Introduction

Our modern understanding of computation stems in large part from Alan Turing's formalization of the mathematical activity of following an effective method for computing a function. Thus the roots of computer science (at least as traditionally construed) are in this sense those of an essentially mathematical science. The science of Physics, on the other hand, ultimately aims to describe the characteristics of concrete systems as they exist in the natural world. It is thus a nontrivial question to ask whether, and how, physics can illuminate computer science and vice versa.

Indeed, the possible questions one may ask regarding the connections between computation and physics are many, varied, and multi-faceted. For the purposes of a philosophical investigation into these connections they can be usefully characterized as falling into two main categories. On the one hand, there are those questions related to the connections between computational and physical systems, and on the other hand there are those questions related to the connections between computational and physical theory in general. These two main categories can further be subdivided into two sub-categories each, which together comprise the four major parts of this volume:

- Interrelations between computational and physical systems
  - I. The computability of physical systems and physical systems as computers
  - II. The implementation of computation in physical systems
- Interrelations between computational and physical theory
  - III. Physical perspectives on computer science
  - IV. Computational perspectives on physical theory

In the remainder of this introductory chapter, we will summarize each of these parts and the particular contributions of this volume that fall under them. Before we do so, however, it will be useful to review some of the basic concepts which will generally be taken for granted in the rest of the book.

1 Computability Theory and the Church-Turing Thesis

Intuitively, computability theory concerns which tasks can be completed in principle by following a completely explicit set of instructions. These instructions must be definite, in the sense that they allow no procedural interpretation or flexibility, and self-contained, in the sense that they require no input other than what is provided in the description of the task itself. Such a set of instructions, called an *effective procedure*, hence demands no creativity of whoever (or whatever) executes it.

Effective procedures have found their greatest application in the mathematical domain, many of whose problems can be reduced to the computation of a function of natural numbers. A function $f : \mathbb{N}^k \to \mathbb{N}$ is said to be *effectively computable* when there is an effective procedure for calculating its value for any argument. For example, the familiar elementary arithmetic functions of addition and multiplication are clearly effectively computable, as are functions composed from them. Further, the effective computability of a mathematical decision problem, such as "Is n prime?", can be encoded into a function whose range is $\{0, 1\}$, corresponding with the "no" and "yes" answers.

To make this informal concept of effective computation formally tractable, myriad models for computation have been proposed, inspired variously from logic, arithmetic, and mechanics.[1] One might naturally expect that these different proposals, various as they are in their starting points, lead to formalizations of differing computational strength. So it is remarkable that they in fact determine extensionally the same class of functions as being computable.

The most important and influential of these proposals, on which we focus in this introduction, is that of the Turing machine (TM). A TM is a type of abstract state machine, consisting of the following components (an example of which is illustrated in Figure 0.1):

- An arbitrarily long tape, divided into sequential squares that can be blank or contain a mark.
- A read/write head, which sits atop a particular square, can read whether it contains a mark, and can perform the following actions: print a mark on the

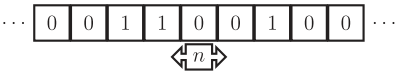

Figure 0.1 A representation of a Turing machine with read/write head in state n and tape entries "0" and "1" representing blank and marked squares, respectively

[1] Examples include representability in a formal system, the λ-calculus, recursive function theory, Markov algorithms, register machines, and Turing machines (what we focus on below). See Epstein and Carnielli (2008, ch. 8E) for brief descriptions and references to these various approaches.

square if it doesn't have one, erase the mark on the square if it has one, move one square to the right, and move one square to the left.

- A program, or finite set of instructions, for the read/write head, each of which has the following form:

  TM Instruction Form In state n, if the current square is [blank/marked], perform [action] and transition to state m. In abbreviated form: $(n, [0/1], [\text{action}], m)$.

The tape is the medium for the input and output to a proposed calculation by the machine, as well as for all the intermediate work required to transition between them. It typically encodes these in binary notation, for example with blank and marked squares representing the numerals "0" and "1," respectively. The read/write head performs the steps leading to the computation through its fixed set of actions. The program encodes an effective procedure for the TM to follow:

1. A TM begins in some pre-specified state at some pre-specified location on the tape.
2. The read/write head reads its current square, then performs the action (if any) specified by the program according to what's read and the current state.
3. It then transitions to another state, according to the program, whereupon step two is repeated.

