$. Consider an oracle machine T^A which on an input (R,I,J) queries the oracle then outputs the point given by the oracle. Then for any $x\in E$, the K-complexity of the output of this machine is at most a constant corresponding to the description length of the machine, a constant corresponding to the i such that $x\in E_i$, $\log_2 r$ to represent r, $\log_2(2^{rs'+2s'})$ to represent the j-index, and a small term for unambiguous reading of the code. This works for every sufficiently large value of r, so applying the definition of upper dimension we have that it is lower than s' for every $s'>s$, and we are done. □

Now let us show

$$\dim_P(E)\leq\min_{A\subseteq N}\sup_{x\in E}\mathrm{Dim}^A(x)$$

Assume for contradiction there exists an oracle A such that $\forall x\in E$, $\mathrm{Dim}^A(x)\leq s'$, with $s'<\dim_P(E)=s$. Then let us choose $c\in(s',s)$ and define the sets

$$C_k=\bigcup\{B_{2^{-k}}(q):K^A(q)\leq kc\}$$

$$E_k=\bigcap_{i\geq k}C_k$$

Note that, by assumption, every element in E is in all E_k for sufficiently large k. For fixed k, consider a packing $V_{r,k}$ of E_k with balls of diameter 2^{-r} for $r\geq k$, $r\in\mathbb{R}$.

Let $B_\delta(x)$ be an element of this packing, and $e=\lceil-\log_2\delta\rceil$. Since this an element of the packing, we have $x\in E_k$ and therefore $x\in C_e$, so there is some q with $K^A(q)\leq ec$. Since $\frac{\delta}{2}<2^{-e}<\delta$, and each ball is disjoint, we can find different q for each ball. Since there at most 2^{ec+1} rational numbers of appropriate K-complexity, there are at most 2^{ec+1} balls of diameter in $[2^{1-e},2^{2-e})$.

Now, as in a previous argument, let $c'\in(c,s)$. Then

$$P^{c'}_{2^{-r}}(E_k)\leq\sum_{S\in V_{r,k}}\mathrm{diam}(S)^{c'}$$

$$\leq \sum_{j\geq r}^{\infty} 2^{ec+1}(2^{1-e})^{c'}$$

As before, this goes to 0, so for each k we have $P_0^{c'}(E_k) = 0$. As we noted earlier,

$$E \subseteq \bigcup_{k\in\mathbb{N}} E_k$$

Thus $P^{c'}(E) = 0$, with $c' < s$, contradicting our assumption that $\dim_P(E) = s$. □

These proofs only differ meaningfully from the source material in the first direction of the Hausdorff part: our method is slightly weaker, as the original paper does not include an $(r' + 2)$ inside the logarithm, but this allows us to use a more digestible proof with fewer indices, and it is not relevant in the limit.

For such a counterintuitive result, these proofs can seem unsatisfyingly simple. The construction of an appropriate countably infinite sequence that allows one to switch the order of infinitary operations brings to mind proofs of equivalence of definitions in elementary topology. The ability to take an alternate perspective when it is appropriate tends to be useful, of course; the original paper gives some further development of K-complexity over $\mathbb{R}^d$, then uses the point-to-set principles to prove the 2-dimensional case of the Kakeya conjecture.

It is not clear to this author how these results could be extended to statements that are more geometrically powerful, and the lack of powerful machinery in the equivalence proof suggests that it may not be possible to do so. However, being able to choose between equivalent definitions as contextually appropriate to give short and elegant proofs of easy problems is sometimes the pathway to the insight that solves more difficult problems.

6. Appendix

One may notice that while we gave a formal definition of Turing machines, we only gave an informal description of the computation process. A formal definition is presented here. For $x \in \Sigma^*$, we denote as x_i the i-th symbol of x, and $|x|$ the number of symbols in x.

Definition 6.1. Given a Turing machine $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$, we say a *computation history on input* x is a sequence $\{C_i\}$ of elements of $\Gamma^{\mathbb{N}} \times Q \times \mathbb{N}$ such that

(1) $C_0 =$ (the function f taking $i \to x_i$ if $i \leq |x|$ and $\to \sqcup$ otherwise, q_{start}, 1)
(2) The function component of $C_{n+1} = (f_{n+1}, q_{n+1}, a_{n+1})$ is identical to that of $C_n = (f_n, q_n, a_n)$, except at a_n, where it is equal to the Γ output component of $\delta(q_n, f_n(a_n))$. Similarly, q_{n+1} is the Q output component of $\delta(q_n, f_n(a_n))$. Finally, $a_{n+1} = a_n + 1$ if the output in $\{left, right\}$ of $\delta(q_n, f_n(a_n))$ is $right$, and it is equal to $a_n - 1$ if that output is $left$.

