8-15-2018 10:30 AM

Computing, Modelling, and Scientific Practice: Foundational Analyses and Limitations

Filippos A. Papagiannopoulos
The University of Western Ontario

Supervisor
Myrvold, Wayne
The University of Western Ontario

Graduate Program in Philosophy
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy
© Filippos A. Papagiannopoulos 2018

Abstract

This dissertation examines aspects of the interplay between computing and scientific practice. The appropriate foundational framework for such an endeavour is rather *real computability* than the classical computability theory. This is so because physical sciences, engineering, and applied mathematics mostly employ functions defined in continuous domains. But, contrary to the case of computation over natural numbers, there is no universally accepted framework for real computation; rather, there are two incompatible approaches –computable analysis and BSS model–, both claiming to formalise algorithmic computation and to offer foundations for scientific computing.

The dissertation consists of three parts. In the first part, we examine what notion of ‘algorithmic computation’ underlies each approach and how it is respectively formalised. It is argued that the very existence of the two rival frameworks indicates that ‘algorithm’ is not one unique concept in mathematics, but it is used in more than one way. We test this hypothesis for consistency with mathematical practice as well as with key foundational works that aim to define the term. As a result, new connections between certain subfields of mathematics and computer science are drawn, and a distinction between ‘algorithms’ and ‘effective procedures’ is proposed.

In the second part, we focus on the second goal of the two rival approaches to real computation; namely, to provide foundations for scientific computing. We examine both frameworks in detail, what idealisations they employ, and how they relate to floating-point arithmetic systems used in real computers. We explore limitations and advantages of both frameworks, and answer questions about which one is preferable for computational modelling and which one for addressing general computability issues.

In the third part, analog computing and its relation to analogue (physical) modelling in science are investigated. Based on some paradigmatic cases of the former, a certain view about the nature of computation is defended, and the indispensable role of representation in it is emphasized and accounted for. We also propose a novel account of the distinction between analog and digital computation and, based on it, we compare analog computational modelling to physical modelling. It is concluded that the two practices, despite their apparent similarities, are orthogonal.

Keywords: Definitions of Algorithms, Real Computability, Computable Analysis, Type-2 Turing Machines, BSS Model, Real-RAM, Foundations of Scientific Computing, Epistemology of Simulations, Conceptual Analysis, Idealisations, Formalisation of Mathematical Concepts, Decidability in Physics, Analog Computation, Physical Models, Analog and Digital, Philosophy of Computing, Representation, Open Texture

Acknowledgements

I would like to express my deep gratitude to my supervisor, Professor Wayne Myrvold, for his incisive comments and criticisms during the long process of writing this dissertation, as well as for all his advice and support. He has been particularly influential on my philosophical development and thinking. This dissertation would have been very different (indeed, even in a different area), had I not met him. For all of my intellectual debt, I would like to offer my deepest thanks.

I would like to express my very great appreciation to Professor Robert DiSalle, for the tremendously inspiring discussions, since my very first year in the program, the long support, the generosity, and the kindness. For all that, I'm profoundly grateful. I am also deeply indebted to Professor John Bell for being a great inspiration to me, through our unforgettable, stimulating discussions. I would also like to offer my special thanks to the Rotman Institute of Philosophy, for the stimulating environment and the workspace, and for all the material support.

Special gratitude is owned to Michael Cuffaro. His continuous support, confidence, help, and feedback on parts of the content of this dissertation had a hugely beneficial effect on my graduate school experience. I hope that I will continue to be honoured by his friendship and kindness for years to come.

Finally, although neither directly nor indirectly involved with this project, Professor Stathis Psillos has made a huge impact on my thinking, being a teacher, mentor, and friend, since I decided to undertake the Master's program in History and Philosophy of Science at the University of Athens. I'm very grateful for everything during these last eight years, and especially for instilling in me the desire to produce the best work I'm capable of.

To my father and the memory of my mother, and to my brother.

Contents

List of Figures

Chapter 1

Introduction

Most of us become familiar with algorithmic routines from our very early school years. We are taught how to perform operations, such as multiplication and long division, in a rather mechanical manner. Constructing truth-tables, finding least common multiples, or solving systems of linear equations by Gaussian elimination are also some examples of such mechanical thinking, still at a high-school level.

So the notion of 'algorithm', although informal, has for the most part been taken as unproblematic among mathematicians, and of no need for further formalisation. And when the need to make it more rigorous arose,[1] all offered mathematical models of algorithmically computable functions over the integers were actually proved equivalent; a fact that justified the certainty that there *is one* clear underlying concept of 'algorithmic computation'. Nevertheless, the situation turned out to be different when mathematicians tried to do the same for algorithmic computation of real-valued functions. Two very different traditions of computation have emerged, within which the notion of 'algorithm' is formalised in different, and often incompatible, ways. Both traditions have for the most part run a parallel non-intersecting course. Both are motivated by a desire to understand the essence of computation and of algorithm. Both aspire to discover useful, even profound, consequences (Blum, 2012), and both claim to offer a foundation for scientific computing (Blum 2004, Braverman and Cook 2006).

A basic aim of this work is to draw attention to these conceptual problems from the point of view of philosophers of (science and) mathematics, interested in the foundations of (scientific) computation. The strikingly limited philosophical literature on real computability is focused on arguing for one or the other of the rival models. But, to the author's knowledge, the implications of this situation for our conceptual understanding of 'algorithm' has gone unnoticed. A second aim is to connect important works on algorithms and computation that have mainly existed in parallel. Thus, I draw upon certain *conceptual analyses* of 'algorithms', coming from computer

[1]More details about this are in the next section.

science and logic, to account for the apparently incompatible pictures of ‘algorithm’, obtained from the different traditions of computation.

The structure and main argument of this chapter go as follows. Algorithmic computation has been unproblematic in denumerable domains, corroborating the Church-Turing thesis as a foundation for computability theory, and the conviction that *the* underlying intuitive concept is precise enough (sec.1.1). But in uncountable domains, the same concept is formalised via incompatible approaches (sec.1.3). And while one of those approaches is a natural generalisation of classical computability, the other expresses indispensable assumptions employed in certain areas of mathematics (sec.1.2, 2.1). Therefore, either one of the two approaches fails to pick out correctly its intended class of mathematical objects, or the intuitive idea of ‘algorithmic computation’ (if there is just *one* such idea) is semantically not as precise as traditionally thought (sec.2.2.2). Now, the history and practice of mathematics is consistent with the existence of a broader meaning of ‘algorithm’ than that captured by models of classical computability (i.e., than just ‘symbolic computation’; sec.2.3). Recent foundational attempts to define an ‘algorithm’ recognise, and are consistent with, various uses as well (sec.2.4). Therefore, it is a more natural explanation to accept that there isn’t just one informal-but-precise idea of ‘algorithm’ spanning all the areas of mathematics which make use of it (sec.2.5–2.6). We either use the term in more than one way, or the intuitive concept is not sharp enough to pick out a clear extension in all (unforeseen) situations; it exhibits *open texture*. I conclude that the latter of the two disjuncts might be the better choice (sec.2.6).

1.1 The Church-Turing thesis: some background

Intuitively, we have a good characterization of what an algorithm is. If a competent mathematician was asked to define ‘algorithmic routines’, her answer would most likely go something like the following. Algorithmic routines are finite step-by-step procedures that are set out in a finite number of instructions. Their steps are “small” enough so that they can be carried out by a calculator (human or otherwise) without significant cognitive capacities. No ingenuity or any acumen should be required from the calculator at any step.

Although rather broad in its expression, the above characterization was adequate for all mathematical purposes for many centuries. Even before Euclid’s time, mathematics had been concerned with seeking mechanical methods for solving classes of problems. In all positive cases, the above intuitive understanding was enough, since whenever asked for such a solution, one would just have (a) to come up with a mechanical method and (b) show that this method is always reliable. And the first part would require that mathematicians are able to recognize an algorithmic routine as such, whenever they have one before their eyes. This ability, of course, has always been taken for granted. There was no need to formalise the intuitive idea of algorithmic calculation, for it must have seemed unproblematic.

Nevertheless, when we move to negative cases, the need for a rigorous characterization arises. Proving that there is *no* algorithm for a specific problem, means that we need to be able to say something about the class of algorithms as a whole; that is, that there is no member of this class which solves the problem at hand. And since we are not able to run exhaustively through each member of this class, we need to be able to talk precisely about the class as a whole. Thus, when Alonzo Church and Alan Turing showed independently that the *Entscheidungsproblem*[2] is unsolvable, they both achieved this by means of the formal counterparts of algorithmic calculation they had put forward: Turing developed the notion of (what is now called) 'Turing computability', that is computability by a Turing machine, and Church developed the notion of 'λ-definability'.

These were not the only accounts put forward at the time. K. Gödel —together with J. Herbrand— had developed the notion of a 'general recursive function'. S. Kleene, who extended the notion of 'computability' to partial functions, later also developed an alternative characterisation of Gödel's notion, namely the 'μ-recursive' functions.

Remarkably, all these notions were soon found to be extensionally equivalent. Turing showed the equivalence of Turing-computability with λ-definability. Church and Kleene also proved that the classes of λ-definable, general recursive, and μ-recursive functions are the same.[3] Putting all these results together, we can accept the following statement as what has come to be known as 'the Church-Turing Thesis' (CTT):

> The effectively computable functions over the non-negative integers are the μ-recursive/λ-definable/general recursive/Turing-computable functions.[4]

Turing machine computation has been a successful *explication* of the intuitive notion of 'effective computation' (that is, 'computation by an idealised agent following an algorithmic routine, and with unlimited time and space), and the CTT is regarded as a foundational principle of computer science in general, and of computability and complexity theory in particular. It is almost universally accepted as correct by mathematicians and computer scientists.

[2]The *Entscheidungsproblem* was the problem of finding an algorithmic method for deciding whether a sentence of 1st order logic is logically valid or not.

[3]Thus, all these terms are often used interchangeably in the relevant literature.

[4]The list of co-extension notions could keep going on, of course, since there now exist more equivalent models available, such as register machines, Post systems, Markov systems etc. Nothing really relies on the choice of those particular four instead of others.

1.2 Is the Church-Turing thesis inadequate for real functions?

Nevertheless, despite the wide acceptance of the Turing machine model as a fruitful explication of ‘algorithm’, there have been complaints by some mathematicians to the effect that it is not adequate as a foundation for areas of mathematics involving algorithms over the *real numbers*, such as numerical analysis.

Blum et al. (1997), for example, say that although numerical analysis is all about algorithms, “there is not even a formal definition of algorithm in the subject” (p.23), despite the field’s origins going centuries back. Furthermore, although scientific computing[5] *is* computing and although the Turing computability model “a firm foundation to computer science as a subject in its own right”, the present view of the digital computer as a discrete object and the Turing machine as a foundation for real number algorithms “can only obscure concepts” (p.23).

Similarly:

> [T]he Turing model with its dependence on 0s and 1s is fundamentally inadequate for giving a foundation to [..] scientific computation, where most of the algorithms [..] are real number algorithms. (Blum, 2004, 3)

Thus:

> We want a model of computation which is more natural for describing algorithms of numerical analysis, such as Newton’s method [..] Translating to bit operations would wipe out the natural structure of this algorithm.

As a result, Blum et al. set out to develop a formal account of real computation and real number algorithms; that is a formal model of computing functions whose domain is (possibly some subset of) $\mathbb{R}$, instead of $\mathbb{N}$, and of solving decision problems about sets which are (possibly some subset of) $\mathbb{R}$.

It should be mentioned, though, that even in his 1936 seminal paper, Turing was not interested in computation over the positive integers only but also over the reals. His motivation in proposing his computational model was to be able to say what it means for a *real* number to be a computable one:

> The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. (Turing, 1936, 230)

[5]Blum et al. treat the terms ‘scientific computing’ and ’numerical analysis’ as synonymous.

Although Turing did not actually give a complete account of computable real functions in that paper, there were many later attempts, stemming from his work, to develop formal models of effective computation over the reals. Here, we in fact have a cluster of various approaches, which are for the most part equivalent in their theoretical results, and which I will put, for the purposes of this work, under the rubric of 'effective approximation approaches'.[6] Different variations of this approach include *Recursive Analysis* (Pour-El and Richards, 1989), *Computable Analysis* (Aberth, 1980; Weihrauch, 2000), *Bit-computation* (Braverman and Cook, 2006), *Complexity of real functions* (Ko, 1991), etc. All of these models, though, are in fact based on the (independent) work of Grzegorczyk (1955) and Lacombe (1955) on computing over the reals.

1.3 Two very different approaches

We here examine the two main different approaches; the model developed by Blum et al. (aka '*the BSS model*') and some different formalisations of the *Effective Approximation* approach.

1.3.1 The BSS model

The BSS model is a model of computation not exclusively over the reals but over any arbitrary field or ring R. This means that not only algorithms over $\mathbb{R}$ can be modelled but algorithms over $\mathbb{C}$ as well. If R is $\mathbb{Z}_2 =< \{0,1\}, +, * >$, then the model becomes reduced to classical computability theory. The standard reference is Blum et al. (1997), but a nice, brief, exposition can also be found in Blum (2004).

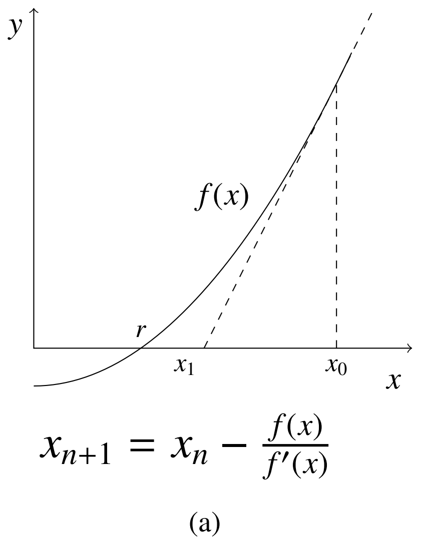
The model is based on the notion of a *machine M over R* (where R is a commutative —possibly ordered— ring or field). This machine has an *input* and *output* space associated with it, a 2-way *infinite* tape with cells, and, similarly to Turing machines, a *read-write* head, as well as the machine's *program*, which is a finite directed graph, with five types of nodes, related either with operations or *next mode* mappings.

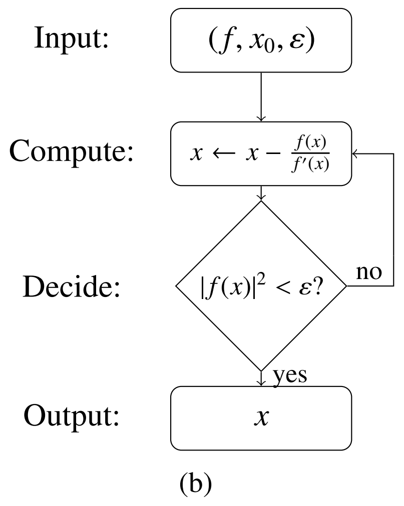
More specifically, the top node represents the input stage and the last node the output stage. At each other stage, passage from a node to another is made either by means of a computation or by means of a decision step whose branches may lead to a new node or a previous one.[7] The last kind of node is a *shift node* associated with shifting the head one cell to the right or to the left.

Within this framework then, the authors prove several results about the decidability of sets over $\mathbb{R}$ and $\mathbb{C}$ (e.g., that the Mandelbrot set is undecidable over $\mathbb{R}$). Figure 1.1 shows an example

[6] I borrow this name from Feferman (2013).

[7] Crucially, each computational node has built in a polynomial or rational map $g_n : R^n \to R^m$, where $n, m \leqslant k$ and k the fixed number of contiguous cells on the tape that the head can view at one step.





Input:
(f, x_0, ε)
Compute:
$x \leftarrow x - \frac{f(x)}{f'(x)}$
Decide:
$|f(x)|^2 < \varepsilon?$
no
yes
Output:
x
(b)

Figure 1.1: (a) Newton-Raphson method for approximating a root of $f(x)$. (b) A BSS program implementing the method

of how Newton's method (perhaps the most standard algorithm in Numerical Analysis texts) is naturally captured by this model (see also Blum et al., 1997, 10). A crucial idealisation that Blum et al. employ, though, is that the machine M is able to manipulate the *exact value* of any real number it operates on. Real numbers are viewed as whole entities and algebraic operations and comparisons are each counted as one unit of work; that is, it always takes one step to add, subtract, multiply, divide, and compare two real numbers, even if they are irrationals. This is not physically possible to implement though, and thus much of the criticism directed at the model is based on that fact. We'll say more about this in due course.

1.3.2 Effective Approximation

As we have already said, other mathematicians have been in the business of extending Turing computability to the domain of the reals, almost since the early 50's. Although there is no universally accepted formalisation of the notion of effective computation of a real-valued function, most models are for the most part equivalent, regarding their theoretical results.

It is not possible to try even a brief exposition of the different approaches here. We will only give the basic flavour of some main characteristics. For further details, a now standard reference text is Weihrauch (2000). Friendly short expositions can also be found in Braverman and Cook (2006) and Pégny (2016).

The main problem facing us when we move from computing over the integers to computing over the reals has to do with representation. Representation is essential in (Turing) computation, since a computing agent (Turing machine or other) actually manipulates symbols from a finite alphabet. Nevertheless, when we move to an uncountable domain, such as $\mathbb{R}$, we still have only countably many names available to denote uncountably many entities. This creates thorny conceptual problems.

Effective approximation tackles this problem by positing that instead of the exact values of the reals, the computor deals with sequences of rationals approximating those values. Intuitively, a function is computable if, given a "good" rational approximation of the input, there is an algorithm that yields a "good" rational approximation of the output. Suppose a real number x and that we want to compute $f(x)$. Since x is a real number, there is a Cauchy sequence of rationals $\langle r_k \rangle_{k \in \mathbb{N}}$ which approaches x as its limit. Now, to compute $f(x)$ means to effectively determine another rational sequence $\langle r_m \rangle_{m \in \mathbb{N}}$ that approaches $f(x)$ as its limit.

Following are some different formalisations of the above intuitive idea.

The Grzegorczyk/Pour-El & Richards approaches We say that:[8]

- A *sequence of r_k rational numbers is computable*, if there exist three recursive functions, $a, b, s : \mathbb{N} \to \mathbb{N}$ such that, for all $k, b(k) \neq 0$ and

$$r(k) = (-1)^{s(k)} \frac{a(k)}{b(k)}$$

- A *sequence of r_k rational numbers effectively converges* to a real number x, if there exists a recursive function $n : \mathbb{N} \to \mathbb{N}$, such that for all m:

$$k \geqslant n(m) \text{ implies } \mid x - r_k \mid \leqslant 2^{-m}$$

- A *real number is computable*, if there exists a computable sequence r_k of rational numbers that effectively converges to x.

- A *function $f : I \to \mathbb{R}$* (where the endpoints of I are computable reals) is *computable* iff:

  – f is *sequentially computable*: if x_i is a computable sequence of points in I converging to $x \in I$, then the sequence $f(x_i)$ is computable and converges to $f(x)$.

[8] I reproduce here from Pour-El and Richards (1989, ch.0).

– f is *effectively uniformly continuous*: there exists a recursive function $d : \mathbb{N} \to \mathbb{N}$ such that for all $x, y \in I$ and for all $n \in \mathbb{N}$,

$$\mid x - y \mid \leqslant \frac{1}{d(n)} \text{ implies } \mid f(x) - f(y) \mid \leqslant 2^{-n}$$

The above definition is actually due to Grzegorczyk (1955). It can be seen as formulating effective versions of the usual definitions in real analysis.

Another approach is due to M.B. Pour-El and Caldwell (1975), suggested by the Weierstrass approximation theorem, which states that every continuous function $f : [a, b] \to \mathbb{R}$ can be approximated by a sequence of polynomials $p_n(x)$ over $\mathbb{Q}$ with non-zero rational coefficients. The effective version of this is as follows.

A function f on a domain I is *computable* if there exists a computable sequence of rational polynomials $p_k(x)$ which effectively converges to f; that is, if there exists a recursive function $n : \mathbb{N} \to \mathbb{N}$, such that for all m and all $x \in I$:

$$k \geqslant n(m) \text{ implies } \mid f(x) - p_k(x) \mid \leqslant 2^{-m}$$

This can be extended over the whole $\mathbb{R}$ as well (see, Pour-El and Richards 1989, ch.0).

This model is provably equivalent to the previous one, on any closed interval I and on $\mathbb{R}$.

Type-2 Theory of Effectivity (TTE) A different formalisation of computable functions is in terms of Turing machines.[9] Here, the basic assumption is that if the output value is one that requires an infinite string of symbols in order to be represented, then the Turing machine computing the function, yields a (series of) approximating representations of it. The following formalisation is due to Weihrauch (2000).

More precisely, in classical computability theory, Turing computation of a numerical function $f :\subseteq \mathbb{N}^n \to \mathbb{N}$ is expressed as computation of string functions $f :\subseteq (\Sigma^*)^n \to \Sigma^*$, where Σ^* is the set of all finite words over a non-empty finite alphabet Σ. Computability of other sets of objects (rational numbers, graphs, etc.) becomes possible by means of number coding. Nevertheless, since Σ^* is actually a countable set, the above formalisation is not adequate for dealing with computability over uncountable domains, such as $\mathbb{R}$. Therefore, we also consider the set Σ^ω of infinite sequences of symbols from Σ.[10]

[9]Commonly referred to as ‘Type 2 Turing machines’. If we see natural numbers as type 0 objects, and reals as functions mapping naturals n to rationals r_n (i.e., type 1), then a real function is actually a functional, mapping functions to functions; hence, it can be seen as a type 2 object. Accordingly, effective computability over the reals is referred to as ‘Type 2 Effectivity’ (TTE).

[10] Σ^ω has the same cardinality as $\mathbb{R}$.

For $k \geqslant 0$ and $Y_0, Y_1, ..., Y_k \in \{\Sigma^*, \Sigma^\omega\}$, a string function $f :\subseteq Y_1 \times ... \times Y_k \to Y_0$ is *computable* iff there is a *Type-2 Turing machine M* with k input tapes such that: given input $(y_1, ..., y_k)$ with each (finite or infinite) sequence $y_i \in Y_i$ written on the i input tape, the machine outputs a $y_0 \in Y_0$, such that one of the following two cases holds:

1. $f_M(y_1, ..., y_k) = y_0 \in \Sigma^*$ and M has halted.

2. $f_M(y_1, ..., y_k) = y_0 \in \Sigma^\omega$, M computes forever and writes y_0 on the output tape.

A *Type-2 Turing machine M* is a Turing machine with k input tapes together with a type specification $(Y_1, ..., Y_k, Y_0)$ with $Y_i \in \{\Sigma^*, \Sigma^\omega\}$ which gives the type for each tape. If M computes forever but writes only finitely many symbols on the output tape, $f_M(y_1, ..., y_k)$ is undefined.

The above is a theoretical model. Infinite strings and infinite computations cannot be written and completed in reality. However, the notion of 'effectivity' implies that an effective computation must be able to terminate after a finite number of steps. To alleviate these concerns, we add the extra restrictions that the input and output tapes are one-way read-only and one-way write-only (no written symbol can be erased). These restrictions guarantee that after some finite number of steps, M will have read some finite initial part of the inputs and will have written some finite initial part of the output. Additionally, the requirement of one-way output guarantees that whatever has been written on the output tape, after *finitely* many steps cannot be erased and is, thus, a correct *prefix* of $y_0 \in Y_0$. Then, the infinite computations of Type-2 machines can be *approximated* by physical computations with arbitrary precision (Weihrauch, 2000, 16).

Ker-I Ko's approach Another route to alleviate concerns about the theoretical idealisation of case 2 above, and tie Turing machines to physically implementable real computation, is to restrict the input strings to $y_i \in Y_i$ where $Y_i = \Sigma^*$ only and equip the Turing machine with an oracle (see, Ko 1991). An oracle is a kind of "black box" that, at any given step of the computation, can be queried by the TM to provide the value of a function $\phi : \mathbb{N} \to \Sigma^*$, in one step. A machine with an oracle is a TM with input and output tapes and with one extra distinguished tape —called 'the query tape'— on which a statement 'call oracle' can appear arbitrarily often. Whenever the machine M reaches this statement, it replaces in one step the current word w written on the query tape with $\phi(|w|)$.[11] According to Ko's formulation, real numbers are represented as *Cauchy functions* and, so, the computing machine M is an oracle TM mapping Cauchy functions to Cauchy functions.

Here is an informal presentation of this approach put to work (we skip a formal treatment, which would be in terms of Cauchy functions, in the interests of brevity). Suppose that for

[11] '$|w|$' denotes the *length* of w.

some x we want to compute $f(x)$, where $f :\subseteq \mathbb{R} \to \mathbb{R}$. The trick is to work with (a series of) approximations of the inputs and outputs to effectively computable precisions.

- We give as input to machine M the integer n such that the output y_0 must be within the error 2^{-n}; that is, $|f(x) - y_0| \leqslant 2^{-n}$.

- M computes the integer m, such that 2^{-m} is the required precision of the input y_1; that is, $|x - y_1| \leqslant 2^{-m}$.

- Now, M refers to the oracle to obtain y_1; that is, the oracle computes in one step $y_1 = \phi(m)$ such that $|x - \phi(m)| \leqslant 2^{-m}$.

- M computes y_0.

The running time of M is the time taken to compute m from n plus the time taken to compute y_0 from $y_1 = \phi(m)$. Crucially, all these have *finite* representations. Thus, the computation of $f :\subseteq \mathbb{R} \to \mathbb{R}$ does not require reading/writing of any infinitely long strings or an infinite number of steps, according to this approach.

The machine M is said to compute f iff, given that the result $\phi(m)$ is a Cauchy function for $x \in \mathbb{R}$, and on input 0^{m}, the output is a Cauchy function for $f(x)$ (for all $x \in \mathrm{dom}(f)$ and $\phi : \mathbb{N} \to \Sigma^*$).

Generalization to multivariate functions $f :\subseteq \mathbb{R}^k \to \mathbb{R}$ is straightforward, by allowing k oracles, $\varphi_1(m), \varphi_2(m), ..., \varphi_k(m)$ that can be queried with respect to m.

All the above models are, as already mentioned, virtually equivalent with respect to their results; for example, which functions over the reals are computable.[12] They also are all based on the (independent) work of Grzegorczyk (1955) and Lacombe (1955); though, the earliest work in this area goes back to 1937-1939, done by Banach and Mazur, but published several decades later (Mazur, 1963) owing to WW2. Nevertheless, the groundlaying work is definitely again Turing's (1936). For a thorough and illuminating examination of the development of computable analysis, from Turing's seminal work to this day, see Avigad and Brattka (2014).

1.3.3 Incompatibility between the two main approaches

There is a clear sense in which the two main approaches are incompatible. Almost all of the functions characterized as computable by BSS are non-computable by Effective Approximation (EA), and vice-versa (see, also, Weihrauch 2000).

More precisely, a major result, which holds in all EA models, is the following:

[12]However, the presentation here is by no means exhaustive. See Weihrauch (2000, ch.9), for other approaches, such as domain theory, Markov's approach, etc., and brief comparisons between them.

Any computable function is continuous.

We have already seen a version of this before (p.8), in Grzegorczyk's definition of computable functions where any such functions were required to be *effectively uniformly continuous*. Although in Grzegorczyk's definition the continuity property is obtained by fiat, as part of the definition, in Weihrauch's model it is a consequence of the requirement that whatever is written on the output tape of a Type-2 TM is already a correct prefix of the output and cannot be erased. Thus, any finite portion of the output is already determined by a finite portion of its input (Weihrauch 2000, Th.1.3.4 and 4.3.1). Intuitively, this holds because if, say, we're computing an approximation to a value $f(x)$, we want this approximation to be a good one for all points near x, since the latter is also given by an (arbitrarily precise) approximation.

A brief comment on the continuity result. One can notice the striking similarity between the continuity requirement in computable analysis and in constructive mathematics. In the latter too, every real valued function is continuous at every point at which it is defined. And one cannot always separate any two real numbers, since in some cases that might require an infinite amount of information, about one or both numbers. Thus, the following statement does *not* always hold for any two reals $x_1, x_2 \in \mathbb{R}$:

$$x_1 < x_2 \text{ or } x_1 > x_2 \text{ or } x_1 = x_2 \tag{1.1}$$

Owing to the facts that (a) the father of intuitionism, Brouwer, held the continuity requirement to be true[13] and, (b) Borel (1912) had held a similar view even earlier, in his discussion about computable real numbers and functions, Myrvold (1995) calls this requirement 'the Borel-Brouwer Thesis'.