A program need not have an action and state transition specified for every state and input from the current square. If it does not, then the read/write head halts, indicating the end of the computation, at which point the contents of the tape represent the computation's output.[2]

Suppose now that an encoding of inputs and outputs of natural numbers on the tape and a starting location for the printing head on the tape have been fixed. One then says that a function $f : \mathbb{N}^k \to \mathbb{N}$ is *Turing-computable* if and only if there is a TM that for all $(n_1, \ldots, n_k) \in \mathbb{N}^k$ eventually halts with output encoding $f(n_1, \ldots, n_k)$ when begun with input encoding $(n_1, \ldots, n_k)$.

Because the TM's tape represents the inputs and outputs of the function it computes and the functionality of the read/write head is fixed, the real source of variability amongst TMs comes from the program. TMs may thus be enumerated by their programs, which can be represented by sets of ordered quadruples of the above specified TM Instruction Form. For example, the set $\{(1, 0, 1, 1), (1, 1, \rightarrow, 1)\}$ defines a program (i.e., a TM) with only one state, in

[2] Despite the physically evocative story involving "components" and so on, a concrete mechanism for implementing or constructing an actual TM is neither provided nor necessary. This is the sense in which the TM is an *abstract* machine providing a *mathematical* definition of computation, rather than a schematic for a physical machine providing an empirical account of computation; cf. Section 4 of this Introduction.

which it writes a mark if the current square is empty and otherwise moves to the right. Since every Turing-computable function must be computable by some TM, it follows immediately that the Turing-computable functions are enumerable. But because there are uncountably many functions of natural numbers, there must be functions that are not Turing-computable – infinitely more, in fact, than those that are.

Two further remarkable facts build upon such an enumeration.[3] First is the existence of *universal* TMs, ones that can simulate the computation of any other TM. In other words, there exist TMs such that, when given input encoding $(m, n_1, \ldots, n_k)$, they eventually halt with output $f_m(n_1, \ldots, n_k)$, where f_m is the Turing-computable function computed by the mth TM. Second is the specification of concrete non-Turing-computable functions. Most famous of these is the halting function $h : \mathbb{N}^2 \to \mathbb{N}$, which is equal to 1 if TM m halts on input n, and is equal to 2 otherwise.

The theory of computability can be developed much further,[4] but to close this section we circle back to the original motivation for TMs: does Turing-computability adequately formalize the concept of effective computability? Clearly every Turing-computable function is effectively computable, for the action of a TM that computes such a function is given by an effective procedure. The statement that the converse is also true is known as either *Turing's Thesis* or the *Church-Turing Thesis*.

Church-Turing Thesis (CTT) Every effectively computable function of natural numbers is Turing computable.

The truth of the CTT would imply that one can identify or replace the extension of the informal concept of effective computability with that of the formal concept of Turing-computability, thereby establishing a completely adequate explication (cf. Carnap 1947, sec. 2; Carnap 1950b, ch. 1) of the former. There is a large literature on the status and interpretation of the CTT,[5] but it is fair to say that it is widely accepted among computer scientists and beyond. That said, the theory of computability and the CTT only make claims about what is possible *in principle* to compute, given the idealizations of arbitrarily large temporal, spatial, and material resources – computing steps, tape squares, and the incorruptible functioning of Turing machinery – that abstract away from their actual abundance. When an accounting of these resources is brought to bear, as in the next section of this Introduction, one can distinguish not just between computable and non-computable functions, but, among the computable ones, those of various degrees of difficulty.

[3] For more on TMs, see Barker-Plummer (2016) and references cited therein.
[4] See, for instance, Immerman (2016) and references cited therein.
[5] See, for instance, Copeland (2015) and references cited therein.

2 Computational Complexity Theory

In computational complexity theory, computational problems are classified based on their resource costs, i.e., those in time and space. We will focus on time, which is the more important measure. Arguably the most basic distinction within the theory is that between those decision problems (i.e., yes-or-no questions) that are "easy" (a.k.a. "feasible," "efficiently solvable," "tractable," etc.) and those that are not (i.e., "hard"). According to the Cobham-Edmonds thesis (Dean 2016b), a decision problem is easy if it is solvable in "polynomial time," i.e., if it can be solved in a number of steps bounded by a polynomial function of its input size, n. Problems so solvable on a deterministic Turing machine (DTM) comprise the complexity class P.