There is nothing particularly interesting, conceptual, or subtle about this definition compared to the non-rigorous description of computation. The function from $\mathbb{N} \to \Gamma$ formalizes the infinite tape containing symbols from Γ, and taking that function on a particular natural number n simply means "the symbol at position n on the tape." Taking $\delta(q_n, f_n(a_n))$ simply represents how a Turing machine head in state q_n at position a_n on a tape with symbols described by f_n behaves, and the previous definition converts that behavior into the notation of functions representing tapes and natural numbers representing machine head positions.

Definition 6.2. Given a Turing machine $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$ and an input x, we say the Turing machine *accepts* x if there exists a computation history of that machine on that input which ever has a step whose component in Q is q_{accept}, we say it *rejects* x if there exists a computation history of that machine on that input which ever has a step whose component in Q is q_{reject}. Otherwise, the Turing machine will continue forever, so we naturally say it *does not halt.*

Note that this is well-defined, since each step of the computation history is entirely defined by the output of a function on the previous step. For a given machine and input, there is only one computation history. As before, a Turing machine *recognizes a language* L if it accepts an input x if and only if $x \in L$. If it also rejects on all other inputs, it *decides* L. Constructing the analogous formal definitions for Turing machines that compute functions, rather than merely recognize languages, and for oracle machines, are useful exercises.

In this text we are careless about what happens if the Turing machine tries to move left while already at the leftmost component of the tape. Without loss of generality we can ignore it; Turing machines that fail to move, "break" and are forced to halt, are never even permitted to move left in such a position, or even have a two-way infinite tape are all computationally equivalent. (One can prove fairly easily that each one can emulate any of the others.)

We also give the previously mentioned explicit construction of a Turing machine that, given an element of Σ^* where $\Sigma = \{0, 1, \circ\}$, accepts it if and only if it is of the form "$\circ\ X \circ Y \circ Z$" where $Y + Z = X$, all of which are binary numbers, and rejects otherwise. It does so first by going over the entire input, making sure it is of the form '$\circ\ X \circ Y \circ Z$' with X, Y, and Z binary integers at all. Then it adds leading 0s to Y in advance to make the later addition easier until it has at least as many 0s as there are digits of Z, plus one. Then it adds Z to Y digit by digit, and finally checks whether the resulting number equals X. If it does, it accepts. Otherwise, it rejects. We use → to denote the image of an element of $Q \times \Gamma$ under δ.

(1) $Q = \{q_{start}, q_{check0}, q_{check1}, q_{check2}, q_{check3}, q_{check4}, q_{check5}, q_{startaddition}, q_{goleft}, q_{findunmarkedY}, q_{goright}, q_{checkiffinishedZ}, q_{findunfinishedZ}, q_{replace0}, q_{replace1}, q_{replacecircle}, q_{bounceonce}, q_{goleftlast}, q_{replace0last}, q_{replace1last}, q_{replacecirclelast}, q_{bounceonceagain}, q_{beginarithmetic}, q_{bring0across}, q_{bring1across}, q_{add0digitY}, q_{add1digitY}, q_{carrythe1}, q_{restartarithmetic}, q_{clearmarks0}, q_{clearmarks1}, q_{gotoendY0}, q_{gotoendY1}, q_{begincomparison}, q_{goocheckXdigit0}, q_{goocheckXdigit1}, q_{leftwardsfinalscan}, q_{rightwardsfinalscan}, q_{accept}, q_{reject}\}$
(2) $\Sigma = \{0, 1, \circ\}$
(3) $\Gamma = \{0, 1, \circ, \sqcup, \dot{0}, \dot{1}\}$

We elaborate the transition function as we explain what it does. First the machine should check that the first symbol is a circle, so we have

- $(q_{start}, \sqcup)$, $(q_{start}, 0)$, and $(q_{start}, 1) \to (q_{reject})$.

Note the slight abuse of notation of not specifying the output symbol and movement direction of this step; they are irrelevant since the computation has just ended. Now, our transition function, having seen a circle, should check to see the next symbol is a digit. After that, it does not matter how many consecutive digits there are, so it should loop until it sees a different symbol. If it sees a blank cell it should reject, and once it sees a ∘ it should repeat this whole process once more. Then it should make sure there is at least one digit after that, and no ∘.