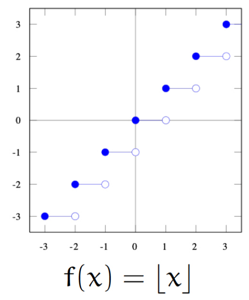
Similarly to constructive mathematics, (1.1) does not always hold in EA models either. The equality relation and comparisons are in general non-computable.

In the BSS/Real RAM approach, on the other hand, there's no similar continuity requirement. That means that functions seemingly easy to define, such as step or floor/ceiling functions, are not EA-computable.[14] See Fig.1.2.

Furthermore, since the comparison operation is discontinuous, almost every non-trivial branching node in the program of a BSS machine introduces a point of discontinuity. Thus, most BSS-computable functions are non-continuous and, so, non EA-computable. Conversely,

[13] In fact, for intuitionists a stronger result holds: any continuous real valued function on a closed interval is *uniformly* continuous.

[14] More precisely, computation of these functions becomes harder the closer we get to the discontinuity point and non-computable at the discontinuity point. Nevertheless, they can be computable over any subdomains in which they are continuous.



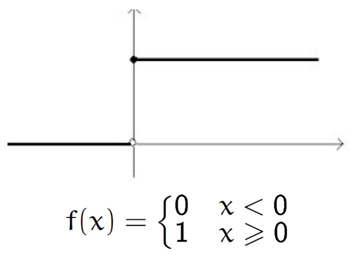


Figure 1.2: A floor and a step function. Functions with discontinuity points are not EA-computable everywhere.

functions such as square roots or exponentials are not BSS-computable, whereas they're unproblematic for EA.[15] One can build such functions into a BSS machine's program as primitive operations, but this does not really address the problem, since there would still remain EA-computable functions (e.g., transcendental ones) that are not BSS-computable.

This, creates a strong incompatibility between the two main approaches. One might even argue that they in fact are incomparable; they cannot be compared because they fulfil different purposes.[16]

While, indeed, the two approaches are meant to serve different purposes in general, there are however two particular aims they share in common. First, they both claim to provide, among other things, a foundation for scientific computing. Blum's (2004) short and very clear exposition of the BSS model in the *Notices of the American Mathematical Society* aims to draw attention on the model as a foundation for numerical analysis and computational science. Nevertheless, two years later, again in the *Notices of the AMS*, Braverman and Cook (2006) presented a version of an EA-model (under the name 'bit computation') as a better foundation for scientific computing; a main point being there that their version tracks and approximates how real computers actually compute better. See, Fig.1.3.

[15]Almost all of the commonly encountered continuous functions in analysis are EA-computable, on an appropriate domain, including, for example, $f(x_1, ..., x_n) = c$, $f(x_1, ..., x_n) = x_i$, polynomials, trigonometric functions, $\min(x_1, x_2)$, $\max(x_1, x_2)$, $\frac{1}{x}$, e^x, $\sqrt[n]{x}$, $|x|$, etc. (see, Weihrauch, 2000).

[16]Pégny (2016), for example, argues that the BSS/Real-RAM models are models of analog computation which are not physically implementable and so don't really pose a challenge to EA approaches.

Computing over the Reals: Where Turing Meets Newton

Lenore Blum

The classical (*Turing*) *theory of computation* has been extraordinarily successful in providing the foundations and framework for theoretical computer science. Yet its dependence on 0s and 1s is fundamentally inadequate for providing such a foundation for modern scientific computation, in which most algorithms—with origins in Newton, Euler, Gauss, et al.—are real number algorithms.

In 1989, Mike Shub, Steve Smale, and I introduced a theory of computation and complexity over an arbitrary ring or field R [BSS89]. If R is $\mathbb{Z}_2 = (\{0,1\},+,\cdot)$, the classical computer science theory is recovered. If R is the field of real numbers been a powerful tool in classical complexity theory. But now, in addition, the *transfer of complexity* results from one domain to another becomes a real possibility. For example, we can ask: Suppose we can show $P = NP$ over $\mathbb{C}$ (using all the mathematics that is natural here). Then, can we conclude that $P = NP$ over another field, such as the algebraic numbers, or even over $\mathbb{Z}_2$? (Answer: Yes and essentially yes.)

In this article, I discuss these results and indicate how basic notions from numerical analysis such as *condition*, round-off, and approximation are being introduced into complexity theory, bringing together ideas germinating from the real calculus

Computing over the Reals: Foundations for Scientific Computing

Mark Braverman and Stephen Cook

Introduction

The problems of scientific computing often arise from the study of continuous processes, and questions of computability and complexity over the reals are of central importance in laying the foundations for the subject. The first step is defining a suitable computational model for functions over the reals.

Computability and complexity over discrete spaces have been very well studied since the 1930s. Different approaches have been proved to yield equivalent definitions of computability and nearly equivalent definitions of complexity. From the tra- computation model. Alan Turing defined the notion of a single computable real number in his landmark 1936 paper [Tur36]: a real number is computable if its decimal expansion can be computed in the discrete sense (i.e., output by some Turing machine). But he did not go on to define the notion of computable real function.

There are now two main approaches to modeling computation with real number inputs. The first approach, which we call the bit-model and which is the subject of this paper, reflects the fact that computers can store only finite approximations to real numbers. Roughly speaking, a real function

Figure 1.3: The two articles appeared in the *Notices*. Both models claim to offer a foundation for scientific computing.

But another purpose for both approaches is to formalise the concept of ‘algorithm’. In the tradition of logic and computer science, effective computation of a function was always understood as computation by following an algorithm. But numerical analysis, being a centuries old field —concerned with mechanical methods for tackling various problems such as, solving linear or differential or integral equations, data interpolation, optimization problems, etc.— also has the notion of ‘algorithm’ at its core. Therefore, although the task of choosing among the various theories of real computation is important on its own, I think that the very existence of such diverse approaches can offer us some insights into our intuitive notion of ‘algorithm’ itself.

Chapter 2

Algorithms

As we have said, ‘algorithm’ is a central concept in numerical analysis, as a long-standing field concerned with methods for numerical problem solving. No wonder then that a formal theory aiming to offer a foundation for the field would also claim that it formalises the notion of ‘algorithm’:

> The situation in numerical analysis is quite the opposite. Algorithms are primarily a means to solve practical problems. There is not even a formal definition of algorithm in the subject. [..]
>
> [But a]s long as the computer is seen simply as a finite discrete object, it will be difficult to systematize numerical analysis. We believe that the Turing machine as a foundation for real number algorithms can only obscure concepts. (Blum et al., 1997, 23)

And, after having sketched the BSS model, Blum et al. continue:[1]

> Now formulating a theory of computation [over a field K and choosing K to be the real numbers $\mathbb{R}$..], we are able to obtain a setting that provides a foundation of numerical analysis. The notion of an algorithm over $\mathbb{R}$ becomes well-defined as a mathematical object in its own right. So we have developed an *extension* of the classical theory to a new theory which can be specialized to the study of real number algorithms. (Blum et al., 1997, 30, emphasis in original)

The contrast of this “opposite situation”, in the first quote, is with the case of classical computability theory. As we have already mentioned earlier (p.4), the Church-Turing thesis is almost universally accepted as a foundational principle for computer science. The groundbreaking

[1]See, also, the quotes on p.5.

work by logicians back in the 30s, and later, successfully captured (extensionally) the notion of an ‘effective procedure’. But saying that ‘a function is effectively computable’ can be translated as ‘there is a sequential algorithm for computing this function’. So, it is generally asserted that the aforementioned logicians’ work successfully captured (extensionally) the notion of ‘a sequential algorithm such that..’.

But, this formal work is just as naturally extended to real function computability by the effective approximation approaches. So it seems equally natural to consider that the notions of ‘effective computation of a real function’ and ‘sequential algorithm for computing a real function’ are suitably *explicated* by some strand from the effective approximation cluster of approaches, say, Type-2 Turing machines. And, indeed, EA is widely accepted as the correct framework for formalising real computability, within certain fields such as logic, theoretical computer science, physics, and also philosophy.[2]

Nevertheless, Blum et al. are by no means the only mathematicians complaining about how unnatural the effective approximation models are for providing a foundation for certain mathematical fields. Researchers from the area of computational geometry have expressed similar complaints and one of the ground-laying texts in the subject (Preparata and Shamos, 1985) uses the Real-RAM model as a formal foundation of geometric algorithms.[3]

Even more, information-based complexity uses Real-RAM as the standard model of computation for the analysis of algorithms. Hopefully, the reader will excuse a brief digression, in

[2]For an insightful defence of EA as the appropriate framework to formalise effectivity, see Pégny (2016).

[3]Here is a rather long but very characteristic quote, distilling the attitude of researchers in computational geometry towards effective approximation (emphasis in original):

> Most (classical) results in computational geometry are heavily tied to issues in *combinatorial* geometry, for which assumptions about coordinates being integral or algebraic are (at best) irrelevant distractions. Speaking as a native, it seems completely natural to consider *arbitrary* points, lines, circles, and the like as *first class objects* when proving things about them, and therefore equally natural when designing and analyzing algorithms to compute with them.
>
> For most (classical) geometric algorithms, this attitude is reasonable even in practice. Most algorithms for planar geometric problems are built on top of a very small number of geometric primitives: Is point p to the left or right of point q? Above, below, or on the line through points q and r? Inside, outside, or on the circle determined by points q,r,s?
>
> [...] For similar reasons, when most people think about sorting algorithms, they don’t care *what* they’re sorting, as long as the data comes from a totally ordered universe and any two values can be compared in constant time.
>
> So the community developed a separation of concerns between the design of real geometric algorithms and their practical implementation; [...] TTE, domain theory, Ko-Friedman, and other models of “realistic” real-number computation all address issues that the computational geometry community, on the whole, just doesn’t care about.

From: Jeffε (https://cstheory.stackexchange.com/users/111/jeff%ce%b5), What are the reasons that researchers in computational geometry prefer the BSS/real-RAM model?, URL (version: 2010-10-13): https://cstheory.stackexchange.com/q/2124 order to discuss how the Real-RAM model is employed in those disciplines (the latter using the name 'real number model').

2.1 The BSS/Real-RAM in other areas of mathematics

Computational geometry. Computational geometry is concerned with the study of algorithms for solving geometrical problems, which, in a sense, are the computational counterparts of rule-and-compass constructions of classical geometry. The latter field had been profoundly influenced by developments in real analysis, for example, metric spaces and convexity theory. Due to these developments, the treatment of classically geometrical notions, such as distance or convexity, was transformed to treatment of analytic notions, such as *metrics* or properties of *finite* subsets. Accordingly, the objects considered in the descendant field of computational geometry commonly are sets of points $(x_1, x_2, ..., x_k)$, where x_i real, in an *n-dimensional Euclidean space* with metric $(\sum_{i=1}^{n} x_i^2)^{1/2}$.

Computational geometry involves two fundamental elements, *algorithms* and *data structures*. The formulation and analysis of the algorithms —e.g. their cost— is with respect to some specific computation model. Data structures are ways of organizing the complex objects manipulated by the algorithms —most commonly, sets, ordered and unordered— by means of the simpler data types directly representable by the computer.[4]

In order for an appropriate model of computation for a mathematical domain to be chosen, the nature of problems dealt with in the domain is essential. Generally speaking, in computational geometry most problems can fit into one of the following categories.[5] (a) Subset selection: given a collection of objects, find a subset that satisfies a certain property (e.g., the two closest ones in a set of n points). (b) Computation: Compute the value of some geometric parameter of a given set of objects (e.g., the distance between a pair of points). (c) Decision: with any instance of the above two categories, we can also associate a decision problem, with a YES/NO answer (e.g., does a specific subset S satisfy property P? Is the distance between points A and B greater or

[4]Data structures can be classified according to what operations they involve. For example, if S is a subset of A, represented in a data structure, and u an arbitrary member of A, the fundamental operations are:
MEMBER(u, S): Decide whether $u \in S$
INSERT(u, S): Add u to S.
DELETE(u, S): Remove u from S.
Assuming now a collection of disjoint sets $S_1, S_2, ..., S_k$, more complex operations can be:
FIND(u): Report $i : u \in S_i$
MIN(S): Report the minimum element of S (S is totally ordered).

Accordingly, data structures are classified on the basis of supported operations. For example, a *Dictionary*, is a data structure supporting MEMBER, INSERT, DELETE, and a *Priority queue* is a data structure supporting MIN, INSERT, DELETE (Preparata and Shamos, 1985).

[5]See, Preparata and Shamos (1985, 27).

equal to a fixed constant k?).

The adopted model is, as we have said, Real-RAM, which is essentially equivalent to BSS in the finite-dimensional case. A real-RAM machine is a *register machine*, but capable of storing the exact value of a real number in each memory cell (register).[6] The inputs to the machine are unanalysed entities in an algebra A, making it an algebraic approach, similarly to BSS. Thus, the four arithmetical operations ($\pm, \times, \div$) as well as comparisons between any two real numbers ($<, \leqslant, =$) are primitive and available *at unit cost*. Relative to specific applications, several analytical functions, such as n-root, trigonometric ones, etc., can be taken as primitive and built into the model.

When a real-RAM executes an algorithm for a decision problem,[7] the overall computation may be viewed as a path in a rooted binary tree, whose particular nodes involve either a calculation, or a branching based on the result of a comparison. Such a tree corresponds to another computational model, the "algebraic decision tree". This is an important fact, because the worst-case running time of a real-RAM program will be at least proportional to the length of the longest path from the root (representing the initial step of the computation) to a leaf (representing a last step and containing a possible answer YES/NO output), in the respective algebraic decision tree. Thus, the Real-RAM model and its connection to algebraic decision trees are very useful for analysing algorithms in terms of lower bounds of their costs.

Information-based complexity (IBC). *Information-based complexity* is the branch of computational complexity that studies the complexity of problems in which information is *partial*, *noisy*, and *priced*. 'Information' in this context is not meant in the same sense as in Claude Shannon's information theory. It rather is about what we know about the problem whose complexity and solution we're interested in. More often than not, information is limited in a variety of problems, from science and engineering to economics and mathematical finance, to control theory, to computer graphics and vision, etc. Assume, as a toy example, that we need to compute $\int_0^1 f(x)dx$. Very often, we cannot apply the fundamental theorem of the calculus, and we have to numerically approximate the solution (Traub et al., 1988). All we can do is to input in the computation the value of f at certain points (perhaps, obtained by some program calculating $f(x)$) and this is the available information. It is *partial*, since there is in general an infinite number of integrands having the same values at these points, and our case cannot be distinguished from any of those; we have limited knowledge only. If we also have round-off or representation errors (e.g., owing to use of a digital computer), then the information is *noisy* or *contaminated*. Both incompleteness and contamination of information bring about an intrinsic

[6]'RAM' stands for 'Random Access Machine', which is another name for register machines.

[7]Recall that both subset-selection and computation problems can be transformed to a decision problem.

uncertainty in the answer, say ε. In addition, it is assumed that information is *priced*: there is a cost for every piece of information we obtain (e.g., in the previous example, for every value of $f(x)$ at a point). All such cases are contrasted with problems studied in *combinatorial complexity* where information is *complete*, *exact*, and *free*, such as the travelling salesman problem. Typically, the way this problem is studied in combinatorial complexity is by assuming that all the cities and all the distances between them are known completely, exactly, and with no cost.

Similarly to combinatorial complexity,[8] information-based complexity deals with the study of all possible algorithms for solving a problem, in order to evaluate its complexity, in a way that the latter will be a problem invariant, that is, independent of any particular algorithm used. Since problems studied in IBC *ex hypothesi* can only be solved approximately, it is required that a solution will not have error greater than a threshold ε. Then, the ε-complexity of a problem is the minimal cost among all algorithms which solve the problem with error at most ε (Traub et al., 1988, 3). Accordingly, the cost and error of algorithms are defined according to three different settings: a *worst case*, an *average*, and a *probabilistic* setting.

The model of computation used in IBC, in order to analyse algorithm costs and, hence, complexity of problems, is called '*Real number model*' and it's equivalent to BSS/Real-RAM. The crucial assumption is, again, that the four arithmetic operations and comparisons between real numbers are primitive and that they can be performed exactly and at unit cost.

Other fields Finally, besides numerical analysis, computational geometry and information-based complexity, the Real-RAM/BSS model has been used in the field of *computer algebra* and mainly in *algebraic complexity*. The concern here is with algorithmic problems which can be solved by means of algebraic algorithms (Bürgisser et al., 2013).

2.2 The problem about algorithms

So it becomes apparent that although 'algorithm' is the central notion in various areas of mathematics, it is however treated and formalised differently. Nevertheless, if all mathematicians sat around a table, so to speak, they would all agree, at an informal level, about what kind of routines qualify as algorithmic ones and what features such routines should have. And, in a sense, this is what has been done for centuries, when the intuitive notion of 'algorithm' was taken as clear and unproblematic and with no actual need for further formalisation. But, then,

[8]Very often and in many contexts, what we refer to here by 'combinatorial complexity' is just called 'computational complexity' with no risk of confusion. However, we here stick to a terminological distinction between 'information complexity' and 'combinatorial complexity', because it matters in this context.

despite any intuitive consensus, mathematicians from different fields actually end up *explicating* the informal idea in not only different but actually incompatible ways. How can this be so?

2.2.1 Three levels of formalisation

Smith (2013, 44-45) distinguishes three levels of conceptualisation in play, when we come to formalise intuitive concepts. I will adopt his scheme for the purposes of this work and with respect to the notions of 'computation' and 'algorithm'.

First, we can say that there is a *pre-theoretic* level, at which the basic notion of 'computation' is actually a vague, rough idea, stemming mainly from paradigmatic cases of algorithmic procedures in the mathematical practice (e.g., the sieve of Eratosthenes or Euclid's algorithm). The ideas are still inchoate at this level and we rather have to do with a cluster of notions instead of a simple one. There's not one definite idea of 'computation' (or 'calculation') but various distinct cases: effective computation, machine computation, analog computation, geometric constructions, etc. All these are not excluded by any pre-theoretical talk of computation or of mechanical procedures.

Second, at a next, *proto-theoretic*, level, things become more specific. This is the level at which we "tidy up" the notion by means of certain abstractions and idealisations. It is important to stress that there may be an element of decision here, in the sense that we choose to pick up certain strands from the pre-theoretic hodgepodge of ideas and *sharpen* the particular notion.[9] So, what the founding fathers of computability did, back in the 30s, was to pick out the strand of 'effective computation' (i.e., computation by following a sequential algorithm) and abstract away from limitations of time and space.

It is not clear whether there is also an element of choice about exactly what idealisations to employ when sharpening intuitive notions; when Church, Turing, et al., for example, idealised from matters of feasibility in order to formulate effective computation. After all, there are other notions of computability as well and it seems reasonable that a different mathematical community could have set, say, some fixed upper bounds to how fast a function grows, in order to be regarded as 'computable', such as to be calculable in at most exponential or hyperexponential

[9] I should emphasize from the outset that, by saying that there is an element of decision in the above procedure, I do *not* mean that the whole process of sharpening the concepts is an arbitrary one. Rather, I mean something along similar lines to Carnap's (1962) views on *explication*; namely that before we go on with our attempt to provide a satisfactory *explicatum* of the concept in hand, we need to make clear what is meant by the *explicandum*. That is, we need to provide explanations of what is and what is not the intended use of the intuitive concept to be explicated; what is intended to be included and what to be excluded. For example, by wanting to explicate 'truth', e.g. in a Tarskian way, we specify that we mean 'truth' as, say, used in the sense of 'correct', as applied to statements, and not as used in phrases such as 'true democracy', 'true love', etc.

time, etc. See Shapiro (2013) for arguments in favour of such a view. In all those cases, functions such as Ackermann's would not qualify as computable (and there *were* voices at the time arguing to that effect). On the other hand, Gödel seemed to hold that —at least in some cases— there is only one correct way of formalising an intuitive concept; we only need to gain "the correct perspective":

> If we begin with a vague intuitive concept, how can we find a sharp concept to correspond to it faithfully? The answer is that the sharp concept is there all along, only we did not perceive it clearly at first. This is similar to our perception of an animal first far away and then nearby. We had not perceived the sharp concept of mechanical procedures before Turing, who brought us to the right perspective. And then we do perceive clearly the sharp concept.
>
> If there is nothing sharp to begin with, it is hard to understand how, in many cases, a vague concept can uniquely determine a sharp one without even the slightest freedom of choice. (quoted in Wang 1997, 232-3)

We need not settle this issue here. But we will have some relevant discussion further on.

Finally, the third, *fully-theoretic*, level is when we have come to formulate rigorous and definite concepts, such as those of 'Turing computability', 'μ-recursiveness', etc. The concepts now in play are rigorously defined and precise, and little (if anything) is left to intuition. In our case, for example, it seems reasonable to assume that were a new function (on the natural numbers) to be discovered tomorrow, there would be a definitive answer whether it is, say, Turing-computable or not (even if it was extremely difficult in practice to find that answer).

So, in the case of computability, the Church-Turing thesis links the *extensions* of the sharpened notions at the second level (i.e., 'effective computation') with those of the rigorous mathematical concepts of the third level (e.g., 'recursive functions'). And what it does is to identify these extensions. And since effective computation is understood as computation by following an algorithm, the CTT can be seen as explicating the idea of 'algorithmic computation' by means of 'Turing computation', (or 'recursiveness', or whichever your preferred model is).

2.2.2 Three possible interpretations

Now, how are we to interpret the fact that although arguably all mathematicians would agree about the *proto*-theoretic characterisation of 'algorithm', they end up formalising it, at the third level, in incompatible ways, according to which mathematical area they come from? I suggest that there are three possible ways to see these apparently inconsistent practices.

A) The first possibility is that only one approach is in fact proper (or, acceptable). The proto-theoretical notion of ‘algorithm’ is as precise as it gets for an informal concept and, despite some vagueness in the *sense* of the term —what is meant by ‘small steps’ in an algorithm, for example?—, we should always pick out the same class of algorithmically computable functions, under any reasonable sharpening. This was the case with the numerical functions over the non-negative integers, and it’s also the case (for the most part) with the different models within the cluster of effective approximation. Thus, effective approximation gets it right, whereas BSS/Real-RAM or other unrealistic models of computation actually miss the point.

B) The second possibility is to say that we in fact use the notion of ‘algorithm’ in mathematics in more than one way. To make this idea more precise, let’s say that a concept is *poly-vague* if our informal talk about it fails to pick out a single mathematical “natural kind”.[10] A poly-vague term, then, can legitimately be disambiguated in more than one way, though consistently with the implicit rules we’d mastered for applying the concept to ordinary cases.

So, following that route, we can say that in the traditions of numerical analysis and geometry the term ‘algorithm’ picks out different classes of routines from those of computer science (though each class might be precisely bounded).

C) The third possibility is that, although already a *proto*-theoretical concept (that is, some theoretic tidying has already taken place), ‘algorithm’ is a concept with *open texture*. This means that the concept itself has some open-ended character in the sense that the so far established use of the language is not adequate to delimit it in all possible directions.

The term ‘open texture’ is due to Waismann (1945), who, at the time, was arguing against *verificationism*, on the grounds that most (though not all) empirical concepts cannot be exhaustively and precisely defined; there can always exist some unforeseen situations or instances falling under their extension. This, as a fact, prevents us from conclusively verifying most of our empirical statements. Assuming that a term is considered defined when the sort of situations in which it is to be used are defined, empirical concepts can never be completely defined, due to an open horizon of possible situations that any empirical description might leave out. Nevertheless, for Waismann, mathematical terms do not suffer from any similar incompleteness of definition, because a definition of, say, a geometrical term, such as a triangle, in a sense includes already all sorts of situations in which it can be used. The description of the term is already complete.

Despite that, Stewart Shapiro, in two related articles (2006; 2013), has argued that open texture can be a property of some mathematical terms as well; ‘number’ and ‘computable’, for example, being cases in point.

[10]I borrow the term and the idea of ‘poly-vagueness’ from Smith (2013).

Now, in terms of the three-level schema, described in the previous section (2.2.1), it is safe to assume that *pre*-theoretic concepts may exhibit open texture. The different notions of 'computation' not excluded from the *pre*-theoretic idea is a good example of this. What is more contentious, though, is the claim that the *proto*-theoretic concept of 'algorithm' may still have some open texture too.

Supporters of all three interpretations (explicitly or implicitly) can be found both in philosophy and mathematics/Computer Science. But before we attempt a more thorough discussion about their plausibility, we need a more 'precise' formulation of the informal idea of 'algorithm'.

2.2.3 Algorithm: the informal concept

Although informal, 'algorithm' has always been taken as a clear and unproblematic notion. Virtually all mathematicians would agree on something like the following features as being the essential ones:[11]

1. An algorithm is a general step-by-step procedure, prescribing a sequence of operations for solving a type of problem. It must be expressed as a set of instructions of *finite size*.

2. An algorithm has a set (perhaps empty) of inputs and a (set of) output(s).

3. For any given input, the computation is carried out in a discrete stepwise fashion (that is, without use of any continuous methods or analog devices). Alternatively put, an algorithm proceeds in discrete time, so that at every given moment the state of the computation is obtained from the state at the previous moment of time.

4. For any given input, the computation is carried out deterministically, without resort to any random methods. The computation state at any given step/moment is *uniquely* determined by the state in the preceding step/time and the list of instructions.

5. The list of instructions that make up the algorithm are to be followed by a computing agent (human or otherwise) which carries out the computation.

6. Each step of an algorithm must be specified to the smallest detail, precisely and unambiguously, such that no acumen or ingenuity or any semantic interpretation is required by the computing agent. Steps should be of a bounded complexity.

[11]The characterisation I provide here is distilled by relevant descriptions in the classic works of Knuth (1997), Rogers (1987), and Malc'ev (1970).

Computations possessing feature (1) are sometimes called 'sequential-time' (as opposed, e.g., to parallel or distributed computations). Computations with features 1 and 6 are sometimes called 'sequential'. Sequential algorithms are a subspecies of sequential-time ones.

The above 1-6 features are almost universally accepted as necessary. There is also the requirement of *finiteness*:

7. An algorithm must always terminate after a finite number of steps.

But, although many texts explicitly pose this requirement (Rogers, 1987; Knuth, 1997), some authors may accept non-terminating procedures satisfying the above criteria as algorithms too.[12] The concern with finiteness arose in the context of effectively computing and is explicitly expressed in Hilbert's 10th problem, posed in 1900 at the International Congress of Mathematicians in Paris: "Given a Diophantine equation [..] devise a process according to which it can be determined *in a finite number of operations* whether the equation is solvable in rational integers" (emphasis added). It is trivial, however, that not all processes which clearly are algorithmic can be carried out in a finite number of steps; consider the calculation of the quotient of two incommensurable numbers, for example.

In practice, though, the process always terminates when it reaches some previously agreed stage. Additionally, for every function which qualifies as computable, in the sense of the existence of an algorithm for computing its values, the corresponding algorithm must, obviously, be terminating.

2.2.4 The problem reformulated

Given the above features, a main question pursued in this chapter is this:

[12]For example, (Hermes, 1969, 2):

> There are terminating algorithms, whereas other algorithms can be continued as long as we like. The Euclidean algorithm for the determination of the greatest common divisor of two numbers terminates; [..] The well-known algorithm of the computation of the square root of a natural number given in decimal notation does not, in general, terminate. We can continue with the algorithm as long as we like, and we obtain further and further decimal fractions as closer approximations to the root.

Or, (Gurevich, 2015, 189):

> In general algorithms perform tasks, and computing functions is a rather special class of tasks. Note in this connection that, for some useful algorithms, non-termination is a blessing, rather than a curse. Consider for example an algorithm that opens and closes the gates of a railroad crossing.

The finiteness condition needs also be adjusted accordingly in the case of Type-2 Theory of Effectivity, as we already saw in sect.1.3.2.