Formally, one can conceive of a decision problem as one of determining whether a given string x of length n is in the "language" L. For example, determining whether x is prime amounts to determining whether it is in the language {10, 11, 101, 111, 1011, 1101, 10001, 10011, . . .} (the set of binary representations of prime numbers). Now, call a language L a member of the class DTIME($T(n)$) if and only if there is a DTM for deciding membership in L whose running time, $t(n)$, is "on the order of $T(n)$," or in symbols: $O(T(n))$. Here, $T(n)$ represents an upper bound for the growth rate of $t(n)$ in the sense that, by definition, $t(n)$ is $O(T(n))$ if for every sufficiently large n, $t(n) \leq k \cdot T(n)$ for some constant k.[6] We can now formally characterize P (Arora and Barak 2009, p. 25) as:

$$\mathrm{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k). \tag{0.1}$$

A *nondeterministic Turing machine* (NTM) is such that it may "choose," when in a given state, which one of a set of possible successor states to transition to; see Figure 0.2. It is said to accept a string x if and only if there exists a path through its state space that, given x, leads to an accepting state. It rejects x otherwise. NTIME($T(n)$) is now defined, analogously to DTIME($T(n)$), as the set of languages for which an NTM exists to decide, in $O(T(n))$ steps, whether a given string x of length n is in L. The class "NP" is defined as:[7]

$$\mathrm{NP} =_{df} \bigcup_{k \geq 1} \mathrm{NTIME}(n^k). \tag{0.2}$$

[6] By "for every sufficiently large n" it is meant that there exists some $n_0 \geq 1$ such that $t(n) \leq k \cdot T(n)$ whenever $n \geq n_0$.

[7] Equivalently (Arora and Barak 2009, p. 42), NP is the set of languages for which one can construct a polynomial-time *deterministic* TM to verify, for any x, that $x \in L$, given a polynomial-length string u (called a "certificate" for x).

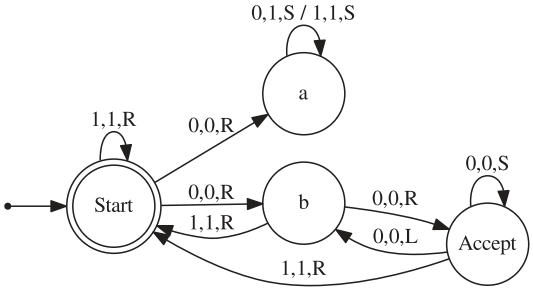

Figure 0.2 This NTM accepts binary strings ending in "00," since for a given such x, there exists a series of transitions which end in "Accept." But this is not guaranteed. The machine is guaranteed, on the other hand, to reject any string not ending in "00." An edge from s_1 to s_2 labeled α, β, P is read as: In state s_1, read α from the tape, overwrite α with β, move the read/write head to P (L = to the left, R = to the right, S = same), and finally transition to state s_2

Exactly how an NTM "chooses" to follow one path rather than another is not defined. In a *probabilistic Turing machine* (PTM), in contrast, we associate a particular probability with each possible transition. We can then define the class BPP (bounded-error probabilistic polynomial time) as the class of languages such that there exists a polynomial-time PTM which, on any given run, will correctly determine whether a string x is in the language L with probability $\geq 2/3$.[8]

It has been conjectured that any language L decidable under a given "reasonable" (i.e., physically realizable) machine model $\mathfrak{M}$ is "efficiently simulable" by a PTM in the sense that a PTM to decide L exists which requires at most a polynomial number of extra time steps compared to a machine of type $\mathfrak{M}$. This is known as the "strong" or "extended" Church-Turing thesis (ECT).[9,10] Over the last three decades, however, evidence has been mounting against the ECT, primarily as a result of the advent of quantum computing (Aaronson 2013, chs. 10, 15), which we will discuss in more detail in the next section.

[8] The particular threshold probability 2/3 is inessential. Any probability $p_{min} \geq 1/2 + n^{-k}$, with k a constant, will yield the same class (Arora and Barak 2009, p. 132).

[9] For further discussion of the ECT and related issues, see Dean (2016a,b,c).

[10] ECT is sometimes defined with respect to the TM rather than the PTM model, for there has been mounting evidence that P = BPP (Arora and Barak 2009). For our purposes the choice of TM or PTM is inessential; a TM is a special case of a PTM for which transition probabilities are always either 0 or 1. Moreover, defining ECT with respect to the PTM model is convenient when comparing classical computation in general with quantum computation, which is probabilistic.

