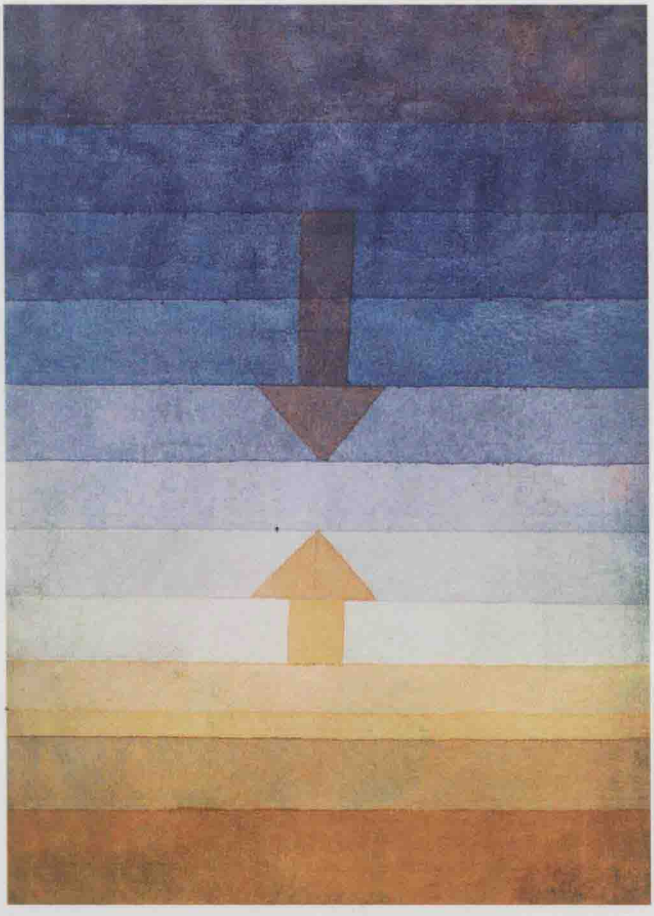
OXFORD

PHYSICAL COMPUTATION

A Mechanistic Account



GUALTIERO PICCININI

Physical Computation

A Mechanistic Account

Gualtiero Piccinini

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

First Edition published in 2015

Impression: 1

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data

Data available

Library of Congress Control Number: 2014957194

ISBN 978–0–19–965885–5

Printed and bound by
CPI Group (UK) Ltd, Croydon, CR0 4YY

Physical Computation

Acknowledgements

I encountered the debate about the foundations of cognitive science when, as an undergraduate at the University of Turin, I took an introductory course in cognitive psychology taught by Bruno Bara. Those early wonderings eventually led to my Ph.D. dissertation at the University of Pittsburgh, and then to a stream of papers on computation and related topics. This book organizes, streamlines, revises, and systematizes most of what I've thought and published about concrete computation. (It leaves out my work on computational theories of cognition.)

I could not have gotten where I am without the help of my teachers, beginning with my parents and continuing all the way to my college and graduate school teachers. In college, the following were especially influential: in philosophy, Elisabetta Galeotti, Diego Marconi, Gianni Vattimo, and Carlo Augusto Viano; in psychology, Bruno Bara and Giuliano Geminiani. In graduate school, the following played important roles in my thinking about computation: my advisor Peter Machamer as well as Nuel Belnap, Robert Daley (Computability Theory), Bard Ermentrout (Computational Neuroscience), John Earman, Clark Glymour, Paul Griffiths, Rick Grush, Ken Manders, John Norton, and Merrilee Salmon. Other important teachers during graduate school were Robert Brandom, Joe Camp, Adolf Grünbaum, Susie Johnson (Theory of Mind), Pat Levitt (Systems Neuroscience), Jay McClelland (Cognitive Neuroscience), John McDowell, Ted McGuire, Robert Olby, Carl Olson (Cognitive Neuroscience), Fritz Ringer, Laura Ruetsche, and Richmond Thomason.