Does the above informal characterisation have enough shape to pick out always a precisely bounded class of routines/computable functions?[13]

If the answer is 'yes', then something like the (A) possibility from the previous section (2.2.2) must be the case; if 'no', then either (B) or (C).

Of course, the case of classical computability theory gives strong evidence in favour of the first option. The informal notion has throughout the history of mathematics been taken as clear, precise, and unproblematic, and when the need for formalisation arose —in order to prove non-existence results about algorithms—, all formal models (all attempted *explicata*, that is) turned out to be equivalent. Hence, despite any 'vagueness' in the informal characterization of 'algorithms', the latter does have enough shape to pick out always, under any reasonable sharpening, the same class of routines/functions, when we are in the domain of the non-negative integers. And this is also in accord with Gödel's view in the quoted passage (sec.2.2.1).

Nevertheless, crucially, computations do not only pertain to denumerable domains, and we saw areas of mathematics with the notion of algorithm at their heart already. So, before attempting an answer to the question above, we need to examine the concept of 'algorithm' as occurs in such domains too. We attempt an investigation within both a historical and theoretical context.

2.3 Historical context

The word 'algorithm' comes from the medieval term 'algorism'. The latter is derived from 'Algoritmi', the latinised version of the name of the Persian Muslim mathematician al-Khwārizmī (c.780–850AD) who lived and worked in the House of Wisdom in Baghdad.[14] Al-Khwārizmī's work is, among others, responsible for the spread of the decimal positional number system across Europe, mainly through the translations of his books on (a) arithmetic: *Kitāb al-Jam' wat-Tafrīq bi-Ḥisāb al-Hind* (*The Book of Addition and Subtraction According to the Hindu Calculation*) and (b) algebra, that is, on finding the positive roots of quadratic equations: *Al-kitāb al-mukhtaṣar fī ḥisāb al-ğabr wag'l-muqābala* (*The Compendious Book on Calculation by Completion and Balancing*).[15] From the time that al-Khwārizmī's work became known in Europe (around the 12th century), a conflict between the advantages of using the new positional

[13]By 'computable' here, we only refer to those functions whose value can be computed by following an algorithm, in the above sense.

[14]The shift from 'algorism' to 'algorithm' is, according to one interpretation, due to a mistaken etymological rooting when European scholars had lost track of the correct root and attributed a Greek root connected possibly with the Greek word 'arithmos' (number). Another interpretation has been that 'algorism' comes from 'Algorismi' and 'algorithm' from 'Algoritmi', both latinised versions of 'al-Khwārizmī'.

[15]The modern word 'algebra' stems from the word 'al-ğabr' in the title of this book.

notation calculation methods over the older abacus and counting table methods appeared in the Latin texts of the Middle Ages. See, figure 2.1. Since the new methods were referred to as 'algorisms' or 'algorismus', the word itself came to refer to the use of specific routine arithmetic procedures.[16] In this context, clearly, the concept referred to a process of symbol manipulation.

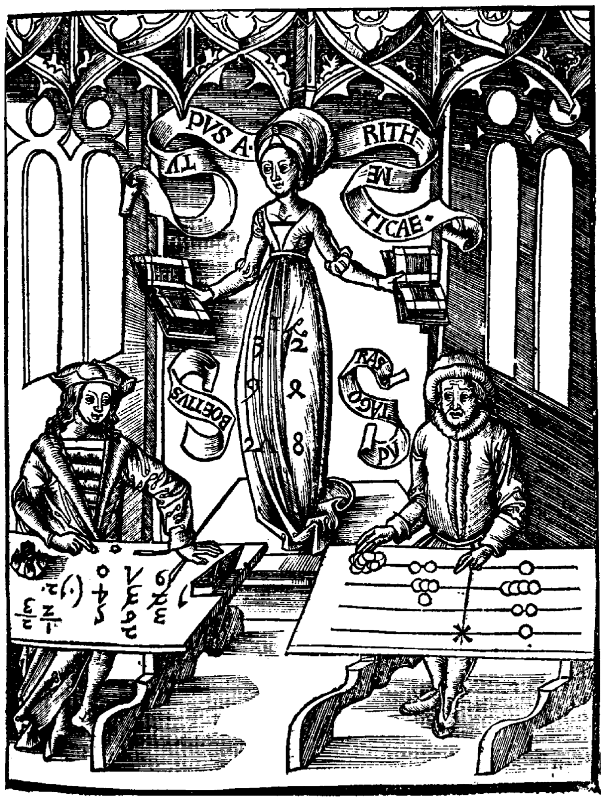


Figure 2.1: Calculating-Table by Gregor Reisch: *Margarita Philosophica*, 1503. The woodcut shows *Arithmeticae* instructing an algorist and an abacist (represented as Boethius and Pythagoras) during a competition.

Later on, however, the meaning of the term became extended, so that it came to signify any given routine of mechanical calculation. Thus, d'Alembert wrote in the *Encyclopédie* about the term 'algorithm':

> Arab term, used by several authors [..] to mean the practice of algebra. It is also sometimes taken to mean arithmetic by digits [..] The same word is taken to mean, in general, the method and notation *of all types of calculation*. In this sense, we say the algorithm of the integral calculus, the algorithm of the exponential calculus, the algorithm of sines, etc. [emphasis added].[17]

[16]See, also, Chabert (1999).

[17]I take the quotation and the translation from Chabert (1999, 2).

This generalised meaning has been around in mathematics for a long time, and in parallel with the more restricted sense of symbolic manipulation. The latter has been more prominent within the tradition that led to the development of logic. More specifically, besides al-Khwārizmī, Gottfried Wilhelm Leibniz's idea of a *characteristica universalis* is based on ultimately reducing computations to manipulations over an alphabet as well:

> In philosophy I found some means to do, what Descartes and others did via Algebra and Analysis in Arithmetic and Geometry, in all sciences by a combinatorial calculus [..] By this, all composed notions of the world are to few simple parts as their Alphabet, and from the combination of such alphabet a way is opened to find again all things, including their truths, and whatever can by found about them, with a systematic method in due time. (Letter to Duke Johann Friedrich of Braunschweig-Lüneburg).[18]

Furthermore, Charles Babbage, in his (1826), wrote about the importance of symbolic notation, so that algebraic symbols could be used mechanically in mathematical reasoning (as opposed, e.g., to the less safe geometrical reasoning). George Boole applied methods from symbolic algebra to logic, and Charles S. Peirce extended Boole's work to the theory of de Morgan relations, introducing also the method of truth tables, which arguably is symbolic as well. This tradition culminates in Turing's work which is about symbolic computation too.

On the other hand, the use of 'algorithm' in the more general meaning of 'calculation method' can be found in several mathematical areas, spanning through the whole history of mathematics. For example, most of Sumerian, Babylonian, and Egyptian mathematics we know of today seems algorithmic in nature; methods for multiplying, dividing, calculating inverses, etc.[19] But these are cases of denumerable domains, so they don't help illuminate our investigation here. Thus, it might be more promising to focus on uncountable domains, that is, calculation methods over $\mathbb{R}$.

Geometry: Greek mathematics was a huge step towards the notion of proof and the development of mathematical theories. Euclid's *Elements* is largely devoted to proofs of certain theorems within Euclid's axiomatic system. But, a good deal is computational too, in the sense of showing that certain constructions exist and can be carried out. And, in the words of two mathematicians: "[t]he Euclidean construction satisfies all of the requirements of an algorithm: it is unambiguous, correct, and terminating." (Preparata and Shamos, 1985, 1). Another important characteristic of Euclidean constructions is that they specify a universe of

[18]Quoted and translated in Thomas (2015, 31-32).

[19]Does it really make sense to distinguish these cases from the symbolic computations described in the previous paragraph? Perhaps not really, since we're still in denumerable domains; we discuss this more later in sec.(2.6).

entities on which the computor operates (points, lines, etc.), a collection of permitted instruments (ruler and compass) and a collection of permitted operations on them (produce the line through two existing points, produce the circle through one point with centre at another point, iterations of those, etc.). Taking the permitted operations as primitive, it then makes sense to say —in somewhat anachronistic terms— that a main computational concern in Euclidean geometry was the closure of the Euclidean entities computed (constructed) under the Euclidean primitives and composition. Nevertheless, other "computational models" were considered too, and many geometers, along the history of the subject, were concerned with investigating the "computational power" of, e.g., compass-only models, or ruler-and-scale models (Hilbert), etc.

Algebra: A big part of algebra has also been concerned with developing algorithms for computing solutions of (systems of) polynomial equations with real or complex coefficients. Babylonians, Egyptians, Diophantus, al-Khwārizmī, Niccolò Ludovico, Ferrari Tartaglia, Gerolamo Cardano et al. are all major stages and figures in the long quest for algorithms for calculating analytic solutions. Recall, for example, the quadratic formula[20]

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

for computing the real roots of the quadratic equation $ax^2 + bx + c = 0$, whenever $b^2 - 4ac \geqslant 0$ and $a \neq 0$. The cubic and quartic equations have similar methods, but for centuries no such method could be found for the quintic. And it was only after the exact requirements were precisely specified —or, again in somewhat anachronistic terms: what the permitted primitive operations are— (viz., a solution obtained by means only of the coefficients and a finite number of arithmetical operations and n^{th} roots, for any n), that Abel was able to show that no such computation algorithm exists (Abel's Impossibility Theorem). An additional example of computational method in algebra is Gaussian elimination; a method for computing solutions of systems of linear equations.

Numerical analysis: More often than not, solutions of equations cannot be computed exactly. In those cases, root-finding algorithms are developed for computing approximate (numerical) solutions. As we have already discussed, the BSS model purports to formalise this practice; nevertheless, the subject goes several centuries back in mathematical history. Typical algorithmic methods here include the bisection method, liner interpolation, Newton's method, etc. (for

[20]One should not think that our characterization of the quadratic formula as an algorithm is just loose talk. The same assertion can even be found in computability textbooks; for one, see Epstein and Carnielli (2008, 63).

computing approximations to roots of functions), and Euler's method (for computing approximations to ordinary differential equations, with given initial value).

Analysis and calculus: Analysis has also had an algorithmic flavour in the past (i.e., before its developments in the 19th century onwards). In the early 1600's, Cavalieri, Fermat, Pascal et. al. developed methods for computing areas of regions bounded by curves (see, Avigad and Brattka 2014). Even the very name 'calculus' indicates the algorithmic nature (at least in the early stages) of the field. The word 'calculus' is of Latin origin, meaning originally 'small pebble' as those used on abaci ('calculation', of course, stems from the same root). As a toy example, the high school routine for differentiating polynomials is typically purely mechanical in nature[21] (Rogers, 1987).

The argument from history and practice

It becomes apparent then that the history and practice of mathematics has been significantly concerned with the development of mechanical routines for computing solutions to (families of) problems.[22] But some of the above examples of methods in fact may not be of the sort that admits of implementation by a Turing machine; certain functions computed by ruler-and-compass constructions, for example (see, Gurevich, 2014, 4). Additional examples could be Gaussian elimination, bisection algorithms, or even the simple decision problem to decide whether a quadratic equation $ax^2 + bx + c = 0$ has real roots, by determining whether $b^2 - 4ac \geqslant 0$. All these cases involve arithmetical operations and equality comparisons between real numbers, and so they are not always computable; if the left-hand part in the latter instance, for example, is very close to zero.[23] But, even Euclid's algorithm, which has been applied —in theory and practice— to the lengths of segments of a straight line, may allow the computation of a partial function[24] which is not Turing computable; this is so because arbitrary lengths cannot in general be put on the tape of a Turing machine[25] (Gurevich, 2014, 4).

In actual practice, of course, we are able and do *approximate* such functions and routines.

[21]This is not to say, of course, that all functions and processes used in analysis, even before the 19th century were computable in the modern sense of computable analysis. Euler, for example, did use piecewise functions which are discontinuous and so non (Turing- or TTE-)computable.

[22]Even Chabert (1999) for example, although a book on the history of algorithms, is to a great extent concerned with the development of methods used in numerical analysis.

[23]The equality relation '=' is not Turing computable. But, even certain arithmetical operations may not be Turing computable either. For instance, there's no Type-2 Turing machine, in the way specified earlier (sect.1.3.2), that computes the product of an infinite decimal fraction (say, 0.333...) and 3 (see Weihrauch, 2000, 18).

[24]The function here is the multivariable $d = gcd(a, b)$ and it's partial because if the two lengths a and b are incommensurable, then the algorithm does not terminate.

[25]As it should be clear by now, in this context, I mean 'Turing computable' in the broadest sense, including both Type-1 and Type-2 Turing machines.

Approximation nevertheless *is* different from exact computation.[26]

Now, if the existence of the broader generalised meaning of ‘algorithm’ as ‘calculation method’ in mathematics is granted, we could use this as evidence against the (A) interpretation from above, and in favour of one of (B) or (C). That is, against the view that the informal notion of ‘algorithm’ is as precise as it gets, and so any explicatory formalisation, in a given domain of application, ought to pick out the same precise class of routines as its extension. According to this line of thought then, the history and practice of mathematics indicates that Turing computability is not enough to capture all instances of the notion of ‘algorithmic computability’. Rather, it shows that either the informal notion is actually used in more than one way in mathematics —the interpretation (B), that is—, or that it is an open-ended term, a term exhibiting open texture —that is, the third interpretation (C).

Nevertheless, someone could object that most of the above talk about algorithms is not really literal. In other words, the broader sense of ‘algorithm’ as ‘calculation method’ is rather loose talk by mathematicians, convenient and harmless for most practical purposes, even if not necessarily theoretically justified.[27] In this respect, we should not really speak about algorithms in numerical analysis or geometry but we should preserve other terms for generally referring to such processes, such as ‘method’, ‘construction’, ‘general procedure’, etc. Only when a straightforward way is available of how to fix the details and turn the routine to one that proceeds at each step in an unambiguous manner, so that it is translatable to a Type-2 Turing machine program, only those procedures would be worthy of the name ‘algorithm’. Note here, though, that most methods presented as algorithms in mathematical practice (from Gaussian elimination, to computing the integral of rational functions by decomposing to partial fractions, etc.) are in general too vaguely expressed to reach the required standard of exactness. But even ignoring that —since it seems clear that in such cases any missing instructions can, in principle at least, be supplemented to make a complete specification— such a restrictive use of the concept would

[26]Actual computers generally work with floating-point arithmetic. Floating-point numbers make up a finite set, with though some unexpected properties: addition is generally not associative or distributive, the set is not closed under addition and it is neither a field, nor a ring or a group. Nevertheless, significant effort has been put in to develop satisfactory representation systems so that the accumulation of round-off and representation errors will be held at a minimum. IEEE Standard 754 double-precision is such a system guaranteeing some desirable properties. For more information, see, for example, Corless and Fillion (2013).

[27]Here’s one more example, from a numerical analysis textbook, in a context of talking about numerical properties of algorithms:

> There are many variants on the definition of an algorithm in the literature, and we will use the term loosely here. [..] we will count as algorithms methods that may fail to return the correct answer, or perhaps fail to return at all, and sometimes the method may be designed to use random numbers, thus failing to be deterministic. The key point for us is that the algorithms allow us to do computation with satisfactory results... (Corless and Fillion, 2013, 29)

rule out cases so intuitively unproblematic as even calculating the product 0.333... $\times$ 3 (see, fn.23).

So, is the above objection adequate to block the argument from the history and practice of mathematics? That is, is the appeal to just loose but harmless talk by mathematicians enough to block the argument that the history and practice give more support to the (B) and (C) interpretations than to (A)? I think that in order to answer this, we also need to examine whether dispensing with the generalised meaning of 'algorithm' could be theoretically motivated too. This is the subject of the next section.

2.4 Theoretical context

So we need to examine whether the proto-theoretical idea of 'algorithm' is indeed as clear as it gets. Or, in Gödel's words (p.21), whether 'mechanical procedure' is a sharp concept which has been there all along. If so, then, the fact that —despite the apparent vagueness in the informal characterisation of 'algorithm'— *every reasonable sharpening* gives rise to extensionally equivalent formalisations should also arise with respect to computability over the reals; meaning that BSS/Real-RAM models just miss the point.

To this end, it seems useful to examine what 'reasonable sharpening' amounts to. I attempt to make this idea more precise by connecting it to *conceptual analysis*, in the sense of the term as first articulated by Demopoulos (2000), with respect to mathematical frameworks, and further extended by DiSalle (2012) to physical theories. That is, in the sense of "recovering a central feature of a concept in use by revealing the assumptions on which our use of the concept depends" (Demopoulos, 2000, 220). Or:

> [C]onceptual analysis finds the interpretation of a theoretical concept by an investigation of the presuppositions under which it is used in some practice of scientific reasoning [..] and the roles that they play in the theoretical framework as a whole.[28] (DiSalle, 2012, 13)

However, this is not to say that all cases where logicians offered models explicating the notion of 'effective computation' were instances of conceptual analysis. Recall, for example, how Church's (1936) definition of 'effectivity' as 'λ-definability' famously failed to convince Gödel (as well as others). I here regard only specific cases of foundational work as instances of conceptual analyses; namely, the work of Turing, Kolmogorov and Uspensky, Moschovakis, and Gurevich.

[28]Two examples of such conceptual analyses are (a) Frege's analysis of numerical identity by characterising Hume's principle as its implicit basis (Demopoulos, 2000) and (b) Poincaré's (1905) analysis of our notion of space by identifying the free mobility of rigid bodies as its conceptual base (DiSalle, 2012).

2.4.1 Turing

In contrast to most modern presentations, Turing in (1936) did not focus on computation over denumerable domains only, but over all reals. Interestingly, he does not use the term ‘algorithm’ in his analysis, although he was primarily interested in *processes* instead of *functions*. Nevertheless, he focuses on the restricted sense of calculation, that is, symbol processing (1936, 249):[29]

> The real question at issue is “What are the possible processes which can be carried out in computing a number?”

And:

> Computing is normally done by writing certain symbols on paper.

Turing was also interested in analysing computation as the lowest level, that is, the level of the most elementary operations:

> Let us imagine the operations performed by the computer to be split up into “simple operations” which are so elementary that it is not easy to imagine them further divided.

Crucially for our purposes here, by restricting the scope of his analysis to effective processes implemented by symbolic manipulations, Turing’s analysis is not affected by the examples of the non-Turing implementable algorithms we have discussed (ruler-and-compass methods, Euclid’s algorithm applied to lengths of segments of a straight line, etc.). But, his choice to restrict the computer’s working space to a one-dimensional tape, and the permitted operations to the most elementary ones, did not make completely obvious that his model could capture all examples of algorithms (though it did capture all effectively computable functions). A. Kolmogorov saw that and set out, together with his student V. Uspensky, to give a more general account, about what it is to follow an algorithm.

2.4.2 Kolmogorov and Uspensky

As mentioned in the footnote of the title of their work, the main purpose in Kolmogorov and Uspenskii (1963) was to give the broadest story possible, about the notions of ‘algorithm’ and ‘computable function’. The rationale was to examine these notions from the point of view of mathematicians, and make it obvious that “there is no concealed possibility of extending the

[29]The same was true of Post’s independent and strikingly similar analysis, in the same year (1936).

range of [these] notion[s]", beyond what is captured by Turing computability and the other equivalent models.

Space limitations do not allow a satisfactory presentation of the K&U account of computation. But what is of interest for us is what K&U take as the essential aspects of an algorithm. Besides a symbolic representation ("[w]ithout fixing a standard way of writing numbers, to speak of the algorithm computing $m = \phi(n)$ from n would not make sense"[30] [p.218, fn.2]), an algorithm for computing $m = \phi(n)$ requires:

> ...the existence of a uniquely determined sequence of operations "transforming" [..] the value n into the value $m = \phi(n)$.

And:

> ...division of the computing process into elementary steps of *limited complexity* (p.219).

Although these constraints may initially seem very similar to Turing's, our interpretation is that they constitute an important step towards the generalisation of the concept. Where Turing would only allow the most elementary basic operations as steps, K&U allowed for broader steps. Here is a very brief and informal summary of the KU model.

A KU computing agent computes over symbols. The work-space can be understood, instead of a tape, as a directed graph, whose vertices correspond to the Turing machine squares. Each vertex is connected to a bounded number of neighbouring ones; the number being fixed in advance, but possibly different for different algorithms. All vertices, and arrows connecting them, have a colour from a fixed finite palette, so that all arrows to and from a vertex have different colours.[31]

At every stage of a particular computation, only a bounded number of nodes —a certain patch— is active. The bounded size of the active patch is fixed in advance but can differ for different algorithms. For some fixed number k of edges, it is the region of the workspace that contains all vertices being reachable from some certain focal vertex by a directed path of length not greater than k. k is called the *locality constant* or *radius of the active part*, and any property depending only upon k-neighbourhoods is called 'k-local'. The next step of the computation is always restricted within the active patch. This captures the intuition that the attention span of the computing agent is limited and fixed (the agent's cognitive capacities do not change during a particular computation). Any actions must also be *k-local*.

[30] All quotations and page numbers are from the English translation (Kolmogorov and Uspenskii, 1963).

[31] K&U's original formulation posits undirected graphs as workspaces, in a way that for every edge $\langle x, y\rangle$ between two vertices, there's also a $\langle y, x\rangle$, of the same colour. But later work has shown both formulations to be equivalent, in the sense that machines from both kinds can be simulated by machines from the other kind.

A single computing step consists of a k-local operation. This means replacing vertices within a bounded part of the active area as well as edges with a new collection (of bounded size) of vertices and edges. Thus, with each computational step —i.e., a local operation transforming the state of the algorithm— not only the vertices but also the topology of the workspace itself can change locally.

An algorithmic process, according to Kolmogorov, is divided into steps of bounded complexity, where each step is an immediate transformation Ω of the current state S into a (uniquely determined) $S^* = \Omega(S)$, based only on information about the bounded active part of S and affecting only that. The algorithmic process is continued as a sequence of iterations of Ω over states S_i: First, the initial state S_0 is transformed into $S_1 = \Omega(S_0)$, then S_1 transforms into $S_2 = \Omega(S_1)$, ..., S_n into $S_{n+1} = \Omega(S_n)$, and so on, until either a next step is impossible (i.e., $\Omega(S_r)$ is undefined for r) or a signal indicating a reached solution is obtained. But it's also possible that the sequence $\langle S_i \rangle$ is infinite.

The initial state is the *input*, belonging to some set (the domain of the algorithm), and the final state (if it exists) is the *output*. The immediate transformation Ω, which is iterated many times, is called 'operator of immediate transformation', and is determined by *a finite list of k-local actions*. When in a state S_i, the KU-computer looks at the list of actions to find the (possibly existing) unique one that is applicable to S_i.

An algorithm $\mathcal{A}$ has a set X of all allowed inputs and a set Y of allowed results. The algorithm can be applied to any input $x \in X$; the subset of X with inputs for which $\mathcal{A}$ terminates with a result, is the *domain* of the algorithm. Any algorithm determines a (total) function defined on its domain, whose value is the output $y \in Y$; this is the function *computed* by $\mathcal{A}$.

2.4.3 Turing's and K&U's conceptualisations

KU-computable functions do not make up a wider class than the class of Turing computable functions. KU-algorithms can compute up to partial recursive functions (if we consider as a function's domain the whole set of allowed inputs). Nevertheless, the K&U's intention was to provide a more general analysis of what an algorithmic procedure is. It is not to say that any algorithm can be simulated by, or reduced to, a KU-algorithm. Rather, it is the stronger claim, that any algorithm just *is* a KU-algorithm.[32] Following Gurevich (1993), we can call 'Kolmogorov-Uspensky thesis' the following claim: *any algorithmic computation performed by means of only local actions at a time is a computation of a KU machine.*

As we saw, K&U's analysis admits of a broader notion of 'computational step', since a

[32]"We are convinced that an arbitrary algorithm process satisfies our definition of an algorithm. [.. I]t is not a question here of the reducibility of any algorithm to an algorithm in the sense of our definition [..] but rather that any algorithm is essentially subsumed under the proposed definition." (p.231).

k-local action can in fact involve changing huge chunks of workspace (how big can we take the locality constant to be, in a given algorithm?). Therefore, the KU analysis is closer to the broader meaning of 'algorithm' as used in common-or-garden mathematical parlance, and in accordance with the authors' goals. And, recall that, attempts to formalise the latter notion include models which recognise arithmetical operations between any two numbers as unit steps. Nevertheless, KU-algorithms perform still at a very low level of abstraction, since steps involving non-local transformations of information are not of KU-type. Thus, Markov's normal algorithms or random access machines are not KU-type models (Uspensky and Semenov, 1993, 19). From that it follows that BSS-algorithms are *a fortriori* not KU-algorithms.

Conceptual Analysis in Turing and KU: Now, given the above discussion, could we attempt to identify the implicit basis for the conceptual framework of algorithmic computation, in the sense of Demopoulos (2000) and DiSalle (2012)? One could argue that a fundamental assumption is that computation is always symbolic, since all analyses took the existence of a given alphabet as essential.

Nevertheless, I think that this would be misleading. Rather, the symbolic constraint should be understood as part of the theoretic tidying of the intuitive concept that takes place when we move from the pre-theoretic to the proto-theoretic level of conceptualisation, as discussed above (sec.2.2.1). Although these days we are so much used to computation by digital computers that the symbolic assumption may not seem as much a restriction as essential, it is in fact a choice of picking out one of the various strands of (algorithmic) computation. One can think, for example, cases of analog computation of a function, such as use of a planimeter,[33] or —if one counters that analog computation is not algorithmic—, use of ruler and compass to compute certain numbers (Plouffe, 1998), Euclid's algorithm applied to lengths, etc.[34]

So what can be seen as implicit principles, analytic of all *symbolic computation* frameworks, in the sense of posing constrains on any conceptual possibilities? I suggest something along the following lines: computation involves step progression in time, in a way that between any two steps of the computational process there must intervene only a *finite* number of actions. Although this is more explicitly expressed in K&U's analysis, by means of the locality constant (any sum of locality constants, however big, will always be finite), it can be found implicitly underlying Turing's non-computability results, such as the unsolvability of the *satisfactoriness* problem. This was the problem of deciding whether a given integer *n* encodes the description of a Turing program that, starting from an empty tape, will eventually halt with only symbols

[33]See chapter 4 for a discussion on analog computation.

[34]Thus, it is, in my opinion, strictly speaking false when some authors make assertions like the following: "Turing recognized that any algorithm is essentially a manipulation of symbols." (Aberth, 1980, 6).

of a certain kind on its tape (so *n* describes a *circular* machine and is *unsatisfactory*) or it will keep printing digits of the same certain kind forever (so *n* describes a *circle-free* machine and is *satisfactory*). In the unsolvability of this problem —as well as of the *Entscheidungsproblem* and the halting problem— lies a diagonal argument, which can be seen as capturing the fundamental fact that an agent trying to step-by-step compute these problems is doomed at some point to fall into some infinity of actions before completion of some certain step (which happens, e.g., when a Universal Turing Machine tries to list every binary sequence and ends up simulating its previous behaviour forever). In other words, a computing agent can only tame at most one level of infinity: it can keep a computation going forever (e.g., one computing digits of π). But —to put it fancifully— it cannot perform computations during which some rabbit hole of infinity lurks in some step, further down the road.

2.4.4 The idealisations strike back: Moschovakis and Gurevich

So far everything seems to suggest that the (A) interpretation about the concept of 'algorithm' is the most plausible one: although informal and vaguely characterised, 'algorithm' is actually as precise as it gets and admits of only co-extensive explications. Any Real-RAM/BSS-like models, which take arithmetical operations between any reals at unit cost, fall out of the conceptual framework we just described, since adding, for example, $\sqrt{2} + \pi$ actually means going at least two levels deep in infinity.

Nevertheless, a serious objection from the practice of mathematics remains. The way mathematicians use the term to refer to computational methods is *not* always limited to *symbolic computations*. Computations are indeed necessarily symbolic when it comes to actual implementations by a physical computer (machine, human, or what have you)[35] but there is nothing in the way that algorithms are constructed by mathematicians in the first place to specify why and how this should be so. For example, methods for deciding whether a polynomial equation (e.g., the method for the quadratic in sec.2.3) has real roots are understood as algorithms, but there's nothing in the way they are developed specifying how they are meant to be implemented. Even more so, if we were to write down, say, the algorithm for the quadratic, we would formulate it quite differently for different computational models/languages. Differently put, the way mathematicians talk about algorithms is as if the latter are meant to operate over *types*, whereas within logic and computability theory, algorithms are from the outset constructed with *tokens* (read: symbolic representations) in mind.