- $(q_{start}, \circ) \to (q_{check0}, \circ, right)$.

- $(q_{check0},\, \sqcup)$ and $(q_{check0},\, \circ) \to (q_{reject})$ This rejects if the input has ended prematurely or has two consecutive $\circ$.
- $(q_{check0},\, 0) \to (q_{check1},\, 0,\, right)$ and $(q_{check0},\, 1) \to (q_{check1},\, 1,\, right)$ respectively. Note the different outputs: we don't want to accidentally change digits of X.
- $(q_{check1},\, \sqcup) \to (q_{reject})$.
- $(q_{check1},\, 0) \to (q_{check1},\, 0,\, right)$ and $(q_{check1},\, 1) \to (q_{check1},\, 1,\, right)$ respectively. When it encounters another digit, it simply does not change it and continues to the right.
- $(q_{check1},\, \circ) \to (q_{check2},\, \circ,\, right)$.
- $(q_{check2},\, \circ)$ and $(q_{check2},\, \sqcup) \to (q_{reject})$. Again, this rejects inputs that end too early or have consecutive $\circ$.
- $(q_{check2},\, 0) \to (q_{check3},\, 0,\, right)$ and $(q_{check2},\, 1) \to (q_{check3},\, 1,\, right)$ respectively.
- $(q_{check3},\, \sqcup) \to (q_{reject})$.
- $(q_{check3},\, 0) \to (q_{check3},\, 0,\, right)$ and $(q_{check3},\, 1) \to (q_{check3},\, 1,\, right)$ respectively.
- $(q_{check3},\, \circ) \to (q_{check4},\, 0,\, right)$. Now we have verified the three $\circ$ and the digits between them. We now want to verify the presence of at least one more digit and the absence of any further $\circ$.
- $(q_{check4},\, \sqcup)$ and $(q_{check4},\, \circ) \to (q_{reject})$.
- $(q_{check4},\, 0) \to (q_{check5},\, 0,\, right)$ and $(q_{check4},\, 1) \to (q_{check5},\, 1,\, right)$.
- $(q_{check5},\, \circ) \to (q_{reject})$.
- $(q_{check5},\, 0) \to (q_{check5},\, 0,\, right)$ and $(q_{check5},\, 1) \to (q_{check5},\, 1,\, right)$.
- $(q_{check5},\, \sqcup) \to (q_{startaddition},\, \sqcup,\, left)$. This check state acts differently than all previous because when it sees a blank symbol, it is clear that the input is of the correct form, rather than of the incorrect form. So it goes to the state which begins the process of addition.

This is where the use of marked digits becomes relevant. For our later convenience, we are going to add leading 0s to Y until it is as long as Z. Our machine has to look at the length of Z, but since it can be arbitrarily long, it cannot remember all of that information using only its states. So we will make the machine head "bounce" back and forth between the two numbers, marking the rightmost digit of each, until one runs out of unmarked digits, at which point it will be clear which is longer.

- $(q_{startaddition},\, 0) \to (q_{goleft},\, \dot{0},\, left)$ and $(q_{startaddition},\, 1) \to (q_{goleft},\, \dot{1},\, left)$ respectively. After encountering an unmarked digit in Z, we switch states and begin to move to Y. When encountering a marked digit, it should continue looking in Z for an unmarked digit, so we have that:
- $(q_{startaddition},\, \dot{0}) \to (q_{startaddition},\, \dot{0},\, left)$ and $(q_{startaddition},\, \dot{1}) \to (q_{startaddition},\, \dot{1},\, left)$ respectively.
- $(q_{goleft},\, 0)$, $(q_{goleft},\, 1)$, $(q_{goleft},\, \dot{0})$, $(q_{goleft},\, \dot{1})$ all to themselves and $left$. The purpose of q_{goleft}, is to literally "go left" from Z to Y and track when it has reached it.
- $(q_{goleft},\, \circ) \to (q_{findunmarkedY},\, \circ,\, left)$. Once it passes the $\circ$ into Y, it should look for an unmarked digit. If it finds one, it should mark it and go back right.
- $(q_{findunmarkedY},\, \dot{0})$ and $(q_{findunmarkedY},\, \dot{1})$ to themselves and $left$.
- $(q_{findunmarkedY},\, 0) \to (q_{goright},\, \dot{0},\, right)$ and $(q_{findunmarkedY},\, 1) \to (q_{goright},\, \dot{1},\, right)$.
- $(q_{goright},\, 0)$, $(q_{goright},\, 1)$, $(q_{goright},\, \dot{0})$, and $(q_{goright},\, \dot{1}) \to$ themselves and $right$.
- $(q_{goright},\, \circ) \to (q_{checkiffinishedZ},\, \circ,\, right)$. We mark digits from right to left, but when we are reading Z we read it from left to right, which means that rather than