Many people's work shaped mine. On computation, the classic work of Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing was paramount. On neural networks and their connection to digital computers, I learned the most from Warren McCulloch, Walter Pitts, John von Neumann, and Norbert Wiener. On analog computers, the work of Claude Shannon, Marian Pour-El, and Lee Rubel was crucial. On artificial intelligence and computational cognitive science, Allen Newell and Herbert Simon had the largest impact. The canonical textbook on computer organization and design by David Patterson and John Hennessy confirmed my early hunch that computational explanation is mechanistic and helped me articulate it. On the philosophy of computation, I found most helpful the works of Jack Copeland, Robert Cummins, Daniel Dennett, Frances Egan, Jerry Fodor, Justin Garson, Gilbert Harman, John Haugeland, Zenon Pylyshyn, Oron Shagrir, Wilfried Sieg, and Stephen Stich. On information, Fred Drestke and Claude Shannon were most significant. On functions, Christopher Boorse and Ruth Millikan made the biggest difference. On mechanisms, I was most influenced by Carl Craver, Lindley Darden, Peter Machamer, and William Wimsatt. On general metaphysics, John Heil and the NEH Summer Seminar he directed helped me an enormous amount.

The work that goes into this book has benefited from so many interactions with so many people over such a long time—including audiences at many talks—that I cannot hope to remember everyone who played a role. I apologize in advance to those I forget. Here are those I remember as most helpful: Darren Abramson, Fred Adams, Neal Anderson, Ken Aizawa, Sonya Bahar, Mark Balaguer, Bill Bechtel, Kenny Boyce, Dave Chalmers, Mark Collier, Carl Craver, Jack Copeland, Martin Davis, Daniel Dennett, Michael Dickson, Eli Dresner, Frances Egan, Chris Eliasmith, Ilke Ercan, Carrie Figdor, Jerry Fodor, Florent Franchette, Nir Fresco, Justin Garson, Carl Gillett, Robert Gordon, Sabrina Haimovici, Gilbert Harman, John Heil, Alberto Hernández, David M. Kaplan, Saul Kripke, Fred Kroon, Bill Lycan, Corey Maley, Marcin Milkowski, Jonathan Mills, Alex Morgan, Toby Ord, Anya Plutynski, Tom Polger, Hilary Putnam, Michael Rabin, Michael Rescorla, Brendan Ritchie, Brad Rives, Martin Roth, Anna-Mari Rusanen, Dan Ryder, Andrea Scarantino, Matthias Scheutz, Susan Schneider, Sam Scott, Larry Shapiro, Rebecca Skloot, Mark Sprevak, Oron Shagrir, Stuart Shapiro, Doron Swade, Wilfried Sieg, Eric Thomson, Brandon Towl, Ray Turner, Dan Weiskopf, and Julie Yoo.

Special thanks to Mark Sprevak, the members of the reading group he ran on a previous version of the manuscript, and an anonymous referee for OUP for their extremely helpful feedback. Thanks to Peter Momtchiloff, my editor at OUP, for his help, patience, and support.

Chapters 1–4 are indebted to G. Piccinini, "Computation in Physical Systems," *The Stanford Encyclopedia of Philosophy* (Fall 2010 Edition), Edward N. Zalta (ed.), URL = http://plato.stanford.edu/archives/fall2010/entries/computation-physicalsystems/

Chapter 1 is indebted to G. Piccinini, "Computing Mechanisms," *Philosophy of Science*, 74.4 (2007), 501–26.

Chapter 3 is indebted to G. Piccinini, "Computation without Representation," *Philosophical Studies*, 137.2 (2008), 205–41 and G. Piccinini, "Functionalism, Computationalism, and Mental Contents," *Canadian Journal of Philosophy*, 34.3 (2004), 375–410.

Chapter 4 is indebted to G. Piccinini, "Computational Modeling vs. Computational Explanation: Is Everything a Turing Machine, and Does It Matter to the Philosophy of Mind?" *Australasian Journal of Philosophy*, 85.1 (2007), 93–115.