[35]Of course, I restrict my attention here to digital computation, excluding other computing paradigms, such as analog computation, since we are already at a proto-theoretical level of analysis here. I'll discuss computation in a broader sense, in chapter 4.

Yiannis Moschovakis saw this and based his foundational work on algorithms —during the late 80's and 90's— on the idea that 'algorithm' is an abstract entity, susceptible to various implementations, within different computation models. The idea has succinctly been expressed in an interview (1997) for the Greek edition of the *Quantum* magazine, with Sturm's algorithm as a particular example.[36]

> Sturm's algorithm is usually defined over the class of all polynomials with arbitrary *real* coefficients, and there is nothing in its description or analysis of its implementation that requires those coefficients be integers or rationals. If we want to *implement* the algorithm to some actual computer or abstract Turing machine, then, of course, we need to approximate the real coefficients by means of rationals, and choose certain symbolic representation. However, there are various ways to go about these choices and none of these is essentially included in *Sturm's algorithm*.[37]

Moschovakis's point then is that we have *one* Sturm algorithm with *many* implementations in different computation models, instead of distinct "Sturm's algorithms" with different properties in the different computation models. Thus, as an abstract object, an algorithm like Sturm's operates on the exact real numbers, whereas the various implementations manipulate symbolic representations of their approximations. (Cf. the long quote in fn.3).

Moschovakis's exact formalisation of algorithms draws on the theory of recursive equations, identifying algorithms with *recursors*; that is, monotone operators over partial functions whose least fixed point includes the function computed by the algorithm. We do not need to go into the details of this, highly technical, work. One can see his (1984; 1998; 2001; 2008). What is important for us here is the "official claim", this time by a logician, that there is a broader understanding of 'algorithms' by mathematicians, as abstract mathematical entities, non-symbolic and non-syntactic:

> Finally, I should mention—and dismiss outright—various, vague suggestions in computer science literature that *algorithms are syntactic objects*, e.g., *programs*. [..] In the absence of a precise semantics, Pascal programs are just meaningless scribbles; to read them as algorithms, we must first interpret the language—and it is then the *meanings* attached to programs by this interpretation which are the algorithms, not the programs themselves (Moschovakis, 1998, 3.6; emphasis in original).

[36]Sturm's algorithm is a method for determining the number of real roots of a polynomial $a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x^1 + a_0$ with real coefficients in any given interval on the real line.

[37]Interview in *Quantum* (Greek), Vol.4 (4), 1997. Accessed from Y. Moschovakis's webpage [Feb, 2018] http://www.math.ucla.edu/~ynm/. The translation from Greek is mine.

Under this view then, Moschovakis sets out to provide a set-theoretic foundation for the theory of algorithms; a framework within which set-theoretic constructions *model* (*faithfully represent*) their mathematical properties. Thus, *recursors* model the mathematical structure of algorithms, much like Dedekind cuts model (or "define") the real numbers.

The view of algorithms as abstract entities was adopted by Yuri Gurevich as well, in a series of articles trying to define the notion. Gurevich follows the axiomatic route to founding the theory of (sequential-time) algorithms. An algorithm is a state transition system that starts in an initial state and transits from one state to the next until, if ever, it halts or breaks (Gurevich, 2014).

Gurevich formulates the following three axioms, in order to capture these features:[38]

1. Sequential time Any algorithm $\mathcal{A}$ is associated with a non-empty collection $S(\mathcal{A})$ of states, a sub-collection $I(\mathcal{A}) \subseteq S(\mathcal{A})$ of initial states and a (possibly partial) state transition map $\tau_{\mathcal{A}} : S(\mathcal{A}) \rightarrow S(\mathcal{A})$.

2. Abstract state The states of an algorithm $\mathcal{A}$ can be *faithfully represented* by first-order structures (that is, non-empty base sets equipped with relations and operations), all of the same finite vocabulary, in such a way that

- $\tau_{\mathcal{A}}$ does not change the base set of the state,
- collections $S(\mathcal{A})$ and $I(\mathcal{A})$ are closed under isomorphisms,
- any isomorphism from a state X to a state Y is also an isomorphism from $\tau_{\mathcal{A}}(X)$ to $\tau_{\mathcal{A}}(Y)$.

3. Bounded exploration There exists a finite set T of terms (or expressions) in the vocabulary of $\mathcal{A}$, such that $\Delta(X) = \Delta(Y)$ whenever every term $t \in T$ has the same value in any two states X, Y of $\mathcal{A}$.

$\Delta(X)$ is the set of updates from X to $\tau_{\mathcal{A}}(X)$, solely dependent on the results of an exploration within a bounded "active zone" of the state X. This is meant to formalise Kolmogorov's locality constraint, which *Gurevich takes as constitutive of the notion of 'algorithm'*.

More precisely, a *location* l of a state X is given by a j-ary function F and a j-tuple $\vec{a} = (a_1, ...a_j)$ of elements of X. F is a vocabulary function, and its value $F(\vec{a})$ is the (new) content of l. Consider a typical algorithm (e.g., Euclid's) and assume that an instruction asks for the replacement of the current content a_i of l with b. This would then be an *update* of X,

[38]Reproduced here from Blass and Gurevich (2003) and Gurevich (2015).

denoted (l, b). The *set of all updates* (i.e., the collection of all equations $F(\vec{a}) = b$), during a given algorithm step, is $\Delta(X)$.

As an example, assume Euclid's algorithm again, and a state X in which the parameters a, b have values 6 and 9, respectively. Also, assume an instruction line requiring that in the next state they change to $a \leftarrow (b \bmod a)$ and $b \leftarrow a$.[39] Then, $\Delta(X) = \{(a, 3), (b, 6)\}$. However, if we have a state X' where $a = 3 = b$, then $\Delta(Y) = \{(a, 0)\}$. Here only the a element is included in the updates, because b does not change in this step and so $(b, 3)$ is not an update. More formally:

$$\Delta(X) = \{(l, b) : b \text{ becomes different from } X \text{ to } \tau_{\mathcal{A}}(X)\}$$

Any change $\Delta(X)$ from a state X to $\tau_{\mathcal{A}}(X)$ depends only on the explorations of the active zone. The third axiom guarantees that there is a finite subset of $\mathcal{A}$-terms (i.e., reached by a bounded exploration) that determine any updates $\Delta(X)$ in a state X.

Finally, *an algorithm is any entity that satisfies the above three axioms*. We'll call that 'Gurevich's thesis'.

Gurevich's conceptualisation

Gurevich's axiomatic approach was inspired by Gandy (1980). His work is meant to encompass all sequential algorithms, symbolic and non-symbolic ones. However, the states, the list of instructions, and the state transition mappings, are all expressed over a certain A-vocabulary.

Nevertheless, there is an important difference from Turing's approach. Whereas Turing explicitly required, in his definition of a 'computable sequence' in (1936), that the machine start with a blank tape, in order to avoid trivialising the concept of 'computable',[40] Gurevich's notion of 'state' (including the initial ones) is so broad, that it includes even non Turing-computable states. Thus, even the satisfactoriness (or the halting) problem could be encoded in an initial state.[41] The reason for such a choice is that Gurevich aims to capture *all* sequential algorithms, including any non Turing-implementable ones (e.g., ruler-and-compass, Gaussian Elimination, bisection algorithms, etc.; see, Gurevich 2015, 2014). As a result, the Church-Turing thesis is *not* immediately entailed by the above axioms; an extra, forth, axiom is required, to the effect that only undeniably-computable operations are available as initial states (Dershowitz and Gurevich, 2008).

[39] Recall that in computer science the '$b \bmod a$' operation refers to finding the remainder of dividing b by a.

[40] If the machine could start with any sequence written on the tape, it could start with an uncomputable number already there and then 'compute' this very number by running a program which just re-writes every symbol scanned on the tape.

[41] For example, the initial state could be a structure whose base set is $\mathbb{N}$ and which includes, among its relations and operations, a relation $S(n)$ which is true iff n is a *satisfactory* number; that is, a description number of a *circle-free* Turing machine (see, 2.4.3).

So Gurevich, like Moschovakis, does accept a broader notion of 'algorithm' —maybe the only similarity between the two approaches, though.

Gurevich's central motto —and most important for our purposes here— is: *every sequential algorithm has its native level of abstraction* (Gurevich, 2015, 203). And his basic goal is to express an abstracted version of Kolmogorov's locality constraint that will be *valid on any such level of abstraction*. Now, we saw earlier (2.4.3) that the K&U analysis was already a step further towards the mathematicians' meaning of 'algorithm', by admitting a broader notion of 'computational step' —though still at the lowest level of abstraction. Gurevich extends this to every level of abstraction.

Consider, for example, Euclid's algorithm. There are actually two different versions of this algorithm, one specifying subtractions at each step and another one divisions (the latter being more efficient). Arguably, these two versions exhibit different levels of abstraction. However, if they were to be implemented by Turing machines, which operate on, say, tallies or binary digits, then they would probably end up as the same program. This is so because at this level we would probably take as primary the operation of incrementing by one (e.g., by adding a tally at the end), along with a couple others (e.g., erase or copy a digit) and then program subtractions —or divisions— by means of nested iterations of the primary ones. Now, if we change computation model (read: programming language) this may mean a different level of abstraction. Each such level is determined every time by what operations are to be executed in one step, as well as by the abstraction levels of the algorithm's states (Gurevich, 2015, 204). In a nutshell, Gurevich accepts a very broad notion of state, and requires that any possible structure isomorphic to a state will also be a state (the set of states is closed under isomorphisms). This fulfils the goal of axiomatising different abstraction levels. But every unit step operation must still be of bounded complexity, and this is captured by the requirements that state transitions $\tau(X)$ respect isomorphisms, that the exploration during a step be always within a finite "active" zone, and that state updates $\Delta(X)$ depend only on the results of this exploration.

2.5 What the historical and theoretical investigations tell us

What the above considerations suggest is: limitations about what operations can be executed in one step, and what counts as bounded complexity of it, are not always absolute but may also be *relevant to some model of computation*.

This thesis, consistent —as I hope that I have showed— with the history and practice of mathematics, is also consistent with the few important —though rather dissimilar— foundational analyses of algorithms we examined.

Gurevich's formalisation, for example, requires that steps be results of exploration within an active zone (similarly to K&U), and that they do not alter the structure of the algorithm's states (dissimilarly to K&U). But the states themselves are quite broadly understood.

For example, a state could be a structure with only one binary relation; in this case it would be a graph, and the algorithm would be one operating on graphs. And, if, in some machine implementation, the nodes are binary numbers then any information details about these numbers would be irrelevant implementation details, and the algorithm itself would know nothing, at its native level of abstraction, about it.[42]

But, a state could also be a structure whose base set is $\mathbb{R}$, equipped with the operations and relations of a field. Then the algorithm would be a *real number algorithm*. Consider, for example, Euclid's algorithm, but now in its generalised version, as expressed by Euclid in his 10th book; that is, for comparing ratios of magnitudes, and deciding on their (in)commensurability. The goal is to identify a real number c, such that the two magnitudes, a and b, are integer multiples of it: $a = kc$ and $b = nc$, where $k, n \in \mathbb{Z}$. So assume the algorithm is in state X, where the parameters a, b have as values the lengths of the side (e.g., 6) and the diagonal of a square, respectively. The next instruction would be that the parameters change as $a \leftarrow (b \bmod a)$ and $b \leftarrow a$. Then $\Delta(X) = \{(a, \sqrt{2}), (b, 6)\}$.[43] This would be an algorithm in a high level of abstraction, since it assumes a computational step which, if implemented, would require either an infinite amount of time to be completed or to be approximated by floating-point systems. But, again, this would be implementation details, irrelevant to the algorithm itself.[44]

Although Gurevich's analysis is not directly concerned with addressing the thorny problems of formulating a theory of real computation, there is nothing in his analysis precluding algorithms like the above, as long as all three axioms are still satisfied. And, provided that any relevant terms can be expressed in the (finite) vocabulary of the algorithm,[45] the vocabulary function $F(a_1, ..., a_j)$ will still be the content of a *location* l. Thus, I believe that Gurevich's approach subsumes the kinds of things that Blum et al. mean as computation algorithms; that is, routines specifying operations between exact values of real numbers in one step.

But Moschovakis wouldn't disagree with our conclusion either. He would agree that what processes qualify as *algorithmic* ones is dependent on what operations are taken as immediate; but that is not a property holding *tout court*:

[42] See, e.g., Gurevich (2015, 202)

[43] In the generalised version of Euclid's algorithm, the quotients of $b \bmod a$, at each step, are still integers, whereas the remainders are now real numbers.

[44] As I take Gurevich's conceptualisation to imply.

[45] We could, for example, assume a language $\mathcal{L}_{\mathbb{R}}$ with a constant symbol $\dot{r}$ for each real number; e.g., $\dot{0}, \dot{\pi}, \dot{\frac{\sqrt{3}}{3}}$, and so on. Such languages were often considered within model-theoretic contexts. I thank Wayne Myrvold for this observation.

> It is tempting to assume that the successor operation $S(n) = n + 1$ on the natural numbers is "immediately computable," an absolute "given," presumably because of the trivial nature of the algorithm for constructing the unary (tally) representation of $S(n)$ from that of n —just *add one tally*; if we use binary notation, however, then the computation of $S(n)$ is not so trivial, and may require the examination of all $\log_2(n)$ binary digits of n for its construction—while multiplication by 2 becomes trivial now–*just add one* 0. The point [.. is] that while there is one, absolute notion of *computability* on $\mathbb{N}$ (by the Church-Turing Thesis), there is no corresponding absolute notion of "algorithm" on the natural numbers—much less on arbitrary sets. Algorithms make sense only *relative* to operations which we wish to admit as *immediately given* on the relevant sets of *data*. Any set can be a data set.. (Moschovakis, 1998, sec.8; emphasis in original).

But if "any set can be a data set", then $\mathbb{R}$, as an ordered field, can be a data set too, and the "operations which we wish to admit as *immediately given*" on $\mathbb{R}$ can well be $(\pm, \times, \div, \leqslant,)$, as in BSS/Real-RAM models. Therefore, although Moschovakis is not directly concerned with models of real computability either, his foundational approach also seems not to be in direct conflict with BSS.

To summarise so far. My goal has been to examine whether the term 'algorithm' exists in an informal-but-completely-precise manner in mathematics. I think that investigation into history, practice, and foundational analyses is consistent with a negative answer. We don't have a sharp proto-theoretical idea of 'algorithmic computation'; hence, the existence of the two traditions of computation, reflected on the two different approaches to real computability. Algorithmic computation over *concrete entities* of any kind (mathematical symbols, abacus pebbles, cogwheels, electrical impulses, etc.) by an idealised agent (human or otherwise) is naturally formalised by models in logic and computer science: Turing machines (Type I or II), KU-machines, recursive equations, etc. Algorithmic computation in the mathematicians' sense over *abstract mathematical entities* (real numbers, matrix rows, points and straight lines of a plane, etc.) is captured by Real-RAM/BSS approaches. Gurevich's axiomatisation can encompass both.

2.6 Which of the three interpretations?

There are still a few points that need to be addressed. First, as we again mentioned when we motivated the problem (p.19), all mathematicians at the informal level would agree about what features any algorithm should have; something along similar lines with the list we gave in sec.2.2.3. And this conviction about their general agreement is also shared by themselves. For example: "The intuitive concept of an algorithm, although it is nonrigorous, is clear to the extent that in practice there are no serious cases when mathematicians disagree in their opinion about whether some concretely given process is an algorithm or not", (Malc'ev, 1970, 18-19).[46] Hence, the need for actually formalising the concept arose only with the need for proving the non existence of certain algorithms. But in the end, we *do* have incompatible formalisations, in the domain of the reals. Where did things go wrong? Can we attempt a philosophical account of the problem?

Second, if the existence of this phenomenon is granted, which of the three possibilities of sect.2.2.2 would be a better account of it? Does the last paragraph of the previous section actually suggest that the second interpretation (B) is correct? That is, we use the concept of algorithm in more than one way? Is perhaps the first interpretation (A) —the informal concept is as precise as it gets and every formalisation should pick out the same class of routines and functions as its extension— still viable? Or, does 'algorithm' in fact have a more open character than expected —something like an "open texture", a property more commonly pertaining to empirical concepts?

I argue that the third interpretation is preferable, and that my account addresses both questions.

Recall the informal characterisation of 'algorithm', from sec.2.2.3. We have already pointed out that despite some *vagueness* in the *sense* of the proto-theoretic concept, we always pick out the same extension of the term under any reasonable sharpening, as long as we remain in the discrete case.

Where is this vagueness exactly located? Positing a computing agent in feature 5 (sec.2.2.3) is surely something not precise:

> The list of instructions that make up the algorithm are to be followed by a computing agent (human or otherwise) which carries out the computation.

But, arguably, the vaguer feature is the one that qualifies the algorithm as a *mechanical* process; that is, what we framed as follows:

> Each step of an algorithm must be specified to the smallest detail, precisely and unambiguously, such that no acumen or ingenuity or any semantic interpretation is required by the computing agent. Steps should be of a bounded complexity.

Phrases like 'smallest detail', 'acumen', 'bounded complexity' are vague. Nevertheless, this vagueness is "harmless", when we are dealing with a domain of entities as "coarse" as that of

[46]Statements of a similar flavour are plenty in mathematical texts, of course. This one here is just a randomly picked instance. Recall also the quote by Gödel (p.21).

the (non-negative) integers. Certain choices with respect to how to formalise these intuitions do not pick out different classes of objects; however, they do when in richer domains, such as $\mathbb{R}$. So we need to examine some such formalisation choices more closely.

First, we said that, as part of a proto-theoretic tidying, Turing narrowed his analysis to symbolic computations. Is this a restriction? Yes, if we are considering the general class of *all* computations. There *are* computations which are not symbolic, since ancient mathematics (ruler and compass, analog, etc.) —see the discussions on p.35 and in ch.4. Nevertheless, it is virtually not a restriction in the class of computations over integers. Not because computations over $\mathbb{N}$ are exclusively symbolic, but because this distinction does not make a difference in this domain. Consider the algorithm for Euclidean division, for example. We could think of it either as concrete, operating on tokens/symbols/names of the numbers divided, or abstract, operating on types/numbers/states-as-first-order-structures. In either case, there is no resulting difference about computability, despite the conceptual difference.

Borrowing Moschovakis's terminology (p.2.4.4), we could say that any algorithms (in the mathematicians' broader sense) in $\mathbb{N}$ cannot be distinguished from their implementations. Therefore, machine-like models (Turing, Post, Markov, register machines, etc.) which involve concrete operations concerned with pushing symbols around are co-extensive with more abstract "mathematical" ones, such as (μ-)recursive functions, which do not have any vocabulary-specific assumptions built into them.

But we can also draw an analogy with what Mostowski (1979) says about computability and computation, as semantic and syntactic ('linguistic') objects respectively:

> However much we would like to "mathematize" the definition of computability, we can never get completely rid of the semantic aspect of this concept. The process of computation is a linguistic notion (presupposing that our notion of a language is sufficiently general); what we have to do is to delimit a class of those functions (considered as abstract mathematical objects) for which there exists a corresponding linguistic object (a process of computation).

In these terms, we can say that in so far as we are in the denumerable case, Mostowski's suggestion *is* doable: We find no discrepancy between our semantic (intuitive) concept and its syntactic counterpart. But we are in a predicament in the uncountable case. Delimiting the class of "those functions .. for which there exists a corresponding linguistic [i.e., syntactic] object" leads to the class of TTE functions. But then semantics here splits with its syntactic counterpart, given all those arguments by some authors to the effect that such 'simple' functions as the step one *should* be computable in a good model.

A main reason for this phenomenon, of course, is that in an enumerable domain, such as ℕ, we can have a token/name/representation for every entity in the domain. Thus, there is no tangible difference if one understands algorithms as routines operating on concrete or abstract entities. And I submit that there is nothing in the proto-theoretical notion of 'algorithm' to suggest a choice of one over the other. The informal characterisation of sec.2.2.3 just does not have enough shape to definitively weigh against one or the other. And this is so, partly because no such need ever occurred in the use of the concept so far. So either way of *sharpening* the notion with respect to this particular question would be perfectly consistent with the implicit informal rules mathematicians have mastered for applying the concept to everyday mathematics. This is exactly what *open texture* means though:

> We introduce a concept and limit it in some directions; [..] This suffices for our present needs, and we do not probe any farther. We tend to overlook the fact that there are always other directions in which the concept has not been defined. [..] In short, it is not possible to define a concept .. with absolute precision, i.e. in such a way that every nook and cranny is blocked against entry of doubt. That is what is meant by the open texture of a concept. (Waismann, 1945, 123)

The second "open" issue —namely, the vagueness with respect to 'smallest detail', 'acumen', 'bounded complexity', or what have you— also ceases to be "innocent" when moving to conceptually richer domains. Before, as long as we remain faithful to the constraint that a computational step cannot be of infinite work (see, p.35), we don't get led astray from identifying the correct class of computable functions, within ℕ. This is why even explications of 'bounded complexity' that in fact may permit huge changes —such as Kolmogorov's model— are still co-extensive even with Turing's model which considers the most elementary operations. But, once again, moving to uncountable domains opens up Pandora's conceptual box. Are we to understand a step as a minimal operation on symbols, as in Type-2 Turing machines? Or, are operations such as "multiply such-and-such numbers", "draw the line between so-and-so points", "check whether $\sqrt{b^2 - 4ac} \geqslant 0$", "check whether point A is above point B" elementary basic steps —hence requiring no acumen— *within their relevant model*? In other words, do we start from what is taken as primitive in our model and regard as computable whatever is produced from that, by mechanical iterations, compositions, etc. or do we start from some *absolute* ideal of primitive operation and ground computability in that?

Interestingly, both approaches end up equivalent in the denumerable domain, and so the question dissolves. But, as a result, again the informal characterisation does not have enough shape to rule out either option when we move to the richer domain of ℝ (or ℂ). The open texture of the proto-theoretic notion becomes apparent in uncountable domains, where algorithms (in the mathematicians' broader sense) *do* differ form their implementations (as opposed to the $\mathbb{N}$ case). The uncountable infinity is what offers the unforeseen case, lurking in the open texture of our intuitive concept.

2.7 Conclusions

Let us iron out some details, summarise main points, and specify some open problems.

- We have said (sec.2.3) that there are methods —understood as *algorithms* throughout the history of mathematics— that are *not* Turing computable. This should not be taken as a point against the Church-Turing thesis, though. Rather, the conclusion should be that 'algorithm' is *not* synonymous with 'effective calculation'. There have been, of course, authors who identified the two notions; Church, for one, in his original article (1936, 356) about λ-definability. On the other hand, Turing never uses the term 'algorithm', at least in his (1936). Interestingly, Blum et al. (1997) do not use the term 'effective' to refer to computations in their model either.

  What exactly is the difference between the two terms? I would suggest the qualification that whatever procedure is effective is also algorithmic, but not vice-versa. And, adopting Knuth's (1997) characterisation of 'effectivity', I would also suggest that we distinguish the two on a basis of pen-and-paper simulation. That is, an algorithmic procedure is effective iff it can in principle be carried out exactly and in a finite length of time by an agent using only pen and paper.

  In that sense of the term, it's naturally expected that all algorithms over the integers are effective, since all integers can be symbolically represented on paper in a finite manner. Hence, we don't get led astray —in the discrete case— when we interpret CTT as being about *algorithms*. But Euclid's original version of his algorithm, for example, being about arbitrary lengths, cannot be simulated exactly on paper; even more so, if the two lengths are incommensurable. And an algorithm like the one for finding the roots of a quadratic equation may or may not be exactly simulable on paper either. So, according to the interpretation proposed here, although a particular implementation may be effective, the intended algorithm itself might not (note also that effectivity in this sense implies the finiteness requirement, which, as we saw, is not necessary for algorithms).

- Of course, someone could argue for just identifying algorithms with effective processes in all cases, on the basis that algorithms which can not be implemented are just abuse of terminology and should be called otherwise ('methods', 'rules', 'schemes', etc.). However, in my opinion, that would be to throw the baby out with the bathwater. We saw that there are a good many instances in mathematical practice that would be ruled out as algorithms, were a proto-theoretical tidying of that sort adopted. And recent foundational approaches try to be so general as to encompass, instead of exclude, general methods like that.

- Another way to put these problems is: must algorithms always be tied to representations? A positive answer would rule out any BSS/Real-RAM models from capturing facts about algorithms. Such models are not bound with any way of representing the entities on which the operations are applied. We again face the same dilemma; whether algorithms are meant as operating on abstract entities (types), or on concrete ones (tokens). Most cases from practice do not have enough shape to definitively suggest for one or the other; this is the phenomenon of open texture. Surely, we have methods, clearly concrete: abacus algorithms, for example. But we do have methods that clearly fall under the abstract side too: geometric constructions. The latter are purely conceptual and not about representations, for they operate on lines, circles, etc. as abstract entities. So again, are we to accept them as algorithms, within certain computation models, describing precise step-by-step calculations, closed under compositions and iterations of few primitives (e.g., compass only, scale and ruler, etc.)? Or we should reject them, since they are arguably not effective, in the sense defined above? Gurevich's axiomatisation seems able to accommodate geometric constructions, if we think of Tarski's (1959) first-order axiomatisation of Euclidean geometry. But, again, the intuitive notion has too much open texture to help us determine. It is by finally choosing in either way, that the community will end up sharpening the proto-theoretic notion further, and get rid of the openness.

- Contrary to 'algorithm', the notion of 'effectivity', as defined above, is quite clear without any openness. Thus, restricting 'algorithms' to effective processes would be a process of sharpening. But set-theoretic definitions are mathematically rigorous and precise too. So going to that direction would be sharpening as well.

  This is also the main reason why I put forward the third interpretation, in terms of 'open texture', instead of the second, in terms of 'poly-vague' concepts (sec. 2.2.2). Apparently, it may seem that we use the term in more than one way, different but precisely bounded. But the case I'm trying to make is that when all this foundational dust settles down, we *will* end up with a unique and sharp concept.

- The whole process of developing formal foundations for algorithmic computation can be seen as following an analogous course to the one described in Lakatos's (1976) about the notion of 'polyhedron' (although Lakatos's account doesn't involve any talk of 'open texture'). The final set-theoretic definition of 'polyhedron' is precise, but much of its intuitive content is lost. Similarly, both courses of action in the case of 'algorithm' would leave out intuitive content as well. Recall, for example, how Gurevich's formulation of states can admit encoding of non-computable situations (sec. 2.4.4). Moschovakis's set-theoretic formulation, on the other hand, also loses part of the intuitive meaning of 'algorithm' (see, Gurevich 2012 for a discussion on this). Contrary to the story told by Lakatos, in our particular case, the community has not settled on how to go about it yet.

Chapter 3

Choosing Foundations for Scientific Computing

Let us grant to those who work in any special field of investigation the freedom to use any form of expression which seems useful to them; the work in the field will sooner or later lead to the elimination of those forms which have no useful function. Let us be cautious in making assertions and critical in examining them, but tolerant in permitting linguistic forms.

R. Carnap, *Empiricism, Semantics, and Ontology*

A main point of the previous chapter was that the concept of 'algorithm' is not *used* in a unique precise manner in mathematics. It rather seems to be understood in two senses. According to the first, there is an absolute concept of 'algorithm'. Informally, the idea of mechanical clerk-like calculation exists in mathematics since its beginnings, and whether a specific process is an algorithm or not holds *simpliciter*. According to the second understanding, an algorithm is any stepwise process which is characterized as such within a certain *model of computation*.

The first sense is the dominant one in the history of mathematics. A few offered explications of it tried to pin down the constitutive elements of 'computation' in a fundamental way. Foundational analyses, like those of Turing's, Post's, Gandy's, Kolmogorov's, et.al's, were concerned with this, absolute, notion of computing. Restricting attention to symbol manipulation and examining steps which seemed undoubtedly elementary (e.g., writing one symbol on a tape-square) is an endeavour to pin down the computable in a non-relative sense. The second sense of algorithm became significant rather as a consequence of the rapid development of computer science and computational complexity theory in the last century, and of attempts to analyse algorithms in terms of their (exact) cost. What makes an elementary algorithmic step here is not immediately (or “objectively”) given but *stipulated* —built into the characterization of the model itself.