3 Quantum Computing

The best-known classical algorithm for factoring arbitrary integers, the number field sieve (Lenstra et al. 1990), requires $O(2^{(\log N)^{1/3}})$ steps to factor a given integer N. Shor's *quantum* factoring algorithm requires only a number of steps that is polynomial in $\log N$ – an exponential speedup over the number field sieve. This and other quantum algorithms provide evidence that the ECT is false, for they seem to show that BQP, the class of languages probabilistically decidable by a quantum computer in polynomial time, is strictly larger than BPP. Note, however, that although the evidence furnished by Shor's algorithm is strong, it is still an open question whether factoring is in BPP.[11]

The state of a classical digital computer, whether deterministic or probabilistic, is describable as a sequence of bits. A bit can be directly instantiated by any two-level classical physical system, such as a circuit that can be open or closed. In a *quantum* computer, the basic unit of representation is not the bit but the qubit. To directly instantiate it, one uses a two-level quantum system such as an electron (specifically: its spin). Like a bit, a qubit can be "on": $|0\rangle$, or "off": $|1\rangle$. In general, however, a qubit's state can be expressed as a normalized linear superposition:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{0.3}$$

where the complex "amplitudes" α and β satisfy the normalization condition: $|\alpha|^2 + |\beta|^2 = \alpha\bar{\alpha} + \beta\bar{\beta} = 1$, with $\bar{c}$ the complex conjugate of c. We refer to $|\psi\rangle$ as the "state vector" for the qubit.

Unlike a bit, not all states of a qubit can be observed directly. In particular, one never observes a qubit in a linear superposition with respect to a particular measurement basis. For example, a "computational basis" measurement – "$|0\rangle$ or $|1\rangle$?" – will never reveal a qubit state of the form of Eq. (0.3), aside from the trivial case where one of α or β is 0. In general, given the initial state in Eq. (0.3), such a measurement on a qubit will find it in the state $|0\rangle$ with probability $|\alpha|^2$ and in state $|1\rangle$ with probability $|\beta|^2$.

Quantum computers can efficiently simulate classical probabilistic computers, since it is "easy" in a complexity-theoretic sense to simulate a fair classical coin toss. For example, one can instantiate the transition Q, defined as:

$$Q|0\rangle \rightarrow \frac{i}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, \qquad Q|1\rangle \rightarrow \frac{1}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle,$$

and then measure in the computational basis.

[11] For further discussion, see Cuffaro (in press).

Unlike classical bits, qubits can sometimes exhibit "interference effects." For example, upon applying the "Q-gate" twice to a qubit in the initial state $|0\rangle$, "destructive" and "constructive" interference is exhibited between the complex amplitudes associated with the $|0\rangle$ and $|1\rangle$ components of the state vector, respectively:

$$|0\rangle \xrightarrow{Q} \left(\frac{i}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \xrightarrow{Q} \left(-\frac{1}{2}|0\rangle + \frac{i}{2}|1\rangle + \frac{1}{2}|0\rangle + \frac{i}{2}|1\rangle\right) = i|1\rangle.$$

A computational basis measurement on the qubit will now yield $|1\rangle$ with certainty. This ability to exhibit interference effects is held by some to be the key to understanding the source of the power of quantum computers (Fortnow 2003; Aaronson 2013).

The combined state of two or more qubits is said to be separable if it can be expressed as a product state:

$$|\alpha\rangle \otimes |\beta\rangle \otimes |\gamma\rangle \ldots.$$

The state

$$\begin{aligned}|\psi\rangle &= |0\rangle \otimes |0\rangle + |0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle \\ &= (|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle)\end{aligned}$$

is an example. Not all states of more than one qubit are separable states. The following is an entangled state; it cannot be expressed as a product state:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

The ability of qubits to form entangled states when combined is another oft-cited source of the power of quantum computers (Steane 2003). Entanglement has been shown to be necessary for achieving quantum speedup when using pure states (Jozsa and Linden 2003). However in the same paper, Jozsa and Linden argue that it may not be a sufficient resource. For an in-depth discussion, see Cuffaro (2017).