Chapter 5 is indebted to G. Piccinini, "Functionalism, Computationalism, and Mental States," *Studies in the History and Philosophy of Science*, 35.4 (2004), 811–33 and G. Piccinini and C. Craver, "Integrating Psychology and Neuroscience: Functional Analyses as Mechanism Sketches," *Synthese*, 183.3 (2011), 283–311.

Chapter 6 is indebted to C. Maley and G. Piccinini, "A Unified Mechanistic Account of Teleological Functions for Psychology and Neuroscience," forthcoming in David Kaplan, ed., *Integrating Psychology and Neuroscience: Prospects and Problems*. Oxford: Oxford University Press.

Chapter 7 is indebted to G. Piccinini, "Computing Mechanisms," *Philosophy of Science*, 74.4 (2007), 501–26 and G. Piccinini, "Computation without Representation," *Philosophical Studies*, 137.2 (2008), 205–41.

Chapters 10–13 are indebted to G. Piccinini, "Computers," *Pacific Philosophical Quarterly*, 89.1 (2008), 32–73.

Chapter 13 is indebted to G. Piccinini, "Some Neural Networks Compute, Others Don't," *Neural Networks*, 21.2–3 (2008), 311–21.

Chapter 14 is indebted to G. Piccinini and A. Scarantino, "Information Processing, Computation, and Cognition," *Journal of Biological Physics*, 37.1 (2011), 1–38.

Chapters 15 and 16 are indebted to "The Physical Church-Turing Thesis: Modest or Bold?" *The British Journal for the Philosophy of Science*, 62.4 (2011), 733–69.

My co-authors for some of the above articles—Carl Craver, Corey Maley, and Andrea Scarantino—helped me develop important aspects of the view I defend and I am deeply grateful to them.

The articles from which the book borrows were usually refereed anonymously. Thanks to the many anonymous referees for their helpful feedback.

Thanks to my research assistant, Frank Faries, especially for drawing the figures for Chapters 8 and 9.

This material is based on work supported by the College of Arts and Sciences at the University of Missouri—St. Louis, the Charles Babbage Institute, the Institute for Advanced Studies at the Hebrew University of Jerusalem, the National Endowment for the Humanities, the National Science Foundation under grants no. SES-0216981 and SES-0924527, the University of Missouri, the University of Missouri—St. Louis, the Mellon Foundation, and the Regione Sardegna. Any opinions, findings, conclusions, and recommendations expressed in this book are those of the author and do not necessarily reflect the views of these funding institutions.

Deep thanks to my friends, partners, and family—my parents, sisters, brothers-in-law, and nieces—for their love and support, especially during my most challenging times. Thanks to my daughters—Violet, Brie, and Martine—for bringing so much meaning and joy to my life.

Contents

Acknowledgements vii

Introduction 1

1. Towards an Account of Physical Computation 4
2. Mapping Accounts 16
3. Semantic Accounts 26
4. Pancomputationalism 51
5. From Functional Analysis to Mechanistic Explanation 74
6. The Ontology of Functional Mechanisms 100
7. The Mechanistic Account 118
8. Primitive Components of Computing Mechanisms 152
9. Complex Components of Computing Mechanisms 162
10. Digital Calculators 176
11. Digital Computers 181
12. Analog Computers 198
13. Parallel Computers and Neural Networks 206
14. Information Processing 225
15. The Bold Physical Church-Turing Thesis 244
16. The Modest Physical Church-Turing Thesis 263

Epilogue: The Nature of Computation 274

Appendix: Computability 277
Bibliography 287
Index 307

Introduction

This book is about the nature of concrete computation—the physical systems that perform computations and the computations they perform. I argue that concrete computing systems are a kind of functional mechanism. A functional mechanism is a system of component parts with causal powers that are organized to perform a function. Computing mechanisms are different from *non*-computing mechanisms because they have a special function: to manipulate vehicles based solely on differences between different portions of the vehicles in accordance with a rule that is defined over the vehicles and, possibly, certain internal states of the mechanism. I call this the *mechanistic account* of computation.