The first understanding of algorithms is the implicit one in the Church-Turing thesis, in talks about ‘effectivity’, and in traditional philosophical literature about the nature of computation. I believe it is the appropriate framework for all open problems about what is (un)computable, in some general *absolute* sense; for example, about topics of computability in physics,[1] hyper-computation, the so-called Physical Church-Turing thesis, thermodynamics of computation, etc. Within this framework, ‘algorithm’ can be identified with ‘effective’ procedure, in so far as we add the extra proviso that the scope is limited to terminating procedures. Accordingly, the Turing machine model and its natural extension by Type-2 Effectivity are the appropriate models for formal investigation of such topics, since they are explicitly concerned with ‘effectivity’.

Algorithm in the second sense of a method *relative* to a model of computation seems fairly recent —expressed in such terms, at least— due to the need for comparison between different methods that often exist for solving the same problem. The need to compare such methods with respect to their efficiency and accuracy, such as time cost and output error, became apparent especially after WWII when the advent of general purpose digital computers was around the corner and so some standards were desirable for comparing methods that would be implemented with inevitable rounding-off errors. Interestingly, a ground-laying paper in this area was again by Turing.[2] Turing (1948) describes methods for solving sets of linear equations and for inverting matrices, but his main concern is with “the theoretical limits of accuracy that may be obtained in the application of these methods, due to rounding-off errors” (1948, 287).

Nevertheless, if one is willing to accept also geometrical constructions as algorithms, then such constructions can be seen as algorithms in the *relative* sense. Such acceptance would then imply that this sense has also existed in parallel all along. More precisely (and maybe somewhat anachronistically), the classical geometrical constructions can be seen as *algorithms within a specific model of computation*; one with stipulated primitives, such as creation of the line through two existing points, creation of a circle with a given point as its centre passing through another given point, and creation of the point(s) in the intersection(s) of two non-parallel lines, of two circles, and of a line and a circle. ‘Computable’ then is whatever belongs in the closure of those primitives, under finite composition. ‘Complexity’ considerations can also be found then. For example, Émile Lemoine (1902), in his *Géométrographie*, developed a scheme to compare various constructions for solving a specific problem. He regimented the Euclidean primitives as consisting of the following operations (Preparata and Shamos, 1985): (a) place one leg of the

[1]Cf., for example, the recently published result about the undecidability of the spectral gap (Cubitt et al., 2015).
[2]The standard term now used for this kind of investigation is ‘algorithmic analysis’, coined by Knuth (1997).

compass on a given point, (b) place it on a given line, (c) create a circle, when one leg of the compass is in one of the previous cases, (d) pass the edge of a ruler through a given point, and (e) create a line with the ruler. Now, Lemoine was able to compare different constructions based on the number of operations needed. For example, he was able to reduce the first solutions to the Appollonius circles problem[3] from over 400 steps to 154 (Lemoine called such numbers the 'simplicity' of the method). Although Lemoine was interested in just improving the constructions and not developing a general "theory of simplicity", his work can still be seen as a form of measuring algorithmic costs and a predecessor of the field of *algorithmic analysis*. Similarly, whenever geometers were investigating what constructions are possible by only using, say, a compass (Georg Mohr), or a ruler and a transferer of segments (Hilbert), or a *neusis ruler* (that is, a markable ruler that was rotatable around a point; Archimedes, Nicomedes, Apollonius), they could be seen as actually working with different models of computation.

Now a main point of chapter 2 is that the two different senses of 'algorithm' —the absolute and the relative— become different only when considering calculation methods in uncountable domains; that is, numerical methods —e.g., Gauss elimination with real numbers as matrix elements— or geometric constructions in Euclidean spaces. On the other hand, in the domain of the integers, the class of functions that are algorithmically computable in the absolute sense is the same as those computable with respect to any reasonable model of computation. For example, if we consider the closure of all functions produced by the initial (successor, zero, and projection) functions under composition, recursion, and regular minimization (i.e., relative to an intuitively reasonable model of computation), then we obtain the same class of algorithmically computable functions as when we consider what is computable by absolutely elementary actions, such as reading/writing/erasing a symbol in a tape square.

3.1 Scientific computing

As mentioned above, it seems that the framework in which 'algorithm' is understood in some absolute sense is the appropriate one for addressing questions of effective (un)computability of problems. The other framework is useful for comparing different computational methods available for solving particular problems.

Now, we said in a previous section (1.3.3) that besides formalising the notion of 'algorithm', the two approaches to real computability, Effective Approximation[4] and BSS/Real-RAM/Real-

[3]The problem was to construct circles that are tangential to three other given circles, on the same plane.

[4]In this chapter I will mostly be using the more common term in the relevant literature of 'bit-computation', instead of 'Effective Approximation'. By 'bit-computation' I will be referring to those formalisations of EA that are more useful for discussing machine computations; that is, those that are based on a TM with an oracle.

number models, also aim to provide a foundation of scientific computing (see Fig.1.3). But they give inconsistent results. So how do we go about choosing the correct approach? Obviously, the way I distinguished between the two senses of 'algorithm' above mirrors the way 'algorithm' is understood, and formalised, within the two approaches. So, in a sense, the question is also about which understanding of algorithmic computation is the 'right' one, for founding scientific computing.

My approach to this question will be a pragmatic one. I'll be arguing that the choice of the appropriate model is dependent on the nature of the problem we're interested in. To do so, I'll discuss some of the flaws and merits of each approach, and how satisfactory they are with respect to certain desirable features. But, in addition to this pragmatic attitude towards the different approaches, I also suggest that the ultimate judge of what is in principle (un)computable by an idealised agent (machine or otherwise) must be the Effective Approximation models.

3.2 Intuitions

Generally, models of computation are abstract entities, specifying a structure that provides the descriptions of the objects of the modelled class of algorithms. Along with the structure, a model defines a *configuration*, that is, a complete description of the current situation (state) of the machine. A computational *step* is a "movement" from one configuration to another, determined by some instruction based on (one or more) elementary operation of the machine/program. A *computation* is a sequence of steps.[5]

Whenever we are to choose a model of computation for modelling some particular domain/class of algorithms, we have certain pre-theoretic intuitions that we want our chosen model to satisfy. What are those intuitions in our case?

A first attempt might be that a theory of computability about a mathematical area should try to characterise as 'computable' at least the fundamental operations used in the area (Gherardi, 2008). The four basic arithmetical operations $(\pm, \times, \div)$ are generally[6] computable in both BBS and EA approaches. However, n^{th}-roots are not BSS-computable, whereas they are unproblematic for EA. On the other hand, although comparisons $(>, <, \geq, \leq)$ and equality are not computable in EA, they are defined as computable in BSS, something that seems desirable for any theory which regiments numerical algorithms. This is so because comparisons between real numbers, as elements of an ordered set, are taken as basic and are built into many such algorithms (consider, as an example, a classic bisection, root-finding method).

[5]This characterisation is a slightly modified version of the one in Hromkovič (2003).

[6]The use of 'generally' here is because we are interested in a broader conceptual comparison between the two approaches that would leave aside complications of the kind we saw in fn.23 of ch.2.

A further factor is what functions each model regards as computable. We have already pointed out that only continuous functions are EA-computable. This is an inevitable consequence, necessary (actually *constitutive*) of *any* framework in which computation is understood as an effective process over uncountable domains—that is, it can in principle be simulated by pen and paper. Of course the uncomputability of the comparison relation comes from the same conceptual reasons. This is an undesirable feature for the purposes of analysing numerical algorithms, full with comparisons, and so it is evaded in BSS by fiat. Furthermore, functions which cannot be defined solely by the four arithmetical operations and/or comparisons —such as roots, exponential, logarithmic, trigonometric, etc.— are not BSS/Real-RAM computable. Whenever a particular problem involves such functions, as parts of the investigated algorithms, their computability is stipulated and their computation comes at some fixed cost. Again, this is for sure problematic, if someone is interested in what is effectively (un)computable in the absolute sense. But it's convenient for analyses of algorithm. Another issue is that *any* constant function $f(x) = c$ ($c \in \mathbb{R}$) is computable in BSS. This is again conceptually suspicious, since all real numbers but denumerably many are non-computable.

Another proposed criterion for evaluating a theory of computability is how well it matches our intuitive notions of 'easy', 'hard', and 'impossible' computation (Braverman, 2008). The usual arithmetic operations should be characterised as 'easy' by any model. Arguably, the same goes for every function found in a common calculator, such as $\sin x$. Also, problems that are known to be easy, hard, or impossible, in the discrete case might be expected to fall under the same categories in the continuous case. For example, the Travelling Salesman Problem, which is NP-complete in the discrete case, or other known hard problems, such as the N-body problem or the Navier-Stokes equations, are expected to be considered hard in the continuous case too (Braverman and Cook, 2006).[7] The fact that for BSS $f(x) = c$ is always computable for *any* $c \in \mathbb{R}$, perhaps owing to the extreme algebraic simplicity of this function, goes against the "criterion" we are discussing here. It is possible to construct particular c's that would encode a solution to the Halting Problem, which we know to be uncomputable. Then such c's would be impossible to compute; however, categorised as 'easy' by BSS.

Finally, there is another aspect that is captured quite differently by the two approaches. The cost of a computation method is measured in computer science and mathematics as a function of the input size. In the classical computational complexity theory, information about costs is exact. However, in numerical analysis and information-based complexity problems, the information is more often than not incomplete (sec.2.1). Now, in Turing models, within EA, the input size is measured in bits (word size). Let's consider a well-conditioned problem. A slight perturbation of the value of the input, say, from 1 to $1 + (\frac{1}{2})^n$, would cause the input bit size to grow from 1

[7]Except for the unlikely case that P=NP.

to n+1 (Blum, 2004). For large *n*, this can spoil the analysis, because although the input size would change considerably, due to the small perturbation, its actual complexity class should not change if the problem is well-conditioned. Thus, bit computation can lead astray, if such circumstances are not taken into account, when evaluating algorithmic costs and, accordingly, problem complexity classes.

3.3 Why the BSS/Real-RAM models work

The point of the previous discussion is that neither of the two approaches satisfies *all* of our pre-theoretic expectations about computability over the reals. Nevertheless, both frameworks have been fruitful in terms of their results. Now, bit-computation is a natural extension of the Turing model and is grounded in plausible and intuitive assumptions, such as those of Turing machines *approximating* the exact values of the computable reals by writing/erasing symbols from a finite alphabet, etc. Thus, bit-computation regiments 'effective computation' successfully. But BSS/Real-RAM models are not concerned with explicating 'effectivity' and are grounded in assumptions and idealisations, that have been criticized as unnatural and unrealistic. See, for example, Weihrauch (2000), for a standard criticism. Despite that, the BSS/Real-RAM models are widely and helpfully applied to various areas of computational mathematics (see, also, ch.2 and fn.3 therein). Wozniakowski's comment is characteristic:[8]

> Before we go on, we pause for a moment and ask why so many people are using the real number model, and why there are so few complaints about this model. In fact, with a little exaggeration, one can say that only some theoreticians are unhappy with the real number model, whereas many practitioners do not experience any problem with it. (Woźniakowski, 1999, 452)

The BSS model purports to offer a foundation of scientific computing, by examining the notion of computation itself. "There is a sense in which this work is a study of the laws of computation", write Blum et al. (1997, 22). And they continue:

> [W]e write not from the point of view of the engineer who looks for a good algorithm [..] The perspective is more like that of a physicist, trying to understand the laws of scientific computation. Idealizations are appropriate, but such idealizations should carry basic truths. (Blum et al., 1997, 22)

And later on:

[8]Wozniakowski, coming from the area of information-based complexity, uses the term 'real number model'. But, as already said, BSS, Real-RAM, and Real Number are all, for the most part, equivalent models.

> [W]e are led to expanding the theoretical model of the machine to allow real numbers as inputs. There has been great hesitation to do this because of the digital nature of the computer. Here we might learn a lesson from the history of science. At the time of Newton, scientists assumed that the world was atomistic [..] Newton accepted that picture according to which all matter is composed of indivisible particles, a finite number in each bounded region. On the other hand, Newton's mathematics was continuous as was Euclid's. [..] It was a substantial problem for Newton to reconcile the discrete world with the continuous mathematics. The resolution was produced by analyzing the effect of replacing an object (e.g., the earth) by a finite number of particles, then making a better approximation with a larger number of particles. In the limit, the mathematics becomes continuous. [..]
>
> Now our suggestion is that the modern digital computer could be idealized in the same way that Newton idealized his discrete universe. The machine numbers are rational numbers, finite in number, but they fill up a bounded set of real numbers (e.g., between −1000 and 1000) sufficiently densely that viewing the computer as manipulating real numbers is a reasonable idealization, at least in a number of contexts. (Blum et al., 1997, 23-24)

The analogy with empirical science seems interesting. But how far can it go? We attempt an answer in the next section. Blum et al. say nothing further, nor do they try to give any more precise account of how and why such an idealised model would even give reliable results in the first place. Can we do better?

3.3.1 Idealisations in scientific representation and the principle of regularity

Much ink has been spilled on how and why idealised models can provide us with knowledge, in the philosophy of science literature. We need not focus on this issue here. But important insights into this matter can be gained by what has been called "the principle of regularity" by Tisza (1963). That is, by the requirement that mathematical solutions of equations modelling physical phenomena be insensitive to small perturbations in the initial data, in order to have any physical meaning:

> Experiment provides us with the values of continuous variables within a certain accuracy. However, the analysis of mathematical physics would be extremely cumbersome and lose much of its precision if the finite width of continuous parameters corresponding to empirical quantities were to be observed at every step. In actual

> practice, the segments of finite widths are sharpened into definite points of the continuum for the purposes of the analysis. However, the results obtained have no physical meaning unless they are sufficiently insensitive to the actual unsharpness of the continuous parameters.
>
> Mathematical solutions satisfying this requirement will be called *regular*. [..] The requirement that mathematical solutions be regular in order to have a physical meaning will prove to be of great importance and deserves to be called a principle, the *principle of regularity*. (1963, 159; emphasis in original)

As it is discussed in Myrvold (1994, 1995), the point of this principle is not to provide some a priori reasoning to the effect that no physical systems exhibit discontinuous dependence on initial conditions or that 'nature does not make jumps'. Rather than being an ontological dictum, the principle makes a methodological point about what predictions of a physical theory can be tested by measurement and what cannot. "Even if *natura facit saltum*, the quantities about which we can make reliable quantitative predictions will be found in the regions where Nature refrains from leaping" (1995, 39). Furthermore, theoretical results which violate this continuity condition would not be of any help for numerical predictions, since in such cases, approximations to the initial data might not necessarily lead us to approximations to the solution (1994, 78).

The above play an important role in understanding why approximations and/or idealisations in models may be reliable: in a successful mathematical representation the points of the continuum, as resulted by the "sharpening" of the error intervals into definite points —as Tisza puts it— are insensitive to the actual "unsharpness" of the continuous parameters; therefore, any theoretical definite point of the representation will be a good approximation for all actual values within a neighbourhood around it. Small variations within the neighbourhood do not render the theoretical point of the model too far removed from the actual one; it'll still be a good approximation. Furthermore, the extent to which we can safely idealise our representation is usually based on a compromise between accuracy and calculability.

3.3.2 BSS/Real-RAM as idealised representations

Interestingly, we can observe an analogous phenomenon in those cases for which BSS is a successful model. When BSS (or Real-RAM) is used to analyse the cost of specific algorithms (and, based on that, the complexity of certain problems), it can be seen as an idealised representation of the actual computation performed by a computer, which in fact employs floating-point arithmetic (see, also, fn.26 of ch.2).

Model	**Values**	**Precision**	**Operations and costs**
BSS	Real	Infinite, exact	Addition, subtraction, multiplication, division, comparison, at *unit* cost
FPA	Rational	Finite, limited	Addition, subtraction, multiplication, division, comparison, at *fixed* cost
Bit-comp.	Rational	Finite, arbitrary	Addition, subtraction, multiplication, division (no comparison), at cost *dependent on the size of the operands*

Table 3.1: Comparison between the two models and floating-point arithmetic

More specifically, the floating-point numbers can be seen as the "analogue" of experimental values. The latter are always measured with some error, which corresponds to the *representation error* of floating-point numbers, since in both cases a real value gets approximated by a rational one. Round-off errors can be corresponded to propagation of errors in experimental physics; that is, uncertainties in *derived* quantities caused by initial uncertainties in the experimentally measured quantities.

Now, BSS/Real-RAM analyse computations between values that are assumed (i) real (ii) exact (iii) able to be added, subtracted, multiplied, divided, and compared at *unit cost*. But, given that such models are actually successful, this might seem to fly in the face of the fact that actual computers use floating-point arithmetic (FPA); especially given some of this arithmetic's bizarre properties, namely that floating-point numbers make up a finite set which is not closed under addition, it is not a field, ring or group, and in which addition is generally not associative or distributive.[9,10]

On the other hand, floating-point computations are between values that are (i) rational (ii) rounded and of finite precision (iii) added, subtracted, multiplied, divided, and compared at some fixed cost.

For comparison, we also mention that bit-computations (TTE) are between values that are (i) rational (ii) of arbitrary precision (iii) added, subtracted, multiplied, and divided (but not compared) at a cost which is dependent on the size of the operands and the desired precision. We summarise these features in table 3.1.

In what follows, we will be examining all these three features separately, in order to under-

[9]Here is another counter-intuitive property of BSS/Real-RAM: those models assume that any number can be a built-in parameter; for example, π. In reality, however, the computation of the n first digits of π can become a difficult problem, if n is large enough; so a fortiori for the whole value of π (Woźniakowski, 1999).

[10]See, e.g., Corless and Fillion (2013).

stand under what circumstances BSS/Real-RAM are reliable.

The first difference between BSS and FPA can be theoretically "alleviated" as follows. Floating point numbers are most commonly defined as integers multiplied by positive or negative powers of 2. This means that all FP-numbers represented this way are dyadic rationals; that is, elements of $\mathbb{D} = \{\frac{a}{2^b} : a \in \mathbb{Z}, b \in \mathbb{N}\}$. The set $\mathbb{D}$ enjoys some properties that make it very convenient for computation models. It is closed under addition, subtraction and multiplication. It is also dense in $\mathbb{R}$; that is, for any real number r, any neighbourhood of r contains at least one dyadic rational in it. Furthermore, any real r can be approximated by a $d \in \mathbb{D}$ of the form $\lfloor 2^k x \rfloor / 2^k$ with arbitrary precision.[11] Finally, the dyadic rationals are precisely those numbers whose binary expansion is finite.[12] Interestingly, some operations between dyadics can be performed exactly, even in FPA, under certain circumstances (e.g., when there's no underflow). For example, dyadic rationals can be added or divided by two without round-off, a property that can make a bisection algorithm having much less roundoff errors (Woźniakowski, 1999). Of course, also some Effective Approximation models use dyadic rationals to represent reals (see, e.g., Braverman 2008), and so they exploit the merits of this system as well.

The second contrast between BSS and FPA is the one that makes the success of BSS in modelling FPA most remarkable. During a FPA computation there are roundoff and representation errors that may accumulate. But most researchers in scientific computing do not experience significant difference between the two models, for most (but not all) cases. Can we identify some reason, analogous to the principle of regularity in scientific representation, which accounts for the fact that BSS can reliably model FPA computation, despite that it abstracts away from roundoffs and idealises FP-numbers as reals? I believe that we can, and that the discussion in Woźniakowski (1999) can be helpful for the question at hand.[13]

Woźniakowski posits two assumptions that are necessary for BSS/Real-RAM to be a good approximation to FPA. The first is that we are modelling an algorithm which is *stable*. More often than not, we don't have access to accurate data, but only to approximations within some error. So, in most practical cases we don't compute the exact solution of the problem but a close one. Based on that, we say that an algorithm is stable, if it does not magnify the error in the

[11]The importance of this fact can be seen in the occurrence of the ρ parameter in equation (3.1), of p.60.

[12]There might seem to be a tension here, since we said that the set of FP-numbers is not closed under addition, whereas $\mathbb{D}$ is. But the claim is not that the two sets are identical; only that FP-numbers can be defined as dyadic rationals. One major difference between the sets, for example, is that FP-numbers make a finite set whereas $\mathbb{D}$ is denumerable.

[13]Woźniakowski's discussion is about the use of Real Number model in information-based complexity, but I think that the results bear on the context of our discussion as well. We heavily draw upon his discussion in what follows.

input data. Differently put, a numerically stable algorithm yields the correct output for a slightly perturbed problem. Now, if we are modelling a stable algorithm in BSS, then the cost analysis we are obtaining will be faithful to the actual cost of the same algorithm in FPA.

We note in passing here, that the cost of an *algorithm*, in general, is the added cost of all particular operation costs, expressed as a function of some input size n, in asymptotic form. If we take the worst case cost for any particular algorithm that solves a particular problem, and then the minimum algorithm cost (of known and unknown algorithms), then we can identify this cost with the complexity of the problem.[14]

Now in order to translate the obtained cost to a complexity result, we also need to know that the examined algorithms are algorithms solving a *problem* which is not too far from the original one. We already know that our algorithm computes exactly a slightly perturbed problem (by the definition of stability). But it is necessary to know that the result obtained in BSS is not far from the actual one, in FPA, in order to be able to transfer the results.

Let ε be the (absolute) forward error; that is, the difference between the actual and the computed solutions. Our stable algorithm solves exactly a slightly perturbed problem, that is, a problem with slightly perturbed initial data. We need in general to be able to measure how much the result of the output will be affected by the perturbations of the input. This sensitivity of the solution to small changes in the data is called 'the conditioning of the problem' and it's measured by the *condition number*, k, of the problem.[15] We also need to take into account the accumulation of roundoff errors due to the algorithm. This is expressed by the *accumulation constant* C *of roundoff errors* of the algorithm, which is often a polynomial in the number of inputs and is expect to be small since we assume a stable algorithm.[16] We also need to take into account the roundoff error ρ of FPA. This depends on the particular system of FPA implemented (single, double precision, size of mantissa, etc.).

We can now state the second assumption of Woźniakowski (1999), in order for BSS/Real-RAM to faithfully model FPA.[17] The assumption is that the accumulation error C of a stable algorithm which solves a problem of conditioning k, combined with the roundoff error of the floating-point numbers, do not altogether get larger than the forward error ε. Formally,

[14]Terminological confusion is in abundance here. Many practitioners use the term 'complexity' both for algorithms and problems. Others, distinguish between saying that an algorithm has a *time complexity*, whereas a problem belongs in a *complexity class*. Whatever the terminology one may adopt, one should always bear in mind that there is conceptual difference between the two cases. Here, we choose to talk about the '*cost*' of an algorithm, and the '*complexity*' of a *problem*.

[15]It is important to keep in mind the distinction that 'stability' is a property of an *algorithm*, whereas 'conditioning' is a property of the *problem*. If a problem is ill-conditioned, a stable algorithm is not enough to save the situation, the computation error will be large. But if a problem is well-conditioned, then all stable algorithms will give results with small errors.

[16]See, Woźniakowski (1999).

[17]The first was the stability of the algorithm.

(Woźniakowski, 1999, 455):

$$C \cdot \rho \cdot k \leq \varepsilon \tag{3.1}$$

So whenever (3.1) holds, we are guaranteed that the total errors of the actual computation will not go above a theoretical error ε (commonly within the interval $[10^{-8}, 10^{-2}]$, for most applications).[18] As long as this relation holds, we know that the solutions we obtained will be trustworthy, even though we used BSS/Real-RAM to model floating-point arithmetic. The parameter C has to do with the algorithm (which is assumed stable, so C is small), ρ has to do with the system of FPA that implements the actual computation, and k with the problem itself.

The suggestion here is that (3.1) might be seen as playing a role analogous —at least in so far as the analogy goes here— to the role of the principle of regularity. In scientific representation, trying to solve, in some idealised model, an ill-posed problem which violates the principle of regularity, will give unreliable quantitative predictions. In a similar vein, in scientific computing, trying to solve in an idealised model (BSS) an ill-conditioned problem which violates (3.1) will give results that are untrustworthy.

Now, to go back to the comparison between BSS/Real-RAM and FPA, the third important characteristic is the cost of the operations. In fact, in this respect BSS is closer to actual FPA computations than bit-computation models. This is so because in both cases the cost of operations is fixed; that is, it is not dependent on the *size* of the operands. So one only needs to replace the unit costs calculated in BSS/Real-RAM with the actual fixed costs of operation in FPA. This is a big difference with TM-based models, such as bit-computation, where the size of operands matters. Since costs of algorithms are expressed as functions of input sizes, in the case of a Turing machine with an oracle, the input includes the desired precision n of the output (see, sec.1.3.2).

3.3.3 When the idealisations fail

What happens when (3.1) is not satisfied? One way to deal with that is to increase the precision of FPA; that is, to reduce ρ. But if the problem is not well-conditioned, then (3.1) might not be satisfied still. In that case, a more "fine-structured" model is necessary. The bit-computation model (TTE) will be the appropriate framework then, since its results with respect to costs and complexity are more sensitive to the available accuracy of the approximation to the initial data. Of course, all this presupposes that the problems of conditioning discussed on p.54 have been taken into account.

[18] Woźniakowski (1999).

For more on how cost and complexity are analysed in bit-computation, see Braverman (2008) and Ko (1991).

3.4 Which foundations?

We return to the main question of this chapter. Given that both approaches have been formulated as foundations of scientific computing, how can we go about choosing between them? I think that this depends on what we mean by a 'foundation for scientific computing' exactly.

One way to interpret the question, I think, is as asking which model/approach is *the* correct one for analysing the algorithms that are implemented in scientific computing. I argue for a pragmatic answer to this question. BSS/Real-RAM is better for "higher level" analyses. As discussed in ch.2, whenever researchers in numerical analysis, computational geometry, etc. develop and analyse algorithms, they treat them as abstract entities, independent of any particular implementation. Often, they also assume that the algorithms operate over exact real numbers (cf., Gauss elimination, bisection algorithms, etc.) for ease of calculations. In other words, their implicit model of computation —which defines what steps are primitive and, hence, what counts as an algorithm— is, in this respect, something similar to BSS/Real-RAM. This is so because such models make the analyses easier. So as long as both the *stability* assumption and condition (3.1) hold, BSS/Real-RAM is preferable, due to mathematical tractability.[19] The "unrealistic" idealisations is the price to pay for manageable analyses. But if our particular problem demands analyses of a finer structure, either because particular applications require smaller ε, or because the problem at hand is too sensitive to perturbations in the initial data, then bit-computation is apt. Here, the difficulty of analyses is the price to pay for more "realistic" assumptions.

On the other hand, if we interpret the question as asking what is the correct approach to inform us about what can ultimately and in principle be computed, by a computer (machine or an idealised agent following an *effective* algorithm and given enough space and time), then I believe that bit-computation is the only appropriate framework to use. It is the only natural extension of Turing machines and *the* correct explication of effective computation; thus, it is the correct model for a generalised Church-Turing thesis over the reals. Due to this fact, of course, even with respect to the previous question about analysing algorithms, bit-computation can be used to analyse any problem as well, at least in principle (Woźniakowski, 1999). But this would be rather inefficient and unmotivated, for the actual practice.

A similar situation can be found in philosophy of science. There is much discussion about ide-

[19]Of course this may vary according to the specific applications we are interested in. The ε parameter is chosen by us, depending on what approximation errors we are willing to accept (see, Woźniakowski 1999).

alised models and "inconsistent" scientific representations, in cases where we employ different, or even incompatible, descriptions of certain systems. To give a much-cited example,[20] in fluid dynamics we often mathematically represent water as a continuous medium (by means of, e.g., the Euler or the Navier-Stokes equations). But we know that water is in fact discrete, consisting of molecules, and we do represent it as such within other contexts (e.g., in a chemistry or QM class). We need not deal with such issues for our purposes here.[21] It is enough to observe that the choice of the appropriate representation turns on pragmatic considerations, related to how much one is willing to make a trade-off between manageability of calculations and accuracy of representation, according to the needs of the specific application. But, no scientist would state that the continuous representation of water is actually true; if a scientist was asked "what water is really like?" the reply would rather be "discrete!".