ignoring marked digits, we should ignore unmarked digits, and to check if the process is complete, we simply need to check if the digit right after the $\circ$ is marked.

- $(q_{checkiffinishedZ}, 0) \to (q_{findunfinishedZ}, 0, right)$ and $(q_{checkiffinishedZ}, 1) \to (q_{findunfinishedZ}, 1, right)$ respectively.
- $(q_{findunfinishedZ}, 0) \to (q_{findunfinishedZ}, 0, right)$ and $(q_{findunfinishedZ}, 1) \to (q_{findunfinishedZ}, 1, right)$ respectively. When this state reaches a marked digit, it obviously doesn't need to mark it; rather, it needs to mark the digit to the left of it and repeat this process. So we can just bring to it back to $q_{startaddition}$ and move it to the left.
- $(q_{findunfinishedZ}, \dot{0}) \to (q_{startaddition}, \dot{0}, left)$ and $(q_{findunfinishedZ}, \dot{1}) \to (q_{startaddition}, \dot{1}, left)$ respectively.

We've elaborated the cases in which this part of the process does not terminate. However, it eventually will. If Y runs out of digits first, we want to add a leading 0 to it and repeat the process. If Z runs out of digits first, we want to add one more leading 0 to Y in case they both start with 1 which would require us to carry the digit later, then begin the process of actually adding the numbers.

- $(q_{findunmarkedY}, \circ) \to (q_{replace0}, \circ, right)$. If the machine head is in $q_{findunmarkedY}$ and reaches the $\circ$, that means Y has already been completely marked, so it needs a leading 0. We need to place a leading 0 there, but that would change the value of the number if that digit is a 1, so we need to move all the symbols one cell to the right. We do this by constructing a state corresponding to each potential symbol that reads a symbol, moves to the right, writes the symbol it just read, and repeats. The states labeled *replace* will do just that. Further, since we are iterating the process, we want to remove the marks on digits, so these states will "forget" that the digits they read are marked.
- $(q_{replace0}, 0)$, $(q_{replace0}, 1)$, and $(q_{replace0}, \circ) \to (q_{replace0}, 0, right)$, $(q_{replace1}, 0, right)$, and $(q_{replacecircle}, 0, right)$ respectively. Deleting the marks, it takes $(q_{replace0}, \dot{0})$ and $(q_{replace0}, \dot{1}) \to (q_{replace0}, 0, right)$ and $(q_{replace1}, 0, right)$ respectively. The two other *replace* states should work the same, but place their corresponding character on the tape rather than a zero.
- $(q_{replace1}, 0)$, $(q_{replace1}, 1)$, and $(q_{replace1}, \circ) \to (q_{replace0}, 1, right)$, $(q_{replace1}, 1, right)$, and $(q_{replacecircle}, 1, right)$ respectively. Again, deleting marks, we have $(q_{replace1}, \dot{0})$ and $(q_{replace1}, \dot{1}) \to (q_{replace0}, 1, right)$ and $(q_{replace1}, 1, right)$ respectively.
- $(q_{replacecircle}, 0)$, $(q_{replacecircle}, 1)$, and $(q_{replacecircle}, \circ) \to (q_{replace0}, \circ, right)$, $(q_{replace1}, \circ, right)$, and $(q_{replacecircle}, \circ, right)$ respectively. Removing marks, we have $(q_{replacecircle}, \dot{0})$ and $(q_{replacecircle}, \dot{1}) \to (q_{replace0}, \circ, right)$ and $(q_{replace1}, \circ, right)$ respectively.