When I began articulating and presenting the mechanistic account of computation to philosophical audiences over ten years ago, I often encountered one of two dismissive responses. *Response one*: your view is obvious, well known, and uncontroversial—utterly dull. *Response two*: your view is counterintuitive, implausible, and untenable—totally worthless. These were not the only responses. Plenty of people engaged the substance of the mechanistic account of computation and discussed its pros and cons. But these radical responses were sufficiently common that they deserve to be addressed upfront.

If the mechanistic account elicited either one of these responses but not the other, perhaps the mechanistic account would be at fault. But the presence of both responses is encouraging because they cancel each other out, as it were. Those who respond dismissively appear to be unaware that the opposite dismissive response is equally common. If they knew this, presumably they would tone it down. For although reasonable people may disagree about whether a view is true or false, it is unreasonable to disagree on whether something is *obviously* true or *obviously* false. If it's so obvious, how can there be equally informed people who think the *opposite* is obvious?

The first dismissive response—that the mechanistic account is so obvious that it's dull—seems to be motivated by something like the following reasoning. For the sake of the argument, let's assume along with many philosophers that computation is a kind of symbol manipulation. There is an important distinction between the syntax of symbols (and, more generally, their formal properties) and their semantics. To a first approximation, syntactic (more generally, formal) properties are those that determine whether a symbolic structure is well formed—they make the difference

between 'the puzzle is solvable' and 'puzzle the is solvable'; semantic properties are those that determine what symbols mean—they make the difference between 'i vitelli dei romani sono belli' in most languages, where it means nothing; in Latin, where it means *go, Vitellus, at the Roman god's war cry*; and in Italian, where it means *the calves of the Romans are good-looking*. Most people find it intuitively compelling that computations operate on symbols based on their formal or syntactic properties alone and not at all based on their semantic properties. Furthermore, many philosophers assimilate computational explanation and functional analysis: computational states are often said to be individuated by their functional relations to other computational states, inputs, and outputs. Therefore, computational states and processes are individuated functionally, i.e., formally or syntactically. Saying that computation is mechanistic, as my account does, is just a relabeling of this standard view. Therefore, the mechanistic account of computation is nothing new. Something like this reasoning is behind the first dismissive response. It is deceptively persuasive but, alas, it goes way too fast.

A first problem is that physical systems don't wear their syntactic (or formal) properties on their sleeves. If the mechanistic account were based on syntactic properties, it should begin with an account of syntactic properties that does not presuppose the notion of computation. I don't know of any such account, and fortunately I don't need one. For the mechanistic account of computation is painstakingly built by specifying which properties of which mechanisms are computational, without ever invoking the notion of syntax (or formal property). Thus, the mechanistic account may provide ingredients for an account of syntax—not vice versa (Chapter 3, Section 4).

A second problem is the implicit assimilation of functional analysis and computational explanation, which is pervasive in the literature. I reject such an assimilation and argue that functional analysis provides a partial sketch of a mechanism (Chapter 5), defend a teleological account of functional mechanisms (Chapter 6), and argue that computational explanation is a specific kind of mechanistic explanation (Chapter 7).

An additional issue is that computations are often individuated semantically—in terms of functions from what is denoted by their inputs to what is denoted by their outputs. And philosophers interested in computation are often interested in how computation can explain cognition, which is usually assumed to deal in representations. After all, cognitive states and processes are typically individuated at least in part by their semantic content. Thus, many philosophers interested in computation believe that computational states and processes are individuated by their content in such a way that at least part of their essence is semantic. I call this the *semantic account* of computation. Therein lies the motivation for the second dismissive response: since computation is essentially semantic and the mechanistic account of computation denies this, the mechanistic account is obviously and horribly wrong.

But the semantic account of computation has its own problems. For starters, the notion of semantic property is even more obscure and more in need of naturalistic explication than that of syntactic property. In addition, I argue that individuating computations semantically always presupposes their non-semantic individuation, and that some computations are individuated purely non-semantically. Therefore, contrary to the second dismissive response, computation does not presuppose representation (Chapter 3).