I think a similar case can be made for the rival approaches to real computability. And, although this phenomenon has been discussed a lot in the context of empirical science, I think that it is an interesting fact that something similar may hold even in the precise and ideal world of mathematics.

3.5 Conclusions

To conclude, BSS/Real-RAM are suitable models for analysing algorithms and problem complexities under most circumstances. Questions about algorithmic computability by an (idealised) agent, in some absolute sense, can be addressed only by bit-computation, as well as analyses of ill-conditioned problems. Of course, *any* problem can be analysed by bit-computation, in principle, but this would be impractical and unmotivated.

Finally, the very existence of (the need for) idealisations within mathematical domains —if computability is to be understood as such— might seem an interesting phenomenon. But, we may perhaps reverse the reasoning and take it to indicate (or remind us of) the fact that implicit *empirical* assumptions may also exist, inherently, in our very concept of 'computation' itself.

[20] See, e.g., Maddy (1997, 1992) and Pincock (2012, 2014).

[21] Despite the many excellent arguments in this discussion, it seems that a case can be made that both sides missed the point of the principle of regularity as an explanatory factor for successful idealisations, as well as that there are some conceptual confusions between *approximations* and *idealisations*. For a discussion about the difference between the two concepts, see Norton (2012).

Chapter 4

Computing and Modelling: Analog vs. Analogue

In this chapter, analog computational models and analogue (physical) models are examined. Most people are familiar today with digital computing devices, such as smart-phones or personal computers. Nevertheless, a case can be made that digital electronic computing makes up a rather small portion of the history and practice of computing. Arguably, in order for one to fully understand ‘computation’ and be able to account for certain aspects of it (e.g., the difference between digital and analog) one has to look into what people have been doing for a long time and calling it ‘computation’.

To this end, we first examine some historical examples of mechanical computing devices and practice (sec.4.1). Then, we look into electronic analog computing and mathematical models of it (sec.4.2). We discuss the various accounts of the distinction between ‘analog’ and ‘digital’ computation in the literature, and put forward one, largely based on Goodman’s (1976) distinction between analog and digital representations (sec.4.3), by also arguing that representation is an essential part of computing. Finally, we briefly examine the use of physical models in science; we distinguish between two classes of physical models (demonstrational and scale models) and we compare their use with the use of analog computational models (sec.4.8). We close by suggesting directions for further related research (sec.4.10).

Throughout this work –except for the quotations–, the American spelling, ‘analog’, is used for computation, in the sense of opposite of ‘digital’, whereas the British spelling, ‘analogue’, is used for models, in the sense of ‘analogous’. The motivation for this choice will hopefully become clearer by the end of the chapter.[1]

[1]This use of the different spellings to refer to the different practices was proposed by S. Sterrett.

4.1 Historical examples of mechanical computing devices

We will briefly review some typical examples of machinery-aided computation, performed by scientists and engineers before the advent of electronic, stored-program, digital computers. Such examples are slide rules, nomograms, planimeters, integrators, and general purpose analog devices. The order of presentation is not necessarily chronological; rather, it is based on the amount of sophistication.

Slide rule: Slide rules are instruments for multiplying and dividing numbers.[2] They consist of sticks graduated with logarithms of numbers, able to slide relative to each other. Since the numbers to be operated upon are represented by lengths on the sticks and lengths are calibrated in logarithmic scales, adding lengths actually corresponds to number multiplication. The use of slide rules goes as far back as the discovery of logarithms.[3] Initially the English mathematician E. Gunter (1581–1626) drew a "logarithmic line" on his *Scales* (calibrated sticks) and, shortly after, E. Wingate (1596–1656) constructed the slide rule by reproducing the logarithmic scale on a "slide" which could be moved along the first scale. This way, the need for use of compasses, in order to add the logarithms, was avoided.[4]

Nomogram: Nomograms are two-dimensional graphical representations of mathematical relations. They have been used extensively for quick graphical computation of certain functions, within precision which is adequate for practical purposes. There are great many varieties of them, but the basic idea is pretty much the same and can be seen in the example of Fig.4.1. A typical parallel-scale nomogram is depicted, used for calculation of the two-variable function: $T = (1.84S + 4.66)^{0.37} \cdot (1.21R)^{\frac{4}{3}}$. The result is obtained by laying a straightedge (red dotted line in the picture) across the known values on the scales and reading the unknown value from where it crosses the scale for that variable.

Other common examples of uses of nomograms include computation of the total resistance of two parallel resistors ($R_t = \frac{R_1 R_2}{R_1 + R_2}$; see, Fig.4.2), missing terms of the law of sines, roots of quadratic or cubic equations, and so on. Applications of nomograms have been vast, ranging from statistics and economics to astronomy, geography, ballistics, engineering, etc. Some nomo-

[2]Other functions, such as reciprocals, logarithms, roots, etc. may be calculated as well, in some cases.
[3]*Encyclopædia Britannica* (1911), s.v. "Calculating Machines".
[4]*Encyclopædia Britannica* (1911), s.v. "Calculating Machines".

grams, like Smith's chart (Fig.4.3), are still in use today in areas, such as electrical engineering.

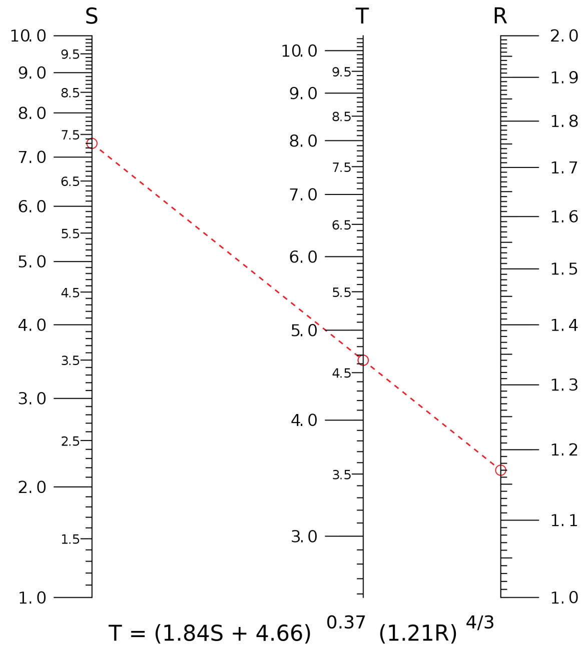


Figure 4.1: A nomogram for quick computation of the two-variable function $T(S,R)$. In the depicted case, a value ≈ 4.65 is obtained for input (7.3, 1.17). [Wikimedia Commons, By Rrrddd (CC BY-SA 3.0)].

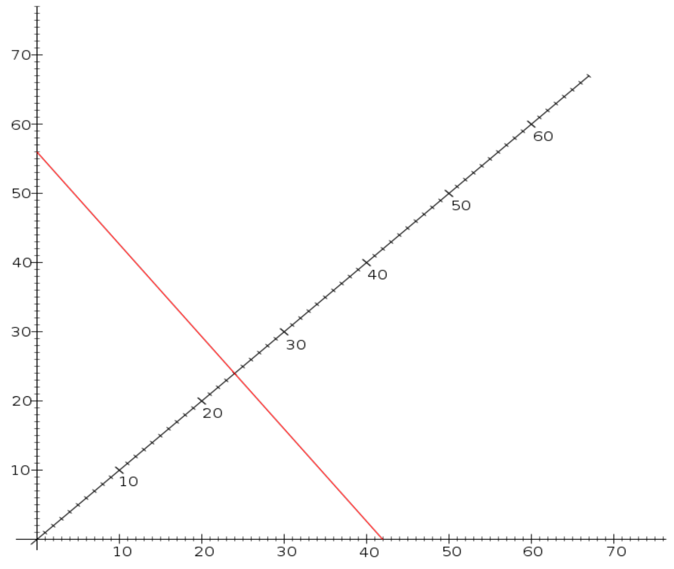


Figure 4.2: A nomogram for computing $\frac{1}{z} = \frac{1}{x} + \frac{1}{y}$. For inputs $x = 42$ and $y = 56$, the output, indicated by the straightedge, is $z = 42$.

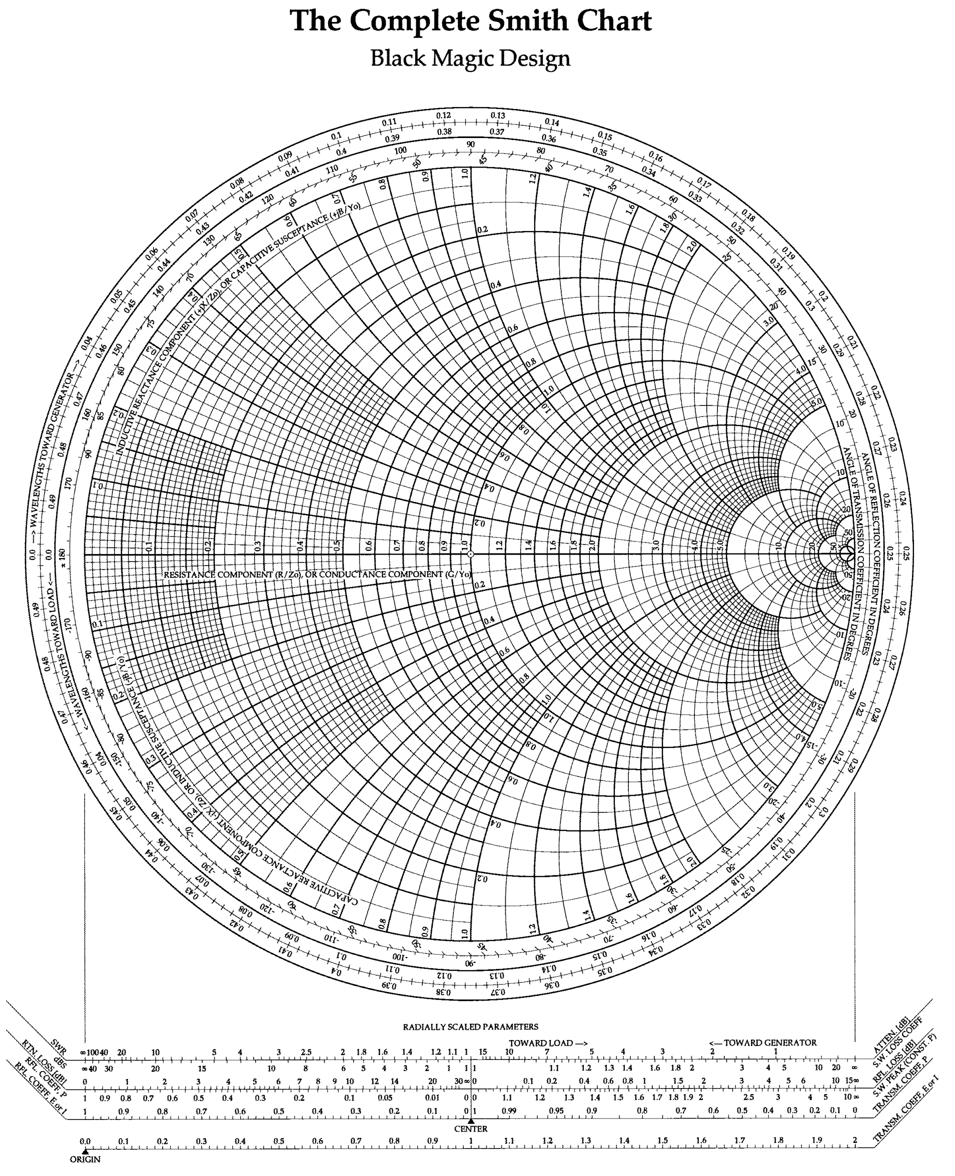
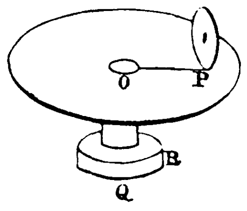

The Complete Smith Chart
Black Magic Design
RESISTANCE COMPONENT (R/Zo), OR CONDUCTANCE COMPONENT (G/Yo)
INDUCTIVE REACTANCE COMPONENT (+jX/Zo), OR CAPACITIVE SUSCEPTANCE (+jB/Yo)
CAPACITIVE REACTANCE COMPONENT (-jX/Zo), OR INDUCTIVE SUSCEPTANCE (-jB/Yo)
ANGLE OF REFLECTION COEFFICIENT IN DEGREES
ANGLE OF TRANSMISSION COEFFICIENT IN DEGREES
0.0 —> WAVELENGTHS TOWARD GENERATOR —>
<— WAVELENGTHS TOWARD LOAD <—
RADIALLY SCALED PARAMETERS
TOWARD LOAD —>
<— TOWARD GENERATOR
SWR
dBS
RTN. LOSS [dB]
RFL. COEFF, P
RFL. COEFF, E or I
ATTEN. [dB]
S.W. LOSS COEFF
RFL. LOSS [dB]
S.W. PEAK (CONST. P)
TRANSM. COEFF, P
TRANSM. COEFF, E or I
CENTER
ORIGIN

Figure 4.3: A Smith Chart, used in microwave engineering for computing complex quantities, such as admittance and impedance on any transmission line, on any load.

Planimeter and integrators: Planimeters are devices for computing the area of a closed curve. In the most common version, the user traces the boundary of the given surface with a pointer (called the 'tracer'), and reads off the area on the device's recording apparatus. In effect, it can be seen as calculating the integral $\int_a^b f(x)dx$, given $f(x)$ as input. Mathematical explanations of how exactly planimeters work are readily available on the internet, but Maxwell's own explanation, when he invented one, is still very lucid. We won't focus on the mathematical principles behind planimeter, but we will quote here a partial description of the mechanics. The described principle became the basis for many analog computing machines in the following years.

> We have next to consider the various methods of communicating the required motion to the index. The first is by means of two discs, the first having a flat horizontal rough surface, turning on a vertical axis, OQ, and the second vertical, with its circumference resting on the flat surface of the first at P, so as to be driven round by the motion of the first disc. The velocity of the second disc will depend on OP, the distance of the point of contact from the centre of the first disc; so that if OP be made always equal to the generating line, the conditions of the instrument will be fulfilled.
>
> This is accomplished by causing the index-disc to slip along the radius of the horizontal disc; so that in working the instrument, the motion of the index-disc[5] is compounded of a rolling motion due to the rotation of the first disc, and a slipping motion due to the variation of the generating line [OP]. (Maxwell, 2011, 231-232)



The planimeter belongs in a class of analog devices often called ‘integrators’, because, as we said, they can be seen as essentially calculating an integral of the input function. James Thomson (1822–1892) produced an improved version of a planimeter, based on Maxwell’s one, but he didn’t see any use for it at the time (Goldstine, 1973, 40). Nevertheless, his brother, Lord Kelvin, realised that he could make use of such integrators to build tide-calculating machines. Thus, Kelvin built ground-breaking machines for tide-related calculations, three of which were a tide gauge, as tidal harmonic analyser, and a tide predictor.[6] Kelvin’s tide predictor remained in use at the port of Liverpool until the 1960’s (Copeland, 2008).

A further remarkable achievement by Lord Kelvin was his theoretical ideas about computing the solutions to differential equations. Kelvin considered the most general second order linear differential equation, which he put into the form:

$$\frac{d}{dx}\left(\frac{1}{p(x)}\frac{du}{dx}\right) = u \tag{4.1}$$

[5]The ‘index-disc’ that Maxwell is referring to is the smaller, vertical disc, in the picture below.
[6]See, Goldstine (1973).

He then considered a device with two integrators such that given an initial guess u_1 for u (obtained by some iterative process) as input to the first integrator, it would calculate:

$$g(x) = \int_0^x u_1(y)dy$$

and then, given this result as input to the second integrator it would compute:

$$u_2(x) = \int_0^x p(y)g_1(y)dy$$

Kelvin then realised that if the result of the second integration was continuously fed into the first integrator, this would "compel" u_1, which is u, to be the same as u_2, and thus effectively solve the problem; u_2 becomes u, and one iteration of the process is enough[7] (Goldstine, 1973, 50). This realisation was so important that it would end up being the key idea underlying analog computing throughout its use, up to the 1980's. In Kelvin's words:

> [T]hen came a pleasing surprise. Compel agreement between the function fed into the double machine and that given out by it [..] Thus I was led to a conclusion which was quite unexpected; and it seems to me very remarkable that the general differential equation of the second order with variable coefficients may be rigorously, continuously, and in a single process solved by a machine.[8]

The 'double machine' Kelvin is talking about is the two integrators combined; so what he effectively describes is the "forcing" of the output solution u_2 to become the same as the function to be solved u, by the feedback trick.

Nevertheless, Kelvin's ideas –developed around 1876– remained only theoretical, due to technological limitations. The main problem was how to transmit the result of the first integrator to the second one. Since the result of the first is measured by the rotation of a shaft attached to the vertical wheel (index-disc) which is in contact to the horizontal disc at point P (see picture in Maxwell's quote above), the torque of this shaft was not enough to transmit the motion to other shafts in order to become the input to further integrators. This problem had to wait until Bush's differential analyser built at MIT in 1931; that is, more than fifty years after Kelvin's work (Goldstine, 1973).

[7]Kelvin initially expected that the process would need a number of iterations to converge to the correct solution.
[8]Quoted in Goldstine (1973, 50).

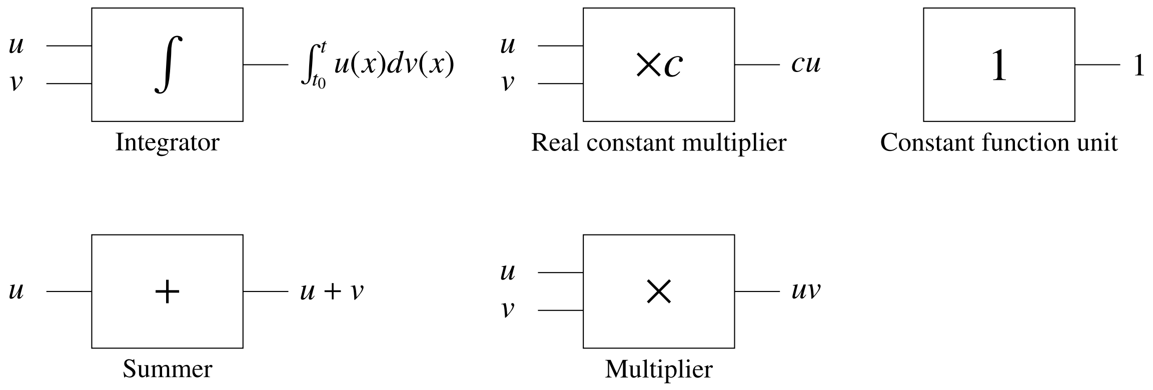


Figure 4.4: Operations performed by GPAC units

4.2 Differential Analysers and the General Purpose Analog Computer

The first general purpose analog computer was built under the supervision of Vannevar Bush, in the 1930's at MIT, known as 'Bush's Differential Analyser'. This was a mechanical analog computer, mainly based on the disc and wheel integrator we saw above.[9] Bush's student, Claude Shannon, developed the mathematical theory of what can be computed by differential analysers in principle. Shannon's mathematical model of the computer became known as 'General Purpose Analog Computer' (GPAC). Soon, however, mechanical disc and wheel integrators were replaced by electronic parts that were able to perform the same operations faster and more reliably. The most common implementation of integrators made use of Operational Amplifiers (OpAmps). We will here very briefly examine the basics of the GPAC model and see some examples of how it can be used to compute solutions to differential equations.

The GPAC can be conceptualised as a circuit consisting of a finite number of 'black boxes', able to perform the following operations on functions of one variable (Fig.4.4).

A function y is said to be '*generated*' by a specific GPAC, $\mathcal{U}$, on some interval $I \subseteq \mathbb{R}$, if we can prescribe some initial conditions to the integrators of $\mathcal{U}$, at $x = a \in I$, such that if x is applied to every input that is not connected to an output, then y equals the output of some unit for values of $x \in I$ (Graça and Costa, 2003, 646).

Shannon (1941) showed that *a function $y(x)$ can be generated by a GPAC iff it is differen-*

[9]It seems that Bush was unaware of Kelvin's work and in essence had to rediscover the feedback technique. See, Goldstine (1973, 50).

tially algebraic; that is, if it is the solution of a polynomial with real coefficients

$$P(x, y(x), ..., y^{(n)}(x)) = 0 \tag{4.2}$$

for some $n \in \mathbb{N}$. This means that common functions, such as polynomials or trigonometric, can be GPAC-generated, whereas some less common ones, such as the Gamma function, $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$, cannot. This has been regarded in the past as showing that GPAC is a weaker notion of computability than computable analysis, because the Gamma function is known to be CA-computable. However, Pour-El (1974) showed that Shannon's original proof was incomplete, and Graça and Costa (2003) gave a refined model of the original GPAC. For the more robust class of GPACs then, –given by Graça and Costa– a stronger property holds:

> A function $y(x)$ is generated by a GPAC if it is a component of the solution $\mathbf{y} = (y_1, ..., y_n)$ of a system of ODEs
>
> $$\bar{y}' = \bar{p}(\bar{y}, t), \tag{4.3}$$
>
> with initial conditions $\bar{y}(0) = \bar{y_0}$ and where $\bar{p}$ is a vector of polynomials.

We will see a concrete example of a system like (4.3) shortly. We also mention that the requirement for all GPAC-generated functions that they satisfy (4.2), and vice-versa, holds for the refined notion of GPAC as well (Graça and Costa, 2003).

We mention in passing here that, although the Gamma function still cannot be generated by a GPAC, Graça (2004) formulated a modified notion of 'GPAC computability', based on computation *via approximations* and inspired by computable analysis, which then admits the Γ function as GPAC-computable.[10] Based on this latter notion then, Bournez et al. (2006) show that every function which is CA-computable is also GPAC-computable and vice versa. Following work (Bournez et al., 2012, 2013) has also shown that –under some restrictions– Turing machines can *efficiently* simulate General Purpose Analog Computers and vice-versa.[11]

We now turn to see two concrete examples of analog computing, via electronic differential analysers.[12] For simplicity of presentation, the second inputs to integrators are omitted at a first

[10]The authors use the term 'GPAC-computable' to refer to the adjusted notion of computation via approximations, while retain the term 'generated by a GPAC' for Shannon's original notion.

[11]For a different result that links CA-computable functions with Moore's $\mathbb{R}$-recursive functions, one can see Bournez and Hainry (2006).

[12]I draw here on the work of Ulmann (2006, 2013).

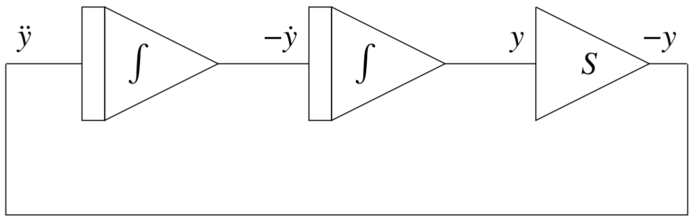


Figure 4.5: A circuit for solving $\ddot{y} = -y$

level.

Assume that we want to compute the solution to a differential equation representing some simple harmonic motion; for example, $\ddot{y} + \omega^2 y = 0$. We also assume for simplicity that $\omega = 1$. We solve for the highest derivative, so in this case we obtain:

$$\ddot{y} = -y \tag{4.4}$$

We assume in the beginning that $\ddot{y}$ is known (cf. Kelvin's method, where he assumed an initial guess for u in eq.(4.1), yielded by some iterative process). By inputting $\ddot{y}$ to an integrator, we obtain as output $\dot{y} = \int \ddot{y}dt + c_0$. Then, if we input that to a second integrator, we can obtain $y = \int \dot{y}dt + c_1$, for some constants c_0, c_1 which depend on the initial conditions. In fact, if we use operational amplifiers, then for technical reasons we obtain reversed signs, so we get $-\dot{y}$ after the first integration, and y after the second. So we also add a computational unit that changes signs at the end, after which we finally yield $-y$; that is, the right hand part of (4.4). Then, making use of Kelvin's feedback trick (p.67), we feedback the output as input to the first integrator, and this "compels" $-y$ to become equal to $\ddot{y}$ (fig.4.5).

Now without initial conditions (all inputs zero) we get the trivial solution $y(t) = 0$ and the system does nothing. But with initial conditions $c_0 = 1$ and $c_1 = 0$, the system will yield a $\cos t$ signal as the output of the first integrator and a $-\sin t$ signal as the output of the second one,[13] which, after the sign changing unit at the end, feedbacks $\sin t$ as input to the first, and so on. This complies with the analytic solution of (4.4):

$$y(t) = c_0 \sin t + c_1 \cos t$$

From a theoretical perspective now (GPAC), if we represent the output of the first integrator by u and the second integrator by v (i.e., we put $\dot{y} = u$ and $y = v$), then the circuit can be seen as

[13]Remember that OpAmps reverse the sign at the output.

satisfying the following system of ODEs:

$$\begin{aligned} u' &= -v \\ v' &= u \end{aligned}$$

that is, an instance of (4.3). With initial conditions $v(0) = 0$, $u(0) = 1$, the system is satisfied by $v = \sin t$ and $u = \cos t$. Since the function $y(t) = v(t) = \sin t$ is indeed a component of the solution of the above system, the theoretical requirement of eq.(4.3) is met; the function $y(t) = \sin t$ is GPAC-generated (as expected).

A slightly more complicated example: damped spring-mass system

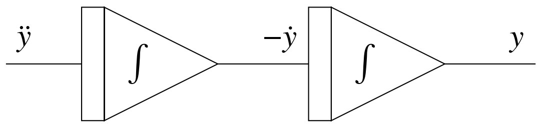
We will briefly sketch a more complex case as well. Consider a damped spring-mass system, where y the positive displacement of the mass m from equilibrium. The independent variable is again the time t. Let $F_s = -ky$ and $F_d = -a\dot{y}$, $(k, a > 0)$ be the forces, where k and a are the spring constant and damping constant, respectively. Then, by Newton's second law, the differential equation characterising the system is

$$m\ddot{y} + a\dot{y} + ky = 0, \ t > 0$$

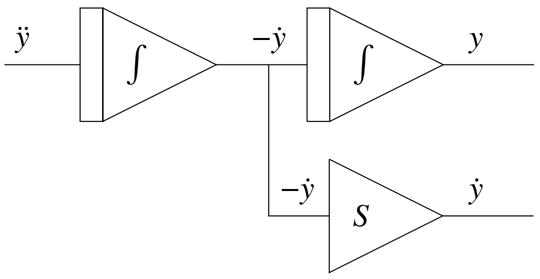
Now, to construct the analog computing circuit, we solve again for the highest derivative (as if it were known), and then we try to construct a circuit such that its final output will be the overall quantity of the right hand of the final form of the DE (so that by feedback, we will equate it to the left hand):

$$\ddot{y} = \frac{-(a\dot{y} + ky)}{m} \tag{4.5}$$

We can use two integrators to yield $-\dot{y}$ from $\ddot{y}$, and then y, exactly as before:



Then, by a multiplier unit, we can change the sign of $-\dot{y}$ to $\dot{y}$. Thus we have produced both y and $\dot{y}$; that is, all the occurrences of the function and its derivatives in the right hand of (4.5).



By using multiplying units again (e.g., potentiometers connected as voltage dividers, if the multiplying factors are between 0 and 1), we can obtain ky and $a\dot{y}$. We then add them together and we get the numerator of the right hand of (4.5), which we will multiply by $1/m$.

Finally, based on Kelvin's trick, we feedback the final result to $\ddot{y}$:

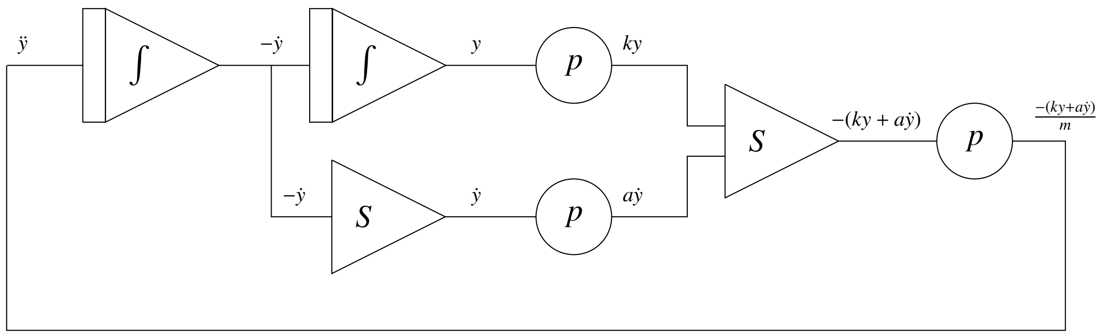


Figure 4.6: Analog computing diagram for the damped spring-mass system.