When these states complete the process of moving all the characters to the right and removing their marks, they will reach a blank symbol, at which point we want them to write down the final character they have stored, and begin the process of adding leading 0s again. So we have

- $(q_{replace0}, \sqcup) \to (q_{bounceonce}, 0, right)$ and $(q_{replace1}, \sqcup) \to (q_{bounceonce}, 1, right)$ respectively.
- $(q_{bounceonce}, \sqcup) \to (q_{startaddition}, \sqcup, left)$. We need this state because as we defined Turing machines, the head must move left or right at every step. After the *replace* states end, if we had them move to the left, rather than being on the final digit of Z,

they would be one digit left of it. So we have them move to the right and enter a state that always moves left once before entering $q_{startaddition}$ and restarting the process.

One might note that we have not defined the outputs of δ on inputs such as ($q_{replacecircle}$, $\sqcup$). This is because, as we are constructing the algorithm, this combination of state and tape input should never occur. Without loss of generality, we can simply map all such combinations to q_{reject}. This particular example never occurs because the algorithm only ever reaches the replacement process if the input is in the correct form, in which case every $\circ$ is followed by at least one 0 or 1, so when moving all symbols one cell to the right, the machine will never have to copy a $\circ$ onto a blank cell.

Now we have completed the description of what happens when Y runs out of digits before Z. This process will iterate until Y is at least as long as Z. Once this occurs, either $q_{startaddition}$ will encounter a $\circ$ or $q_{checkiffinishedZ}$ will encounter a marked digit. In the first case, the machine head will be at the $\circ$ between Y and Z and in the second it will be at the digit that immediately follows it. In both cases we want the head to move to the beginning of Y, add one more leading digit, then begin the actual process of addition.

- ($q_{startaddition}$, $\circ$) $\to$ ($q_{goleftlast}$, $\circ$, $left$).
- ($q_{checkiffinishedZ}$, $\dot{0}$) $\to$ ($q_{startaddition}$, $\dot{0}$, $left$) and ($q_{checkiffinishedZ}$, $\dot{1}$) $\to$ ($q_{startaddition}$, $\dot{1}$, $left$).
- ($q_{goleftlast}$, 0), ($q_{goleftlast}$, 1), ($q_{goleftlast}$, $\dot{0}$), ($q_{goleftlast}$, $\dot{1}$) all to themselves and $left$. This is exactly the same as q_{goleft}. We give this state a new name because when it eventually reaches a $\circ$, it should go to a special set of replace states that "remember" that this is the last time we perform this process.
- ($q_{goleftlast}$, $\circ$) $\to$ ($q_{replace0last}$, $\circ$, $right$). The *replace last* states, named for being the last instances of replacement, work exactly the same as the previous *replace* states, so we omit writing their identical behavior. They only differ in behavior upon encountering a blank symbol.
- ($q_{replace0last}$, $\sqcup$) and ($q_{replace1last}$, $\sqcup$) to ($q_{bounceonceagain}$, 0, $right$) and ($q_{bounceonceagain}$, 1, $right$). We add this bounce state as an intermediary to a change to another state for exactly the same reasons as previously.
- ($q_{bounceonceagain}$, $\sqcup$) $\to$ ($q_{beginarithmetic}$, $\sqcup$, $left$).

Entering $q_{beginarithmetic}$ is the beginning of the actual process of adding Z to Y. At each step, we we want the algorithm to mark the rightmost unmarked digit of Z, remember it, then travel to the corresponding digit of Y, then add them. Y may be arbitrarily long, so we aren't able to remember the digit position of Y using states. So we mark a digit of Y, starting from the right, every time we mark a digit of Z, and thus we know which digits of Y we have finished adding to.

- ($q_{beginarithmetic}$, 0) $\to$ ($q_{bring0across}$, $\dot{0}$, $left$) and ($q_{beginarithmetic}$, 1) $\to$ ($q_{bring1across}$, $\dot{1}$, $left$) respectively.
- ($q_{beginarithmetic}$, $\dot{0}$) $\to$ ($q_{beginarithmetic}$, $\dot{0}$, $left$) and ($q_{beginarithmetic}$, $\dot{1}$) $\to$ ($q_{beginarithmetic}$, $\dot{1}$, $left$) respectively. As we described, this ignores those digits and looks for an unmarked one.
- ($q_{bring0across}$, 0), ($q_{bring0across}$, 1), ($q_{bring1across}$, 0), ($q_{bring1across}$, 1) all $\to$ themselves and $left$.
- ($q_{bring0across}$, $\circ$) $\to$ ($q_{add0digitY}$, $\circ$, $left$) and ($q_{bring1across}$, $\circ$) $\to$ ($q_{add1digitY}$, $\circ$, $left$) respectively. The *bring* states ignore all symbols until they read a $\circ$, detecting they

have crossed over into Y. When they do so, they will look for the first unmarked symbol to add their digit to.