But if we reject the view that computation presupposes representation, we risk falling into the view that everything performs computations—pancomputationalism (Chapter 4). This is not only counterintuitive—it also risks undermining the foundations of computer science and cognitive science. It is also a surprisingly popular view. Yet, I argue that pancomputationalism is misguided and we can avoid it by a judicious use of mechanistic explanation (Chapter 4).

The mechanistic account begins by adapting a mechanistic framework from the philosophy of science. This gives us identity conditions for mechanisms in terms of their components, their functions, and their organization, without invoking the notion of computation. To this general framework, a mechanistic account of computation must add criteria for what counts as computationally relevant mechanistic properties. I do this by adapting the notion of a string of letters, taken from logic and computability theory, and generalizing it to the notion of a system of vehicles that are defined solely based on differences between different portions of the vehicles. Any system whose function is to manipulate such vehicles in accordance with a rule, where the rule is defined in terms of the vehicles themselves, is a computing system. I explain how a system of appropriate vehicles can be found in the natural (concrete) world, yielding a robust (nontrivial) notion of computation (Chapter 7).

After that, I develop this general mechanistic account by explicating specific computing systems and their properties in mechanistic terms. I explicate the notion of primitive computing components (Chapter 8), complex computing components (Chapter 9), digital calculators (Chapter 10), digital computers (Chapter 11), analog computers (Chapter 12), and neural networks (Chapter 13). After that, I return to semantic properties under the guise of information processing (in several senses of the term); I argue that processing information is a form of computation but computation need not be a form of information processing (Chapter 14). I conclude the book with the limits of physical computation. Once the relevant question is clarified (Chapter 15), the evidence suggests that any function that is physically computable is computable by Turing machines (Chapter 16).

1

Towards an Account of Physical Computation

1. Abstract Computation and Concrete Computation

Computation may be studied mathematically by formally defining computing systems, such as algorithms and Turing machines, and proving theorems about their properties. The mathematical theory of computation is a well-established branch of mathematics. It studies which functions defined over a denumerable domain, such as the natural numbers or strings of letters from a finite alphabet, are computable by algorithm or by some restricted class of computational systems. It also studies how much time or space (i.e., memory) it takes for a computational system to compute certain functions, without worrying much about the particular units of time involved or how the memory cells are physically implemented.

By contrast, most uses of computation in science and everyday life involve *concrete* computation: computation in concrete physical systems such as computers and brains. Concrete computation is closely related to mathematical computation: we speak of physical systems as running an algorithm or as implementing a Turing machine, for example. But the relationship between concrete computation and mathematical computation is not part of the mathematical theory of computation per se and requires further investigation. This book is about concrete computation. We will see in due course that questions about concrete computation are not neatly separable from mathematical results about computation. The following mathematical results are especially crucial to our subsequent investigation.

The most important notion of computation is that of digital computation, which Alan Turing, Kurt Gödel, Alonzo Church, Emil Post, and Stephen Kleene formalized in the 1930s (see the Appendix for a sketch of the most relevant background and results). Their work investigated the foundations of mathematics. One crucial question was whether first order logic is decidable—whether there is an algorithm that determines whether any given first order logical formula is a theorem.

Turing (1936–7) and Church (1936) proved that the answer is negative: there is no such algorithm. To show this, they offered precise characterizations of the informal notion of algorithmically computable function. Turing did so in terms of so-called Turing machines—devices that manipulate discrete symbols written on a tape in

accordance with finitely many instructions. Other logicians did the same thing—they formalized the notion of algorithmically computable function—in terms of other notions, such as λ-definable functions and general recursive functions.