Implementation

What do integrators look like? As we have already seen, mechanical integrators were based on disc and wheel set-ups. Different implementations have also existed, based on water flow, vacuum tubes, etc. But the dominant implementation from the early 1940s onwards has been by means of operational amplifiers; leading to the development of electronic analog computers. The first such implementation was made in Germany during military research for the 3rd Reich (in particular, simulations of rocket flight).[14]

An integrator circuit based on Operational Amplifier can be seen in Fig.4.7. Its output voltage U_{out} is given by:

$$-U_{out}(t) = U_o + \int_0^t \sum_{i=1}^{n} \frac{U_i}{R_i C} dt \tag{4.6}$$

[14]See, e.g., Ulmann (2013) for some historical notes.

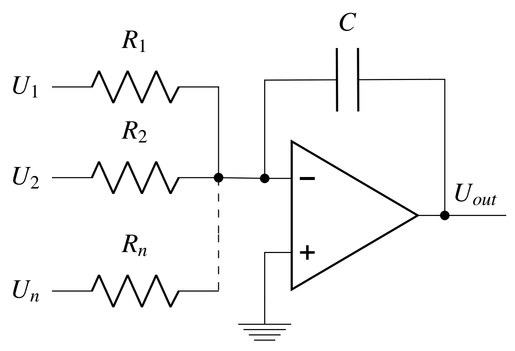


Figure 4.7: An integrator circuit. The output voltage is proportional to the integral of the sum of the input voltages, as in eq.(4.6).

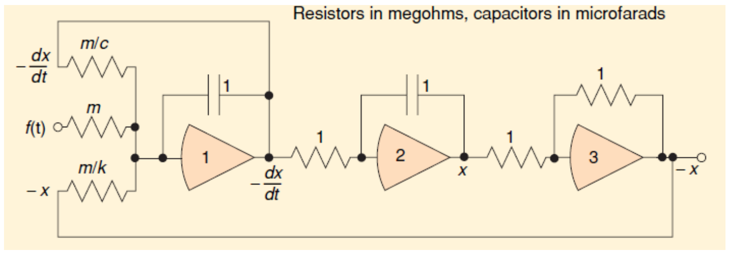


Figure 4.8: An electronic circuit implementing the analog computer of Fig.4.6 (from Howe 2005).

where U_o an initial condition. If the capacitor C is replaced by a resistor, the circuit functions as a summer: the output voltage is proportional to the sum of the input voltages. In Fig.4.8, a convenient implementation for computing the solution to the damped spring-mass system can be seen. As we have already said, integrators implemented by operational amplifiers invert the signs at the output.

Using an oscilloscope, the solution $y(t)$ and its evolution can be simulated on a screen in real time.

4.3 Accounts of 'Analog'

What makes an analog computer *analog* exactly? The aim of this section is to sketch an account for the distinction between analog and digital computation.

Generally, the distinction has been discussed in the literature relatively little, and usually in passing either in introductory chapters to computer science texts or in computing history surveys or in literature relevant to philosophy of mind issues. There are some notable exceptions, such as Goodman (1976), Lewis (1971), Haugeland (1981), Katz (2008), Maley (2011), and

Beebe (2016), which include substantial discussion of the distinction itself, but they're rather exceptions and some of them are unsatisfactory.

A number of attempts to pin down where the analog/digital difference exactly lies —especially in computer science, but also in philosophy— are focused on the computing machines themselves. Such attempts try to locate the features and structural properties of the devices that give rise to what we would characterise as 'analog' or 'digital' computation. An alternative approach bases the distinction on properties of the involved representations, instead of looking at the machines themselves. We will critically discuss both approaches.

Within the first approach, there have been two main views about the analog/digital distinction; one resting on the continuous/discrete dichotomy, and one based on the existence of analogies or not.

4.3.1 Accounts based on intrinsic properties of the computational machines.

According to the first view, the difference is based on the distinction between the continuous and the discrete. An analog computer is one which operates on continuous variables and/or has continuous internal states, whereas a digital computer operates with discrete numbers and/or discrete states (see, e.g., Jackson 1960, Mycka 2006, Piccinini 2015, ch.12). Piccinini, for example, writes:

> Analog and digital computers are best distinguished by their mechanistic properties. [..] Whereas the inputs and outputs of digital computers and their components are string of digits, the inputs and outputs of analog computers and their components are continuous or *real variables* (Pour-El, 1974) [..]
>
> The operations performed by computers are defined over their inputs and outputs. Whereas digital computers and their components perform operations defined over strings of digits, analog computers and their components perform operations on portions of real variables. Specifically, analog computers and their processing units have the function of transforming an input real variable received during a certain time interval into an output real variable that stands in a specified functional relation to the input, The discrete nature of strings makes it so that the digital computers perform digital operations on them [..], whereas the continuous change of a real variable over time makes it so that analog computers must operate continuously over time. (2015, 200-1; emphasis in original)[15]

[15]We should stress, however, that the continuous/discrete dichotomy is not sufficient for Piccinini to pin

Finally, although not explicitly stated as a definition, a similar view is implicitly taken in the technical works on GPAC by Pour-El (1974), Rubel (1989), and Bournez and Campagnolo (2008).[16] But, these particular cases should not be understood as attempts to regiment or define the notion of 'analog computation', as Piccinini mistakenly —in my opinion— takes them to be when he cites Pour-El (1974). Rather, they should be understood as just picking out the domain of application of the particular mathematical models considered in these works.

The second view grounds the analog/digital distinction in the existence or not of some *analogy*, usually between the variables operated upon in the analog computer and some physical law or the problem that is to be solved/simulated. The most characteristic formulation of this idea can be seen in a recent authoritative introduction to analog computing:

> First of all it should be noted that the common misconception that the difference between *digital computers* on one side and *analog computers* on the other is the fact that the former use discrete values for computations while the latter work in the regime of continuous values is wrong! In fact there were and still are analog computers that are based on purely digital elements. In addition to that even analog electronic analog computers are not working on continuous values — eventually everything like the integration of a current boils down to storing (i.e., counting) quantized electrons in a capacitor.
>
> If the type of values used in a computation —discrete versus continuous— is not the distinguishing feature, what else could be used to differentiate between *digital* and *analog* computers? It turns out that the difference is to be found in the structure of these two classes of machines: A digital computer in our modern sense of the word has a fixed structure concerning its constituent elements and solves problems by executing a sequence (or sequences) of instructions that implement an algorithm. These instructions are read from some kind of memory, thus a better term for this kind of computing machine would be *stored-program digital computer* since this describes both features of such a machine: Its ability to execute instructions fetched from a memory subsystem and working with numbers that are represented as streams of digits.

down the difference between analog and digital computational systems. Rather, additional factors are taken into account, such as differences in the design of the two kinds of devices, in their computational power, and in programmability/universality. However, this doesn't affect our point here; that is that Piccinini bases the analog/digital distinction on objective features of the computational devices themselves.

[16]Compare: "For as we know the distinguishing feature of an analog computer -as opposed to a digital computer- is that the variables change continuously" (Pour-El, 1974, 2).
And: "Church's thesis [..] is not usually thought of in terms of computability by a *continuous* computer, of which the general-purpose analog computer (GPAC) is a prototype" (Rubel 1989, 1011; emphasis in original).

> An analog computer on the other hand is based on a completely different paradigm: Its internal structure is not fixed — in fact, a problem is solved on such a machine by changing its structure in a suitable way to generate a *model*, a so-called *analog* of the problem. This analog is then used to *analyze* or *simulate* the problem to be solved. Thus the structure of an analog computer that has been set up to tackle a specific problem represents the problem itself while a stored-program digital computer keeps its structure and only its controlling program changes (Ulmann 2013, 2; emphasis in original)

According to this view then, the difference should be understood in terms of structural features of the two classes of computational devices. Whereas the structure of digital computers is fixed, the structure of analog ones is manipulable, so that it is made to model the particular problem at hand. Other textbooks on analog computing share similar views as well.[17]

It is easy to see what the grounds for this view of analog versus digital computing are. The dominant paradigm of analog computing during the 20th century, as we've seen, has been by using general-purpose analog computers, based on integrators. Recalling the example with the damped spring-mass system from earlier (p.72), we can see that the array of the operational amplifiers were such that the total system is governed by the same differential equation as the problem at hand; thus, making the analog computer a *model* (an *analog*), in a sense, of the problem.

The second approach to the analog/digital distinction is based not on some objective properties of the computing devices themselves but on properties of our *representation*.

4.3.2 Accounts based on representational properties.

Similarly to the previous approach, here again there are two views, roughly based either on the continuous/discrete dichotomy or on the existence or not of some kind of analogy. The difference is that in this case, the focus is on the *representation* of the quantities/solutions that are to be computed, instead of the internal states/modes of operation of the machines themselves.

Thus, according to the first view, analog computing is one which operates on continuous representations whereas digital computing operates on discrete representations. This is clearly

[17]For example: "The analog computer, on the other hand, works with continuous variables the same way as nature does. It can be used to solve particular problems or to construct a physical model of the situation under study. In this latter instance the machine variables are directly analogous to the variables in the physical system under study" (Jackson, 1960, 5). Jackson, however, bases the distinction between analog and digital primarily on the continuous/discrete dichotomy, as we've already seen.

expressed in MacLennan (2012) and Moor (1978),[18] but Goodman's (1976) and Haugeland's (1981) accounts are along similar lines as well —though more nuanced.

According to the second view, in analog computing representation of numbers is by means of some suitable physical quantities, either "whose values, measured in some pre-assigned unit, are equal to the numbers in question" (von Neumann, 1986), or which "are either primitive or almost primitive" where 'primitive' refers to terms in "some *good* reconstruction of the language of physics—good according to our ordinary standard of economy, elegance, convenience, familiarity" (Lewis, 1971, 324-25). Along similar lines, Smith (1991), Copeland (2008), and Maley (2011) define analog representations as those in which the representational medium *models* properties of the represented system, while Smith is also critical of identifying analog representations with continuous ones:

> Calling continuous representations "analogue" is both unfortunate and distracting. "Analogue" should presumably be a predicate on a representation whose structure corresponds to that of which it represents: continuous representations would be analogue if they represented continuous phenomena, discrete representations analogue if they represented discrete phenomena. That continuous representations should historically have come to be called analogue presumably betrays the recognition that, at the levels at which it matters to us, the world is more foundationally continuous than it is discrete. (Smith, 1991, 271)

It is easy to see that these latter views are again inspired by the paradigmatic case of differential analysers; recall, for instance, how in the damped spring-mass system above, the physical quantity voltage V_{out} is meant to be analogous to the displacement of the mass from equilibrium.

Finally, it is consistent with these views that an analog representation can even be discrete, and Maley (2011, 122) and Copeland (2008) offer to that effect, respectively, the examples of a jar with marbles the number of which represents the number of days passed since a given time, and of an architect's model with clear plastic squares the number of which represents the number of windows in the actual building modelled. Digital representations then are those in which there is not any kind of analogy between the represented numbers and the representing features, but, instead, numbers get represented by means of (strings of) digits.

[18]"The essential difference between digital and analogue computers is generally taken to be in the type of information processing. In a digital computer information is represented by discrete elements and the computer progresses through a series of discrete states. In an analogue computer information is represented by continuous quantities and the computer processes information continuously" (Moor, 1978, 217).

Before embarking on the evaluation of the above approaches and views, we will shortly digress to a discussion about the nature of computing, in general. I believe that a view of computing in its broadest generality would be useful when trying to account for more particular aspects of it, such as the analog/digital distinction.

4.4 A comment on the nature of computing

It seems that the advent of —both analog and digital— computers in the 20th century which, once programmed, require only an initial input in order to perform complete computations in dizzying speeds and degrees of accuracy, led to a big shift in understanding the process of computing, as a process of trying to solve a problem. At first glance, a smart-phone running some version of MATHEMATICA, my PC compiling this document in LaTeX, or even an older TELEFUNKEN analog computer simulating in real time the airflow around a wing (Fig.4.14), based on OpAmp circuits, might all seem as computing just by themselves. With minimal inputs by us, such machines seem to do all the job of delivering the results of incredibly complicated computations. This view seems so powerful and intuitive that has given rise to all kinds of computational interpretations to natural phenomena around us: from algorithmic views of biological entities and physical objects (proteins, tables, etc.) to "algorithmic views of the universe" and to computational theories of mind.

This view of the computer as a device computing on its own also underlies the first approach to the analog/digital dichotomy; that is, the one which views the dichotomy as one based on intrinsic features of the devices themselves. The motivation might seem somewhat understandable for computer scientists or philosophers of mind. If computer science is to be a science of something, then it seems desirable to have an objective and human-independent domain of application. And if the brain is a computing system, then it must be a system performing computations by itself. That might be perhaps why some philosophers try to offer accounts of computation that are grounded in mechanistic features of the computing devices themselves and dispense with any talk about representation (see, e.g., Piccinini (2015)).

Nevertheless, I think that understanding 'computing' solely in this way might be tricky and potentially misleading. To the extent that our notion of 'computing' should subsume the historical cases as well, it seems that we need a more "anthropocentric" view of it.[19] People have been computing since ancient civilizations, always using whatever tools they had available to help them, from sand, pen and paper, ruler and compasses, and abacuses to astrolabes, sundials, Napier's bones, nomography, and even the MONIAC.[20] What all these cases seem to have in

[19]I would even suggest that it is a category mistake to say that a PC, or a smart-phone, 'computes by itself'.

[20]The MONIAC was a hydraulic analog computer, built by Bill Philips in 1949, which was used to model the

common is that computing has been a process *followed by an agent* trying to obtain an answer to a specific problem, aided by some tools or machinery. It has *not* been a process which can happen on its own, or some ‘natural kind’ so to speak under which only processes meeting certain criteria would fall.

Thus, computing should rather be understood as an epistemic notion, always relative to individual agents or epistemic communities. Computing is performed by an epistemic agent who is trying to obtain an answer with respect to a certain state-of-affairs that *she* interprets as a problem. As a result, representation is an essential part of the computational process. Its function is the mapping of physical states of the computing machinery[21] to corresponding states, or values, of the (physical or mathematical) problem, to be computed; for example, the rotation angle of an astrolabe disc to the position of a celestial body in the sky. Therefore, if we want to make justice to the time-honoured practice of computing, we need to reject the approaches which give to the analog/digital distinction a more “realistic” or “objective” interpretation than appropriate. But before moving on, let me briefly consider some possible objections to the view of computing I have just defended here.

Rigorous theories of computation. Objection: This view makes the notion of ‘computation’ unacceptably open-ended and relativistic. If ‘computing’ is always a process relevant to some epistemic agent(s), then how are explications, such as Turing Machine computation, or rigorous theories, such as computability theory, possible? Answer: As argued in ch.2, ‘computing’ is a pre-theoretical notion —indeed, one with open texture— guided by paradigmatic cases. In other words, it is rather a cluster of ideas, including various cases such as ‘effective computation’, ‘analog computation’, ‘machine computation’, ‘geometric constructions’, etc. Once some such strand is picked up and theoretically “sharpened”, such as ‘effective computation’ or ‘computation by using certain units such as those in Fig.4.4’, then explications, such as *Turing machine computability* or *GPAC-computation* become possible. However, the implicit involment of an agent who interprets the results of the computation as such exist in each case of the cluster.

Turing machines, etc. Objection: If computing is always with respect to some agent seeking an answer, who is the agent in the case of a Turing machine which computes a function $f : \mathbb{N} \to \mathbb{N}$ by following its program? Isn’t this a case (model) of a machine computing by itself, with no agent involved? Answer: This is a wrong way of looking on Turing machines. A TM is actually a mathematical model of *an agent*, computing with or without the aid of some machinery. Jack Copeland has emphatically and convincingly argued for the view that TMs

British economy. One of the few machines built can be seen today in the Science Museum, in London, UK.

[21] I mean ‘machinery’ here in the broadest sense; thus, a ‘state of the machinery’ as meant here might even include the position of an arithmetical symbol in an string on a piece of paper.

are actually about *human agents* performing clerical calculations. “The Turing machine is a model, idealised in certain respects, of a human being calculating in accordance with an effective method” (2015, sc.3). As Wittgenstein also put it: “Turing’s ‘Machines’. These machines are humans who calculate.”[22] The calculating aids, used by the modelled agent in this case, are not mechanical devices or so, but what corresponds to paper (tape) and pen (read-write head). Representation is again present here, embodied in the way natural numbers are encoded by means of the finite alphabet the TM is using (at least, to the effect that we want to say that a TM is computing *a function*).

One might further object that there are mechanical implementations of TM (e.g., by LEGO), simulating faithfully how an abstract TM operates, by writing, e.g., 0s and 1s on a squared tape, and that such LEGO machines seem to compute by themselves without any agent around. This brings me to the third possible objection.

Physical computers, etc. Objection: Even if one accepts the views above, it is still very counter-intuitive to say that every time my laptop compiles this document in LaTeX , considered in and of itself, *it* does not compute. Answer: It does seem counter-intuitive, especially nowadays with devices performing all kinds of tasks around us, by following instructions written in lines of code that we call ‘algorithms’ or ‘programs’. But if computing has always been a process done by an agent with the aid of machinery (from just a table with pre-made measurements and calculations to such complicated machines as the Antikythera mechanism), then the difference is only a matter of how much aid the agent receives from the calculating machine. That is, there is a trade-off between how much effort is put by the agent themselves and how much computation burden is left to the device. But this is only a difference of degree. Calculating my position in the middle of the ocean by using astrolabes, maps, compasses and astronomical tables, or by just turning on my GPS device and reading off the output, differ only in the amount of the (clerical) work done by me. But it’s always *I* who computes in either case, and not just the GPS device in the latter case, exactly as it is not the astrolabe and maps alone that compute in the former.[23] I, the computing agent, am necessary in order to have a process of computing happening there, because it is I who gives rise to the representation relation and thus read off (interpret) the GPS’s output as about my position on earth and not as about electric currents in the device’s internal circuits.

In this regard, it might be worth seeing an excerpt from the opening two paragraphs of Piccinini’s

[22]Quoted in Copeland (2015).

[23]Unless one would be willing to accept that if somehow I connected the astrolabe to a power source and made it operate by itself, then it would be a computer machine. But this then would make the computing process boil down to a matter of who or what provides the necessary power to the machine, which seems absurd.

(2017) entry on physical computation in the *Stanford Encyclopedia of Philosophy*:

> In our ordinary discourse, we distinguish between physical systems that perform computations, such as computers and calculators, and physical systems that don't, such as rocks. [..] What is the principled difference, if there is one, between a rock and a calculator, or between a calculator and a computer?
>
> [..] what counts as a computer? If a salesperson sold you an ordinary rock as a computer, you should probably get your money back. Again, what does the rock lack that a genuine computer has?

Although an exact answer to the last question is not in the scope of this chapter, I think that the role of *representation* must definitely be emphasized in any such answer. And representation, as a kind of mapping, requires an agent (or an epistemic community) to interpret it as such. Thus, it might be true that if a salesperson sold a rock as a calculator, the customer might complain that they cannot calculate anything with it and ask for something else. But if the salesperson then slid the rock, together with a few others, over wires, and placed a frame around them, then they could give it back to the customer as an —ugly but fully functional— abacus; that is, a proper computing device. The difference, of course, between the two cases lies in the *ability to support representations of mathematical or physical structures*. While one pebble alone does not have this ability, many pebbles arranged in an abacus-like structure do.

Similarly, one wouldn't normally expect to find a domino game for sale in a computer store; nonetheless, dominoes have been used to simulate logical gates and perform computations. Representation makes all the difference in the world and is necessary.[24]

Deep Blue vs. Kasparov Taken to the extreme, the view of 'computing' I defend here implies that, strictly, Deep Blue itself wasn't really computing when it beat Kasparov in chess, in the famous games of 1997. This indeed might strike one as counter-intuitive initially, but every account has a bullet to be bitten. After a second thought, however, the claim might not seem so ludicrous. Similarly to the GPS case, whenever my PC or Deep Blue is running a chess program, it's fundamentally all about electric charges flowing through transistors, diodes, and the like. In order to see 'computation' there, a higher level of description is required, perhaps one that involves some functional (or rule-following) interpretations as well as mapping various states of the machine to stages of the computation —including mapping one particular state to the computation's output. But both this functional interpretation and the mapping representation are agent-based. Thus, Deep Blue alone did only stuff with electrons going through transistors, etc.

[24] Though, perhaps not sufficient.

It takes the existence of Kasparov himself, or the community of agents who watch(ed) the game, to *interpret* what Deep Blue was doing *as computation of* chess moves in a game.[25]

Remark A final remark is in order here. What is at issue above, really, is not whether it actually makes sense to say that a machine performs a computation or not. If a resistor in an electronic differential analyser is burnt, or a screw in a mechanical one gets loose, we may obtain incorrect results in our computation, and say that the machine 'malfunctioned' or 'miscalculated'. Machines are built in order to fulfil certain functions and purposes and they can also fail to do so. Rather, the purpose of the above discussion is to emphasise the fundamental role of an epistemic agent in all this, as well as the fundamental role of representation.

Here is another way to put the issue, drawing on Myrvold (1994). Myrvold accepts that machines can compute. In a context where he in effect identifies 'computing' with following the specific rules that govern the calculation, he says:

> Not every physical system is a machine; machines are constructed to perform some purpose, and not all of the machine's physically possible behavior will conform to that purpose. That is, it makes sense to speak of a machine functioning correctly or incorrectly.
>
> [..]
>
> Because we have a conception of the proper functioning of a machine, the way is open to apply normative concepts to the machine, and to say that it is (or isn't) following a rule.
>
> The crucial analogy [..] is between the rules used in a calculation and the rules which express the proper functioning of the machine. (1994, 32-33)

That is, *a machine computes* as long as *it faithfully follows the rules governing its proper functioning*. Put in such terms then, my point here is that it takes an epistemic agent (or community) to tell what the rule is and to interpret whether the machine *is* behaving according to the rule (or not).[26]

[25]Another way to put the issue might be by stating the obvious claim that computation is always computation *of* something. But then the view of computing as a merely syntactic process, (and there is a long-standing honourable tradition in logic, going back to Leibniz and Al-Khwārizmī, understanding computation as manipulation of symbols, i.e., as purely syntactic (see, sec.2.3)), although accurate to capture certain strands of the general notion of 'computation', is not enough to let us determine what a certain computation is about (e.g., a function value, a position on the map, or a castling move by a king). To construe a certain process *as* computing *of* something, semantics is necessary.

[26]As an aside note, this is why I believe that the existence of computers does not provide by itself a counterexample to the "Kripkesteinian paradox" (Kripke, 1982) about rule-following and private language, despite that there are philosophers who subscribe to that view.

4.5 Against the analogy view of ‘analog’

After this, rather long, digression, we return to the views about the analog/digital dichotomy. It should by now be clear why I think that the first approach —which bases the dichotomy on intrinsic properties of the devices themselves— should be rejected. Despite its initial attractiveness, owing to the big computers of the 20th century (both analog and digital), it overlooks the longest part of historical computing practice.[27]

Assuming now that the distinction becomes meaningful only at the level of representation, which view is the preferable one? Is the fundamental criterion whether the computational process operates on continuous or digital representations or whether the numbers to be computed are represented by some physical quantities? This question is more difficult, since the difference now is more subtle than before. The latter view bases the distinction on the existence of *analogies* and it seems to account for a good many of cases, mechanical and electronic.

Nevertheless, I believe that after closer inspection the former interpretation should be preferred. I will give two reasons for this. First, the analogy-based account leaves out an important class of computational processes. It seems unable to account for analog representations, such as in the long-standing field of nomography or devices like “analog clocks”, where there doesn’t seem to exist an analogy of any kind between properties of the representational and the represented system. A Smith’s chart, for example (Fig.4.3), does not seem to represent the values of the computed functions by means of any physical magnitude, and the motion of the hands of an analog clock (to use MacLennan’s 2012 example) does not purport to mimic the motion of the rotating earth or the position of the sun relative to it. One might retort here that the computed values in an analog clock are measured by means of physical quantities (angular position of gear discs etc.), but this is irrelevant to the computational process, since there is not any representational function associated with the positions and angles of the discs and gears themselves. That is, we don’t represent 3 o’clock as 90° after the vertical direction, in the same way that we represent the displacement of the mass in the spring-mass system as the voltage value. The representation relation is only between the position of the hands and marks on the face of the clock, not about angles per se. As a further example, consider the nomogram in Fig.4.1. This is definitely a device used for calculations, but again there is nothing in the computational process that rests on the fact that numerical values are related to physical lengths.

The second reason is that the analogy-based account may become problematic by characterising as ‘analog’ cases that we would intuitively be inclined to describe as digital.

[27]We should, however, exempt Ulmann from this criticism, since in the quoted passage (p.76) he explicitly restricts the scope of his distinction only to the modern notion of ‘computer’.

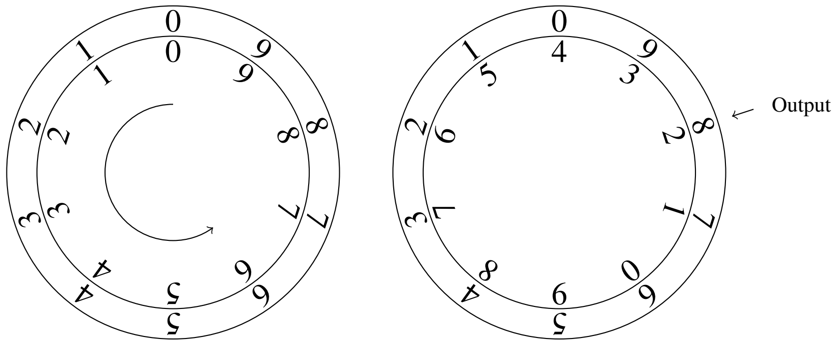


Figure 4.9: Addition with concentric discs.

We will consider an example from gearwheel computing.[28] Wheels can be used for performing addition, by adding angles of rotation. Consider two concentric wheels with numbers on them, which can freely rotate relative to each other (Fig.4.9).

Assume we want to add, say, 6 and 2. We rotate the inner disc until its zero aligns with number 6 on the outer one, and then we read off the number of the outer that is aligned with 2 on the inner (in this case, 8).

Multiplication by a constant is also easily implemented. If two discs C_A, C_B, with diameters r_A, r_B respectively, contact tangentially and can roll without slipping, then C_A will rotate at a rate of $\frac{r_B}{r_A}$ times the rotation of C_B (Fig.4.10a). If the discs are gears, with teeth of the same size, then the ratio of their angular velocities will equal the ratio of their number of teeth. This can implement multiplication by a constant; viz., by $\frac{r_B}{r_A}$ (Fig.4.10b).

Now, are the above devices all analog? For the two toothless discs in contact (Fig.4.10a), the answer by both approaches would be 'yes'. The numbers which are operated upon can take continuous values and they are represented by physical quantities (angles of rotation and ratios of diameters). And the representational medium can be said to mimic the represented state of affairs (the angles of rotation represent the first factor and the product). But what about the cogwheels of Fig.4.10b? The addition of the equal sized teeth to both wheels implies that now multiplication is only by rational numbers instead of arbitrary reals; that is, ratios of numbers of teeth. There is a sense then in which cogwheel computing performs "digital arithmetic".[29] However, the analogy-based approach would still describe this case as analog, since it is not qualitatively different from the previous one from an analogy perspective. Thus, the analogy-based approach is not able to capture the above difference, whereas the continuous-vs.-discrete approach is.

[28]I draw on Cockshott et al. (2012), for the examples here.

[29]This point is made in Cockshott et al. (2012), in a slightly different context.