- $(q_{add0digitY},\ \dot{0})$, $(q_{add0digitY},\ \dot{1})$, $(q_{add1digitY},\ \dot{0})$, and $(q_{add1digitY},\ \dot{1}) \to$ themselves and $left$.
- $(q_{add0digitY},\ 0) \to (q_{restartarithmetic},\ \dot{0},\ right)$ and $(q_{add0digitY},\ 1) \to (q_{restartarithmetic},\ \dot{1},\ right)$ respectively. Adding 0 obviously does not change the corresponding digit.
- $(q_{add1digitY},\ 0) \to (q_{restartarithmetic},\ \dot{1},\ right)$ and $(q_{add1digitY},\ 1) \to (q_{carrythe1},\ \dot{0},\ left)$.
- $(q_{carrythe1},\ 0) \to (q_{restartarithmetic},\ 1,\ right)$ and $(q_{carrythe1},\ 1) \to (q_{carrythe1},\ 0,\ left)$. Note that the state which carries a 1 does not mark the digits it alters. Otherwise, in future steps, the digitwise addition would become mismatched. Further, this process of addition is so convenient to state now because we spent so much effort earlier adding leading digits to Y, and thus do not have to worry about the case in which we have to add a new digit mid-computation.
- $(q_{restartarithmetic},\ 0)$, $(q_{restartarithmetic},\ 1)$, $(q_{restartarithmetic},\ \dot{0})$, $(q_{restartarithmetic},\ \dot{1}) \to$ themselves and $right$. This should continue until it reaches the end of Z, then restart the addition process.
- $(q_{restartarithmetic},\ \sqcup) \to (q_{beginarithmetic},\ \sqcup,\ left)$. Note that we do not need to add a bounce step here, since we are leaving this space blank.

This iteration should terminate once all of Z has been added to Y, at which point all of Z should be marked. At that point, $q_{beginarithmetic}$ will end up reading a $\circ$ since it goes to itself when it reads a marked digit. So we can let:

- $(q_{beginarithmetic},\ \circ) \to (q_{clearmarks0},\ \circ,\ left)$.

We have now modified Y such that it is equal to the sum of the original values of Y and Z. It suffices to check whether this Y and X are equal. To do this, we first remove all existing marks.

- $(q_{clearmarks0},\ 0)$ and $(q_{clearmarks0},\ \dot{0}) \to (q_{clearmarks0},\ 0,\ left)$. Similarly $(q_{clearmarks0},\ 1)$ and $(q_{clearmarks0},\ \dot{1})$ both go $\to (q_{clearmarks0},\ 0,\ left)$. The number on the state is present to count the number of circles this has passed; once it reaches the second $\circ$, it is necessarily at the beginning of the tape, because this process started at the cell before the third $\circ$.
- $(q_{clearmarks0},\ \circ) \to (q_{clearmarks1},\ \circ,\ left)$. $q_{clearmarks1}$ behaves identically to $q_{clearmarks0}$ on inputs other than $\circ$.
- $(q_{clearmarks1},\ \circ) \to (q_{gotoendY0},\ \circ,\ right)$. Now that we have cleared the tape of marks, we must return to the end of Y and begin the actual comparison process.
- $(q_{gotoendY0},\ 0)$, $(q_{gotoendY0},\ 1)$, $(q_{gotoendY0},\ \dot{0})$, $(q_{gotoendY0},\ \dot{1})$ to themselves and $right$. We will be reusing these states later, which is why we define their behavior on marked digits, even though as of now there are none on the tape.
- $(q_{gotoendY0},\ \circ) \to (q_{gotoendY1},\ \circ,\ right)$. As before $q_{gotoendY1}$ behaves identically to $q_{gotoendY0}$ on inputs other than $\circ$.
- $(q_{gotoendY1},\ \circ) \to (q_{begincomparison},\ \circ,\ left)$.

Now that the tape is clear of marks, we can safely use marks to actually compare X and Y. To do this, we look at the rightmost unmarked digit of the modified Y, remember it, and mark it. If it fails to corresponds with the rightmost unmarked digit of X, we reject the input. If it does correspond, we mark that digit of X and repeat this process. Eventually, if no rejection occurs, all digits of either X or Y will be marked. At this point, we can search both for unmarked digits. If all unmarked digits are 0, then they only differ by leading 0s and are therefore equal, and we accept the input. If there is an unmarked 1, the inputs differ and we reject the input.