To their surprise, all such notions turned out to be extensionally equivalent, that is, any function computable within any of these formalisms is computable within any of the others. They took this as evidence that their quest for a precise definition of 'algorithm,' or 'effective procedure,' or 'algorithmically computable function,' had been successful. They had found a precise, mathematically defined counterpart to the informal notion of computation by algorithm—a mathematical notion that could be used to study in a rigorous way which functions can and cannot be computed by algorithm, and therefore which functions can and cannot be computed by machines that follow algorithms. The resulting view—that Turing machines and other equivalent formalisms capture the informal notion of algorithm—is now known as the *Church-Turing thesis* (more on this in Chapter 15). It provides the foundation for the mathematical theory of computation as well as mainstream computer science.

The theoretical significance of Turing et al.'s formalization of computation can hardly be overstated. As Gödel pointed out (in a lecture following one by Tarski):

> Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness (or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen (Gödel 1946, 84).

A standard Turing machine computes only one function. Turing also showed that there are *universal* Turing machines—machines that can compute any function computable by any other Turing machine. Universal machines do this by executing instructions that encode the behavior of the machine they simulate. Assuming the Church-Turing thesis, universal Turing machines can compute any function computable by algorithm. This result is significant for computer science: you don't need to build different computers for different functions; one universal computer will suffice to compute any computable function. Modern digital computers approximate universal machines in Turing's sense: digital computers can compute any function computable by algorithm for as long as they have time and memory. (Strictly speaking, a universal machine has an unbounded memory, whereas digital computer memories can be extended but not indefinitely, so they are not quite unbounded in the same way.)

The above result should not be confused with the common claim that computers can compute *anything*. Nothing could be further from the truth: another important result of computability theory is that most functions are not computable by Turing machines (thus, by digital computers). Turing machines compute functions defined over denumerable domains, such as strings of letters from a finite alphabet. There are uncountably many such functions. But there are only countably many Turing

machines; you can enumerate Turing machines by enumerating all lists of instructions. Since an uncountable infinity is much larger than a countable one, it follows that Turing machines (and hence digital computers) can compute only a tiny portion of all functions (over denumerable domains, such as natural numbers or strings of letters).

Turing machines and most modern computers are known as (classical) *digital* computers, that is, computers that manipulate strings of discrete, unambiguously distinguishable states. Digital computers are sometimes contrasted with *analog* computers, that is, machines that manipulate continuous variables. Continuous variables are variables that can change their value continuously over time while taking any value within a certain interval. Analog computers are used primarily to solve certain systems of differential equations (Pour-El 1974; Rubel 1993).

Classical digital computers may also be contrasted with *quantum* computers (Nielsen and Chuang 2000). Quantum computers manipulate quantum states called *qudits* (most commonly *binary* qudits, or *qubits*). Unlike the computational states of digital computers, qudits are not unambiguously distinguishable from one another. This book will focus on classical (i.e., non-quantum) computation.

The same entities studied in the mathematical theory of computation—Turing machines, algorithms, and so on—are said to be implemented by concrete physical systems. This poses a problem: under what conditions does a concrete, physical system perform a computation when computation is defined by an abstract mathematical formalism? This may be called the *problem of computational implementation*.

The problem of computational implementation may be formulated in a couple of different ways, depending on our ontology of mathematics. Some people interpret the formalisms of computability theory, as well as other portions of mathematics, as defining and referring to abstract objects. According to this interpretation, Turing machines, algorithms, and the like are abstract objects.[1]

Abstract objects in this sense should not be confused with abstraction in the sense of focusing on one aspect of something at the expense of other aspects. For instance, we talk about the economy of a country and discuss whether it is growing or contracting; we do so by abstracting away from many other aspects of the objects and properties that constitute that country. I will discuss this notion of abstraction (partial consideration) later. Now let's deal with abstract objects.

Abstract objects are putative entities that are supposed to be non-spatial, non-temporal, and non-causal. In other words, abstract objects have no spatial location, do not exist through time, and are causally inert. The view that there are abstract mathematical objects and that our mathematical truths describe such objects truly is called *platonism* (Balaguer 2009; Linnebo 2011; Rodriguez-Pereyra 2011; Rosen 2012;

[1] E.g.: 'Computational models are *abstract entities*. They are not located in space and time, and they do not participate in causal interactions' (Rescorla 2014b, 1277, emphasis added).