Other purported sources of quantum speedup include: massive quantum parallelism achievable through the ability of qubits to realize superposition states (Pitowsky 2002; Duwell, this volume); the same but with an additional ontological posit of many computational worlds (Hewitt-Horsman 2009; Cuffaro 2012); quantum contextuality (Howard et al. 2014); and the structure of quantum logic (Bub 2010). For an overview and further discussion, see Hagar and Cuffaro (2017).

4 Computational Implementation and the Physical Church-Turing Theses

The previous sections of this Introduction concerned what could be computed and how efficiently, largely abstracting from most of the physical, mechanical, and engineering details that would be necessary to describe adequately a concrete computer executing a concrete computation (but noting the possible differences between classical and quantum for complexity theory). This abstraction of computability and computational complexity theory is perfectly unproblematic when the latter are considered as branches of mathematics. But their application to putative concrete computers and computations demands an account of how their abstract objects adequately represent physical objects and processes, or how their descriptions of computation are adequate abstractions from concrete ones. What, in other words, would it take for a physical system to implement a computation?

There is no consensus about the correct account of computational implementation, but it will be helpful for the remainder to keep several different proposals in mind.[12] The *simple mapping* account states, roughly, that a physical system performs a computation when there is a mapping from the sequential states of the system to the computational states of a computational model (say, the state and tape contents of a TM) such that physical state transitions get mapped to computational state transitions. This account is very liberal, in that it designates multitudinous physical processes as implementing multitudinous computations. It is thus often associated with the thesis of (unlimited) *pancomputationalism*, that (nearly) all physical processes implement all (or many non-equivalent) computations, in some sense.[13] Because many take (unlimited) pancomputationalism to be implausible, many other accounts of computational implementation add extra conditions to the mappings of the simple mapping account to make computation less abundant. Causal, counterfactual, and dispositional accounts require that the state transitions support various modal conditions. Semantic and syntactic accounts take seriously the idea of computation as manipulation of meaningful symbols, requiring respectively that the mappings be representational, according to some account of proper representation, or syntactical, according to an account of what it means for states and changes thereof to be syntactically structured. Mechanistic accounts demand that the physical system or process implementing a computation does so in terms of a functional mechanism, an organization of the components of the system suited to the task of manipulating computational vehicles, those components whose states are mapped to computational states.

[12] See Piccinini (2017, sec. 2) for a more thorough review of proposals for computational implementation.
[13] See Piccinini (2017, sec. 3) for more on the different types of pancomputationalism.

Regardless of how the issue of computational implementation is settled, it raises the further question of an analog of the CTT for physical computations. Recall from section 1 that the CTT subsumed the extension of an informal concept – effective computability – under a formal one – Turing computability. Effective computability, though vague, concerns in some idealized sense what can be in principle computed by a human agent aided only with simple memory aids such as paper and pencil. Physical computability, by contrast, concerns what can be computed by any physical process made eligible by one of the above accounts. So a physical version of the CTT would subsume a presumably wider set of physical processes under Turing computation, and the converse of a physical version should easily follow from an argument similar to that for the converse of the CTT.

Several versions of a physical CTT have been proposed. Following Piccinini (2011, 2015), it is helpful to distinguish between two classes of physical CTT:

Modest Physical CTT Any function of natural numbers that is physically computable is Turing computable.

Bold Physical CTT Any physical process is Turing computable.

The modest version, like the CTT itself, focuses on the computation of the values of numerical functions. Sometimes Gandy (1980) is interpreted as having advanced such a thesis:

Gandy's Thesis M Any function of natural numbers computable by a discrete deterministic mechanical assembly (DDMA) is Turing computable.

A DDMA is any physical device of which an adequate theoretical description uses discrete dynamics for finitely many parts of bounded complexity that affect each other only locally and deterministically. Gandy (1980) then proves that under a certain formalization of DDMAs, Thesis M follows. This thesis is physical in the sense that DDMAs are intended to be models for arbitrary machines that humans might construct to aid them in computations. However, unless one has an essentially anthropocentric account of computational implementation, it is less plausible that physical computations are exhausted by such machines. Accordingly, Thesis M sits conceptually between the CTT and the modest physical CTT.

The bold version of the physical CTT requires a bit of interpretation: what does it mean for a process to be computable? Typically, this means that an adequate theoretical description of the physical process can be simulated by a TM, in the sense that there is an injective map from the physical states of the system undergoing dynamical evolution to the computational states of a TM. A version of this thesis has been advocated by Deutsch (1985):

Deutsch's Principle Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means.