- $(q_{begincomparison}, \dot{0})$ and $(q_{begincomparison}, \dot{1}) \to$ themselves and $left$.
- $(q_{begincomparison}, 0) \to (q_{gocheckXdigit0}, \dot{0})$ and $(q_{begincomparison}, 1) \to (q_{gocheckXdigit1}, \dot{1})$.
- $(q_{gocheckXdigit0}, 0)$, $(q_{gocheckXdigit0}, 1)$, $(q_{gocheckXdigit1}, 0)$, and $(q_{gocheckXdigit0}, 1) \to$ themselves and $left$.
- $(q_{gocheckXdigit0}, \circ) \to (q_{checkXdigit0}, \circ, left)$ and $(q_{gocheckXdigit1}, \circ) \to (q_{checkXdigit1}, \circ, left)$. Once the $\circ$ is read, the machine head is in X, and should look for an unmarked digit to compare to the stored digit in its state.
- $(q_{checkXdigit0}, \dot{0})$, $(q_{checkXdigit0}, \dot{1})$, $(q_{checkXdigit1}, \dot{0})$, $(q_{checkXdigit1}, \dot{1}) \to$ themselves and $left$.
- $(q_{checkXdigit0}, 1)$ and $(q_{checkXdigit1}, 0) \to (q_{reject})$. Detecting a difference in the corresponding digits implies the stored numbers are not equal, so we reject the input and are done.
- $(q_{checkXdigit0}, 0) \to (q_{gotoendY0}, \dot{0}, right)$ and $(q_{checkXdigit1}, 1) \to (q_{gotoendY0}, \dot{1}, right)$.

This iteration can end one of two ways. If Y runs out of unmarked digits first, $q_{begincomparison}$ will encounter a $\circ$, in which case we want to scan leftwards for any unmarked 1s, reject if we find any, and accept if none are found. So we have

- $(q_{begincomparison}, \circ) \to (q_{leftwardsfinalscan}, \circ, left)$.
- $(q_{leftwardsfinalscan}, 0)$, $(q_{leftwardsfinalscan}, \dot{0})$, and $(q_{leftwardsfinalscan}, dot1) \to$ themselves and $left$.
- $(q_{leftwardsfinalscan}, 1) \to (q_{reject})$.
- $(q_{leftwardsfinalscan}, \circ) \to (q_{accept})$ since reading a $\circ$ means we have confirmed all digits of X, and thus we are done.

If X runs out of digits first, one of the *checkXdigit* states will encounter the $\circ$ at the beginning of X. In this case, it should pass back into Y, rejecting if it encounters an unmarked 1 and accepting otherwise.

- $(q_{checkXdigit0}, \circ) \to (q_{rightwardsfinalscanstart}, \circ, right)$.
- $(q_{checkXdigit1}, \circ) \to (q_{reject})$.
- $(q_{rightwardsfinalscanstart}, 0)$, $(q_{rightwardsfinalscanstart}, 1)$, $(q_{rightwardsfinalscanstart}, \dot{0})$, and $(q_{rightwardsfinalscanstart}, \dot{1})$ *to* themselves and $right$.
- $(q_{rightwardsfinalscanstart}, \circ) \to (q_{rightwardsfinalscan}, \circ, right)$.
- $(q_{rightwardsfinalscan}, 0) \to (q_{rightwardsfinalscan}, 0, right)$.
- $(q_{rightwardsfinalscan}, 1) \to (q_{reject})$.
- $(q_{rightwardfinalscan}, \dot{0})$ and $(q_{rightwardfinalscan}, \dot{1}) \to (q_{accept})$. Since Y is marked right-to-left, once a single marked character is found going rightwards, there will be none more, which means the values are equal, and we're done. □

Now we present the proof that any Turing machine can be emulated by a Turing machine with $\Gamma = \{0, 1, \sqcup\}$.

Let L be such a language, T the Turing machine that decides it, and Γ be the collection of symbols it uses. We construct T' to behave as follows:

(1) First, using *replace* states, it moves the entire input two cells to the right. Then it checks to make sure the binary digits of the input are in the form prescribed by γ,

rejecting otherwise. Then it places two 1s at the start of the input to mark it: none of the sequences generated by γ have two consecutive 1s.

(2) Then, the machine iterates the following process: Go to the end of the input and parse left, ignoring sequences of the form (k many consecutive 0s, a single 1, $|\Gamma| - k$ many consecutive $\sqcup$s), until it reaches a sequence of the form created by γ. When it does so, parse to the left until it encounters a 1, counting how many consecutive 0s it sees. Call this quantity k. There are always only boundedly many 0s, so we can use the machine's states to remember them. Go back to the 1 that ends this sequence, and move everything to the right of it $|\Gamma| - k$ cells to the right, putting $\sqcup$ in the intervening spaces. Again, we do not need to use marks to perform this counting process, because it has boundedly many steps. Once it reaches two consecutive 1s, we have reached the start of the tape, so we are done with this step.[14] Now we go to the state of T' corresponding to the start state of T, which we will soon describe.

What did the process we just performed do? The original input to T' was a sequence of elements of $\gamma(\Gamma)$; each of which is a series of consecutive 0s followed by a 1. The process we performed separates the input into "blocks" of length $|\Gamma| + 1$ by adding blank symbols in between the distinct parts of the input. Now, the tape is broken into pieces of length $|\Gamma| + 1$, each of which corresponds to $\gamma(x)$ for some $x \in \Gamma$.

(3) Now, let us examine the transition function δ of T. Say, for example, in state q_i, it takes every symbol x to some corresponding state q_j, writing symbol $\Gamma_{(i,x)}$ on the tape and going to the left or right. We can construct an analogous transition function with analogous states for T'. When in state $q_{(i,0)}$, it will read either a 0 or 1. When it reads a 0 while in state $q_{(i,n)}$ with natural number n, it travels right and enters the state $q_{(i,n+1)}$. When encountering 1 in such a state, it travels left $n + 1$ cells, back to the start of that block. Note that each n corresponds to having read n 0s followed by a one, which is the image of some x under γ. Recall the behavior of the original machine T. T' now writes $\gamma(\Gamma_{(i,x)})$ on the tape, travels right until it reaches the next block, and enters $q_{(j,0)}$.

What did we just construct? Well, for each state of T, we created $|\Gamma| + 2$ states of T' corresponding to it with an index used to count 0s. The states parse right, counting 0s using the index, until it reads a 1, at which point its current index tells it what symbol it has just read. Then it writes down the binary notation of the symbol the original machine would have written, travels to the next $|\Gamma| + 1$-length block, and enters the 0 index of the state which the original machine would have entered.

This is why we separated the tape into blocks. Each block is a place to store a long sequence corresponding to a symbol used by T, and our new machine is just a copy of T that treats each block as a tape cell. We can do this because by definition, the original machine had finitely many states and finitely many language symbols, which means we can explicitly code into the states of T' what to do in every situation to imitate T.

[14] One may worry that since we are using $\sqcup$ for computation, the machine may think that it has reached the end of the input when it has not. We can prevent this the same way we have dealt with other problems: the process we are defining only ever writes boundedly many consecutive $\sqcup$, so when we want the machine to check whether it has reached the end of the input, we can have it travel a fixed but larger number of cells to the right. If any one those cells are nonblank, it is still in the input, and if all are blank, it truly has reached the end of the tape.

(4) The q_{accept} and q_{reject} states of T' relate to the other states exactly as defined in the previous step. T' reaches such a state by reading each sequence in its inputs and copying the original machine's reaction to it, meaning that it reaches these steps if and only if the original machine T accepts or rejects on the corresponding input. Thus the languages they decide are equivalent as we described. □

This exact same proof can be adapted to the more general case of computing functions, rather than merely deciding languages, with some appropriate cleaning of the tape.

One might also notice that in this proof, it did not seem like we "needed" all three symbols, $\{0, 1, \sqcup\}$. This is a correct intuition: with some carefulness, we can strengthen this result to not just binary, but *unary*, in which the input is simply a long string of 1s, and the machine uses 1 and $\sqcup$ to perform computation. This construction is sometimes preferred, but we neglect it due to the often strange behavior of unary. There are k^n distinct inputs of length n written in k symbols. For $k = 1$, this is a constant, meaning unary is exponentially inefficient at conveying information, and thus a terrible context in which to motivate K-complexity.

References

[LL18] Jack H Lutz and Neil Lutz. "Algorithmic information, plane Kakeya sets, and conditional dimension". In: *ACM Transactions on Computation Theory (TOCT)* 10.2 (2018), pp. 1–22.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Third. Cengage Learning, 2013, p. 168. ISBN: 978-1-133-18779-0.