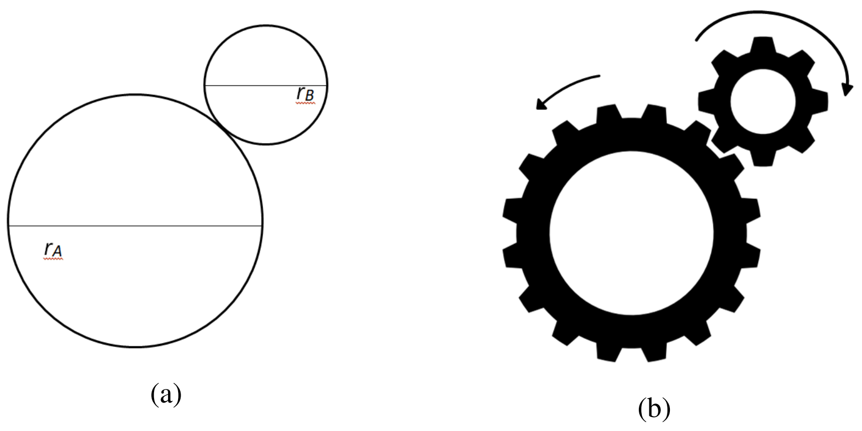


Figure 4.10: (a) Two discs in contact, implementing multiplication by a constant. The constant is equal to the ratio of their diameters, $\omega_A/\omega_B = r_B/r_A$. (b) Two cogwheels with tooth ratio 16/8 can implement multiplication by a constant (2). If the big gear rotates by, e.g., 40°, the small rotates by 80°.

In sum, when the criterion is just the existence of an analogy implemented as representation by physical quantities, cogwheel systems may qualify as analog as long as the criterion holds. This makes it difficult to see in what sense devices such as Babbage's differential engine qualify as digital —although this is how they are described, even by Copeland (2008). Even more, given that according to the same criterion the lack of analogy means digital computation, nomography should qualify as a species of digital computing. Thus, despite its initial appeal, the analogy criterion seems to give opposite answers for some paradigmatic cases of analog and digital computing. The alternative account seems more promising to capturing naturally the distinction.

4.6 Continuous vs. discrete representations

The rival view, as we have already said, grounds the analog/digital distinction in the continuous/discrete dichotomy. But this should not be understood as a some kind of an absolute or objective distinction. Rather, the most appropriate way to make sense of it would be based on the *intended reading* of the representation by us. This is perfectly consistent with the view about computing defended above, where I argued that representation, as an act of mapping performed by an epistemic agent, is fundamentally involved.

Moor (1978) has put the point aptly:

> [T]he digital or analogue characterisation is an interpretation on the symbolic level. The relevant physical feature will be abstracted. If it is a digital interpretation, continuities will be ignored; and if it is an analogue interpretation, discontinuities will be ignored. Undoubtedly, some physical systems are more easily interpreted in

> one way than the other. Nevertheless, in principle most, if not all, physical systems which might be considered to be computers could be interpreted either in digital or in analogue terms. For example, consider an early computer by Pascal which performed simple calculations by the movement of cogged wheels. Is Pascal's computer a digital or analogue machine? If we interpret the cogs in the gears as digits and understand the completed movements of the gears as discrete states, then Pascal's device is a digital computer. On the other hand, if we interpret the gears as representing a continuum of values and focus on the continuous movement, then Pascal's device is an analogue computer. (Moor, 1978, 218)

But even Turing (1950) implies that the analog/digital distinction is fundamentally dependent on the reading of the representation.

> The digital computers [..] may be classified amongst the 'discrete state machines'. These are the machines which move by sudden jumps or clicks from one quite definite state to another. These states are sufficiently different for the possibility of confusion between them to be ignored. Strictly speaking there are no such machines. Everything really moves continuously. But there are many kinds of machine which can profitably be *thought of* as being discrete state machines. (Turing 1950, 439; emphasis in original)

There are two important points made by Turing in this quote. The first —also made by Moor—[30] is that the same physical system can be represented in more than one way. This is a well-known fact in the philosophy of science,[31] but it's important to keep it in mind in this context as well, since the role and pragmatic aspects of representation is frequently neglected in the context of computing. The second point is Turing's comment that the states of a digital representation must differ to such an extent that *the possibility of confusion between them to be ignored.* This is a crucial point and was independently stressed by Nelson Goodman, to the views of whom we now turn.

But before reviewing Goodman's view, a quick remark might be in order at this point. One might object that since the analogy-based account of analog computing also is about representations (and, hence, implicitly involves agents), it can also be understood as about the intended reading of the representation as well; that is, whether we read the representation as analogous to the computed state-of-affairs, or not. However, it seems to me that nothing in the computing process per se would rest on that. On the other hand, whether we read the representation as *dense* or *differentiated* affects how we interpret the result, and so makes a

[30]But even by Ulmann as well, in defence of the analogy view; see his quote earlier, on p.76.

[31]See, also, the discussion in sec.3.4 about different representations of fluids.

difference to the computing process itself. This will become clearer after the discussion of the next section, where the terms 'dense' and 'differentiated' are also explained.

4.6.1 Goodman's account

To my knowledge, the first thorough account of the analog/digital dichotomy was formulated by Nelson Goodman in his (1976). Today it still remains the most detailed philosophical approach, although it might strike some as overly persnickety and some of its terminological choices as odd.

Goodman is concerned with what he calls 'notational systems'. These are special kinds of symbol systems, satisfying some certain syntactic and semantic requirements. A *symbol system* consists of a symbol scheme correlated with a field of reference. At a first level, one can think of this as a specific syntax (signature) correlated with a specific semantics (interpretation). As an example, a natural language, like English, can be thought of as including a symbol system, which, however, would not be a notational system. As another example, a music score would be a symbol system which would constitute a notational system as well.

There are five requirements for a symbol system to be notational; two syntactic and three semantic. Goodman uses the notion of a 'character', which is a class of certain inscriptions or utterances or marks that can be freely exchanged for one another without any syntactical effect.[32] The first requirement is that all inscriptions (tokens) of a character be "indifferent" between them. That is, two inscriptions are character-indifferent if each one belongs to a character and neither one belongs to a character that the other does not.[33]

The second requirement is that the characters be *finitely differentiated*: for every two characters A and A' and an inscription a (token, instance) which does not belong to both, it is always theoretically possible to determine either that the inscription does not belong to A or it does not belong to A'. Differentiation means that the characters are disjoint classes. But the additional requirement that differentiation be finite is also necessary. Assume, for example, a system in which only straight marks are concerned and that marks which differ by even the smallest fraction of a centimetre are defined to belong to different characters. This is a differentiated system (characters are disjoint classes) but not finitely. So no matter how precise

[32]We can understand a 'character' as a *type*, which in this case is the class of all its tokens. Although Goodman is aware but does not make use of the type/token terminology, we will take the liberty to use it here, whenever we think it is consistent with Goodman's intentions and make the presentation easier.

[33]This requirement might strike some as odd, given how used we are in computer or machine-made typing. But if one thinks of handwriting, then such requirements become clearer. For example, many Greek authors, when writing by hand in English, often use both inscriptions 'a' and 'α interchangeably for the first character of the English alphabet. Similarly, some English speakers use interchangeably 'a' and '*a*' for the same character. Character-indifference guarantees that such habits are harmless for communication purposes.

our measurements are, there will always be two different-but-close characters for which it will be impossible to determine that a certain mark does not belong to one of them. It is worth noting that this requirement in effect exists in Turing (1936) as well.

A system which possesses the completely opposite property of finite differentiation —that is, no mark can be determined to belong to just one instead of many characters— is *syntactically dense*. A dense scheme is one that provides for infinitely many characters so ordered that between any two there is a third.

Inscriptions (tokens) of characters have extensions. A musical note, for example, denotes a sound. Inscriptions may denote more than one object in the respective field of reference; for example, the English letter 'c' denotes two different sounds. The extension of the token is the class of all denoted objects. A token which denotes different objects at different times or contexts (such as the letter 'c', with respect to sound) is *ambiguous*. A character whose even one of its members (tokens) is ambiguous is also ambiguous. Therefore, in order for a character to be unambiguous, all of its tokens must be unambiguous and also have the *same* (unique) extension.

The first semantic requirement then, for a symbol system to be also notational, is that no character and no token is ambiguous (therefore, the English alphabet is not notational). The second is that all extensions are disjoint classes. This actually rules out most ordinary natural languages (consider the extensions of 'philosopher' and 'logician'; they're clearly not disjoint[34]). The third requirement is *semantic finite differentiation*: for every two characters A and A' with non-identical extensions, and every object of the domain of reference which is not denoted by both, it is always theoretically possible to determine either that the object is not denoted by A or it is not denoted by A'. This requirement parallels that of syntactic finite differentiation. And, similarly to the syntactic case, a system which violates this condition everywhere is *semantically dense*. Consider, for example, a system where the symbol scheme consists of fully reduced Arabic fractional numerals which are stipulated to denote objects of corresponding weight in fractions of a kilo. Although syntactic and semantic disjointness are both met, no matter how precise our measurements are, there will always be different characters for which it will be impossible to determine that a certain object is not denoted by them all.

Now, according to Goodman's account, *a system is analog, if it is both syntactically and semantically dense*. On the other side of the spectrum, *a system is digital, if it is syntactically and semantically finitely differentiated throughout*. If it is also unambiguous and syntactically

[34]These are actually *compound* characters. Goodman discusses in detail rules and properties of compounding atomic characters, within symbol systems, to form compound ones, but we don't need consider such details for our purposes here.

and semantically disjoint, then it will be notational (1976, 160).

Goodman gives the example of a simple pressure gauge with a circular face and a single pointer which moves clockwise as the pressure increases. If there is no symbolic scheme employed here (no marks on the dial) but the absolute position of the pointer is meant to represent the exact amount of pressure, then this is an analog system; every difference in the position of the pointer belongs to a different character (syntactically dense) and denotes a different pressure value, so there are always different positions (characters) for which it is impossible to determine that a certain pressure does not correspond to all of them. If there is some symbolic scheme on the dial (e.g., some marks or numbers), then the representation is analog or digital depending on how it is intended to be read. If, again, the absolute position of the pointer is what matters as before, the system is again analog, since syntactically and semantically dense. If, on the other hand, the representation is meant to be read in a way that each mark, or number, signifies the centre of a region such that every occurrence of the pointer in it counts as a token of the same character, and if those regions are disjoint and separated by some small gaps, and if the same features hold for the denoted pressure values as well (i.e., syntactic and semantic differentiation), then the symbol system will be a notational one, and the gauge a digital instrument.

4.7 Analog vs. digital computation: the proposed account and concluding remarks

Drawing on Goodman's theory of analog and digital symbolic systems, we are ready to formulate an account for analog and digital computing.

Computation[35] can be seen as an epistemic process evolving in time, by means of which an agent (or community) obtains an answer, aided by some machinery or instrumentation (possibly just pen and paper), with respect to a certain state-of-affairs (mathematical or physical) that she interprets as a problem. Since computing involves instrumental aid and it is a process in time, it entails changes of the physical states of the instrument as time progresses. Some of those states are mapped to corresponding states of the computed state-of-affairs, by means of a representation relation. I submit that two representation mappings are actually employed by the computing agent, which are conceptually distinct. In the first, the initial state of the problem to be computed is mapped (represented, encoded) in the initial state of the machinery

[35] Again, we are concerned here with the general, pre-theoretic, notion of 'computation', and not with some particular, proto-theoretic strand, such as effective computation or other. Therefore, this formulation here is just a broad account, and should not be understood as an attempt to explication.

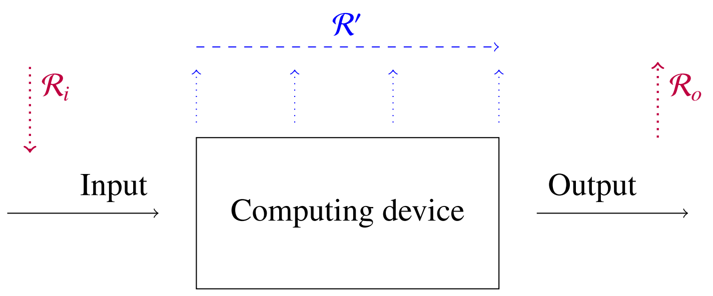


Figure 4.11: Computation involves two kinds of representation mapping. One relevant to encoding and decoding inputs and outputs (red, dotted) and one relevant to the transformation of the internal states (blue, dashed). For the input and output mappings, the mapping schemes used ($\mathcal{R}_i$ and $\mathcal{R}_o$) are not necessarily the same.

(input) and the final state of the machinery is mapped (represents, encodes) the answer to the problem (output). In the second, the agent(s) employ(s) a representation of the evolution of the computing machinery in time —i.e., the transformation of its internal states—, at a level of analysis which is relevant to the computing process.[36]

Both representations necessarily make use of some symbol system. If in both representations the symbolic systems are *interpreted* (*taken*, *used*) by the computing agent(s) as syntactically and semantically dense —that is, analog—, the computation is analog. If they are both read as syntactically and semantically finitely differentiated —that is, digital—, the computation is digital. The two representation mappings are conceptually independent processes; a fact that allows for "hybrid" cases as well.

Examples:

(1) Use of slide rules, nomographs, and planimeters constitutes instances of analog computation because (a) we normally read (use) the symbolic systems on the devices as dense,[37] and (b) we represent the transition through the states of the devices (sliding of the rule or straightedge, tracking of the boundary of the surface whose area is calculated, rotations of the disc and wheel, etc.) as dense; each absolute position marks a character and denotes an object of the reference domain.

(2) Electronic general purpose analog computers are analog because (a) we normally read the input and output values as dense (e.g., the symbolic system on the screen of the

[36]Since the second kind of representation is about the (mathematical or otherwise) description of the evolution of the physical system which is used as the computing device, it is close to the notion of 'scientific representation'.

[37]In these examples, by just 'dense', I will mean 'both syntactically and semantically dense', but will avoid the complete expression for easiness of presentation. Same holds for 'differentiated'.

oscilloscope), and (b) we represent the variation of the voltage values as continuous[38] (dense) (see, e.g., eq.4.6).

Comment: It is crucial to stress here how these cases of analog computing make it clear that we actually employ *two* distinct representations mappings, and that it is the way we *choose* to read these representations in which the analog/digital distinction is grounded. Slide rules, for example, involve a motion of the sticks that we represent as continuous. Now, if I need to explain to you how a slide rule works, this continuous (dense) representation of the sliding is *all* that actually matters for the theory of how we use these devices to compute. It might be that spacetime is fundamentally discrete and so that it's not strictly speaking correct to say that the sticks slide in a continuous, smooth manner. Still, a slide rule (or a planimeter, etc.) can be used as a computational aid exactly because the relevant representations for our purposes are in terms of continuous (dense) schemes. Similarly, we know that at a fundamental level Operational Amplifies are based on discrete phenomena (basically electrons moving around); and yet, we represent their output voltage values as continuous functions of their inputs (e.g., eq.4.6). So, again, not only is the absolute accuracy of representations not necessary for the purposes of computation here, but rather it is exactly because we choose a representation which is not absolutely accurate –yet it is convenient– that use of OpAmps *as* differential integrators becomes possible. Of course, it should already be apparent by now how the existence of representation is an *essential* feature of any computation.

(3) The mathematical GPAC model is a model of analog computation because (a) we represent the input (and output) functions operated upon by the computing units as smooth (continuous), and (b) we represent the operations performed by the units (transition through the states) as compositions of continuous functions and, hence, continuous as well (Fig.4.4).

(4) Babbage's differential engine is characterised as digital because (a) decimal numbers are represented by the positions of ten-tooth cogwheels (Fig.4.12), but the position of each tooth is not read as absolute but as belonging to the same character (numeral) within a certain region denoting the same number (extension), and (b) we are not interested in the continuous rotation of each cogwheel but in the intermediate states when each tooth is in a region denoting a number; that is, we represent the state transition as differentiated, and the same holds for the number representation. Similarly for an abacus, where we

[38]In the following I'll sometimes be using for brevity the term 'continuous' as a property of representations, but what I will mean is that the representation is syntactically and semantically dense throughout in the sense that each absolute position marks a character, and denotes a unique object in the field of reference (e.g., a number or a voltage value).

don't focus on the absolute position and movement of the pebbles on the wires, but rather on (differentiated) regions.

(5) A Turing machine is considered digital because (a) inputs and outputs are represented by means of a symbol scheme which is by stipulation syntactically differentiated and, since the domain of reference is the natural numbers, it is semantically differentiated as well, (b) transitions between states are represented by the *configuration function, $Conf$,* of the TM, which, for a given (code number of a) machine e, initial number n on the tape, and each k-th step, yields the (code number of) the state of the machine s (i.e., tape contents, position of head, and label for the quadruple to be executed next): $Conf(e, n, k) = s$ (this function is recursive). This representation is differentiated as well; we can always determine for any two different characters s and s' and any state of the machine either that the state is not denoted by s or it is not denoted by s'.

(6) For modern PCs, smart-phones, etc. we have various levels of representations. However, those relevant to computational levels of analysis —as opposed to, say, analyses at the level of the microelectronics components— are normally interpreted as employing differentiate symbol systems. We may represent computing processes at the level of machine code; that is, as operating on binary systems (which are syntactically and semantically differentiated), and evolution in time as logical operations performed by logical gates (which are again differentiated). Or, we may use representations at higher levels; such as using a floating-point arithmetic (FPA) symbolic system and transition of states as operations in FPA (i.e., again as differentiated symbolic systems). In all such cases, the computers are interpreted as digital devices.

(7) Since the two representation mappings are mutually independent —as we propose— there can be "hybrid" cases of computations. Consider again an array of OpAmps, implementing the circuit for computing the mass displacement $y(t)$ of the damped spring-mass system. $y(t)$ is represented by the value of the output voltage, and the representation is read as analog. But consider now that instead of an oscilloscope, we have connected in the end a digital voltmeter.[39] Is this an analog or a digital computing device? According to the account I put forward here, this is a hybrid case, because now the two representation mappings are of the opposite kind. We represent the output by means of a symbol system which is differentiated, but the transitions between states are still represented as continuous.

Before we close the first part, some last remarks are in order.

[39]This example was suggested to me by W. Myrvold.

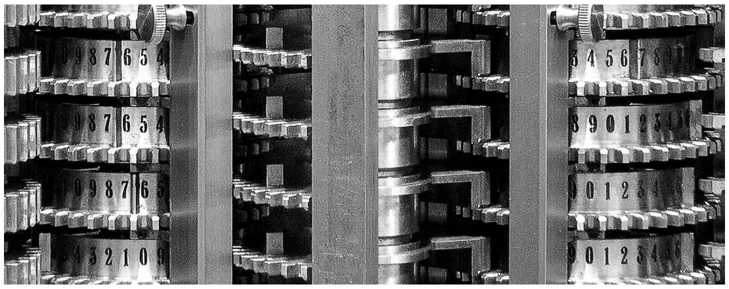


Figure 4.12: Detail of Babbage's Differential Engine (London Science Museum, UK)

- It might be argued that the fundamental difference between an analog and a digital computing system is *universality*; that is, the fact that in digital computation there is no fundamental distinction between program and data (Copeland, 2004), giving rise to the possibility of *Universal Turing Machines* (UTM) that can simulate any particular TM. This is a valid point of course, since universality —in this sense— is exhibited by digitally computing systems only. General purpose computers, analog or digital,[40] either have their respective programs stored in different units from data or programming itself is of a different nature from computing. Nevertheless, I submit that universality can be seen as a *consequence* of the properties of digital systems discussed above. Turing machines' tables and standard descriptions are couched in notational systems which, as such, have the good properties of unambiguity, definiteness, and readability. These are sufficient[41] for translating (mapping, encoding) symbolic systems from one to another. Thus, to the extent that a digital computing system employs *two* kinds of digital (notational) representations, according to our account, these can be "translated" (mapped) from one into the other, admitting, in this case, of encoding Turing programs as inputs into a UTM's tape. Thus, universality is the combined result of the two facts that digitally computing systems employ *two* representation mappings and that both mappings are based on notational systems that can be translated to each other. On the other hand, a symbolic system that is analog —or, at least, is not both syntactically disjoint and syntactically finitely differentiated— could not be used in an effective, unambiguous encoding, since we wouldn't even be able to determine which characters exactly we are mapping to which others.

- Goodman recognises that not all symbolic systems are either analog or digital. Indeed,

[40]Examples of the two cases are GPAC and Babbage's Analytical Engine (which is general-purpose but not universal in this sense), respectively.

[41]I'm not saying 'necessary', though, because their semantic components, differentiation and disjointness, are not required. The necessary properties would just be the syntactic requirements. To make meaningful encodings, however, that simulate and compute something indeed, the semantic requirements are needed as well.

since there is a number of syntactic and semantic requirements that have together to be met, for a system to fall under one of the two categories, most systems fall just in between and are neither. This is a major difference with the analog-based account of the distinction, which partitions all computing devices into two exhaustive and disjoint classes: whatever representation is not based on some analogy between the representing and represented systems is digital.

- There are many systems that can be interpreted either as analog or digital, according to their intended reading by us. An example is the cogwheel-based computing device of Fig.4.10b, implementing multiplication. (Recall, also, Moor's quote, on p.86). This is not inconsistent with our claim that in a sense this device operates on "digital arithmetic" (p.85). If we read the position of the cogwheels absolutely (densely), we have multiplication of a real number by a rational, which yields a real product (in that case, we don't have digital arithmetic). But if we read the position as differentiated —which is the more natural reading—, then the position of the first wheel must always represent a rational number, which means that the product of the multiplication will be a rational number as well (hence, digital arithmetic here). The analogy-based approach seems unable to account for the two different possibilities here.

- Recall Turing's quote on p.87 and his qualification regarding the "discrete state machines" that their states be "sufficiently different for the possibility of confusion between them to be ignored." It can definitely be seen as expressing the same requirement that Goodman would call later 'syntactic and semantic differentiation'.

- An interesting consequence of Goodman's account of analog and digital systems is that Type-2 Turing machines do not operate on a digital symbolic system. The symbolic system they operate upon is differentiated syntactically but not semantically (we saw a similar system, with Arabic fractions denoted objects of corresponding weight, on p.89). We assume here that the domain of reference is the field of computable numbers. Then, although syntactic and semantic disjointness are met, there can be cases where we have many characters (outputs of computations) for which it will be impossible to determine that a certain object (computable real number) is not denoted by them all. However, this does not hold throughout; meaning that it doesn't make the system semantically dense.

  Consider the following example. It is a theorem of Computable Analysis that if $f(x)$ is a function in the closed interval $[a, b]$, with $f(a)$ and $f(b)$ of opposite signs, then there is at least one number $c \in (a, b)$, such that $f(c) = 0$ (this is the CA-counterpart of Bolzano's theorem, which holds because each computable function is continuous). However, there is no general method for finding c; only arguments c' may be found with $|f(c')| \leqslant \epsilon$, for any $\epsilon \geqslant 0$ (Aberth, 1980, 4). This gives an example then, where it is impossible to determine that the number c is not denoted by more than one c', as is the requirement of semantic differentiation.

But this should not be surprising. The facts that neither the equality relation nor comparisons are TTE-computable, and that all computable functions are continuous are also different manifestations of the same phenomenon.

With an account of 'analog' computation at hand, we can now move on to comparing analog computing with *analogue modelling* in science.

4.8 Physical models

Analogue (or physical)[42] models are still widely in use in scientific practice today (engineering, fluid mechanics, etc.). But their fundamental importance and necessity have for the most part been ignored in mainstream philosophy of science, and they are often demoted to just demonstration purposes, as for example the use of an architect's model of a building or a molecule model made by sticks and spheres.

The use of a physical model to draw inferences regarding a target (modelled) system can be seen as a species of *analogical reasoning*. From the fact that system A possesses a number of properties $a_1, a_2, ..., a_n$ and the recognition of a similarity between all these properties and properties $b_1, b_2, ..., b_m$ of another system, we infer the similarity between a further property a_k of A and a further property b_l of B.

4.8.1 Kinds of physical models

For the kind of modelling we are interested in here, we can distinguish between two classes of physical models. The first class contains models used for demonstration purposes, such as the examples mentioned before. The second is the class of *scale models* (largely neglected in philosophy of science until recently), on which we will mainly focus in this section.[43]

The role of demonstrative physical models is representational (denotational). Depending on the purposes of demonstration, they may bear geometrical similarity to the target system (more often), or other kinds of similarity (same materials, be a working toy model of a machine,

[42] I will be using the terms 'analogue' and 'physical', for a model, pretty much interchangeably.

[43] Although the focus, aim, and motivation are different, the discussion here about analogue modelling is heavily inspired and influenced by Sterrett (2002, 2017).

etc.). Being representational, such models can be read as analog, digital, neither, or both in certain respects, based on Goodman's account. To use some of his own examples, a molecule model made by sticks and balls is (usually read as) digital, a working model of a machine may be analog, and an architect's model of a university campus may be analog with respect to dimensions and digital with respect to materials (Goodman, 1976, 173). Since the role of such models is mainly representational, there can hardly be any extra knowledge gained by their construction and use, that was not known before (at least for the agent who constructs them). They may be used to transfer knowledge between agents, but normally the one who constructs it knows already anything that can be learnt by use of the model alone.

Scale models are different. They involve more than just geometrical similarity to the target system; though, the latter is also necessary. They also involve the existence of *dynamic* (also called '*physical*') similarity. Their role is not merely denotational and, owing to the existence of dynamic similarity, genuine knowledge about the target system can be gained from their use. They are widely used in fluid mechanics, upon which we also draw for the discussion here. *Geometric similarity* exists between a model and its prototype when the model is a scale replica of the target with respect to shape and dimensions. The gain in knowledge, however, arises from the requirement for *dynamic similarity;* that is, *the assumption that the relative importance of different types of forces is the same for model and prototype.* This implies (but is not implied by) *kinematic similarity*; that is, the assumption that the flow velocity at any point in the model is proportional to the flow velocity at the corresponding point in the target system. Kinematic similarity, in turn, implies (but is not implied by) geometric similarity.[44]

4.8.2 Working example: flow around an aeroplane wing

In order to ensure dynamic similarity —i.e., that all forces in the model scale by a constant factor to corresponding forces in the prototype— we turn our attention from identifying dependence relations (functions) among all the relevant parameters of the problem, as usual, to identifying dependence relations among *dimensionless* parameters of the problem. This is achieved by performing dimensional analysis and the method is based on a fundamental mathematical result, the *Buckingham Pi Theorem*.

Informally, the Pi theorem states that for any physically meaningful relation, relating m quantities $q_l, q_2, ..., q_m$, of the form:

$$f(q_1, q_2, ..., q_m) = 0$$

[44]See, e.g., Cengel and Cimbala (2018).

there is an equivalent relation

$$F(\pi_1, \pi_2, ..., \pi_r) = 0, \ (r \leq m)$$

relating the dimensionless quantities, $\pi_1, ..., \pi_r$, that can be formed from $q_1, ..., q_m$.[45]

Consider, for example, the design of an aeroplane wing.[46] We need to examine its behaviour when the wind flows around it. More specifically, we are interested in the lift force L. We need to identify all the relevant parameters for the problem, but, *crucially*, we do not need to come up with the exact law or equation governing the phenomenon. Let's assume, for the wing case, that the relevant parameters are the speed u of the wing, the properties of the fluid flowing around the wing, that is the density ρ and viscosity μ of air, the wing's angle of attack α, and the wing's chord C (Fig.4.13).

We assume, that is, existence of a relation of the form:

$$L = f(u, \mu, \rho, C, \alpha) \tag{4.7}$$

or, equivalently,

$$f'(L, u, \mu, \rho, C, \alpha) = 0$$

Now, by dimensional analysis, we obtain the following three dimensionless parameters:

$$\pi_1 = \frac{L}{\rho u^2 C^2}, \ \pi_2 = \frac{\rho u C}{\mu}, \ \pi_3 = \alpha$$

The π_2 parameter is the *Reynolds number*, whereas π_1 is called the '*lift coefficient*'.

The Pi theorem then assures us that a functional relation, between those parameters, of the following form holds as well:

$$\frac{L}{\rho u^2 C^2} = g\left(\frac{\rho u C}{\mu}, \alpha\right) \tag{4.8}$$

for some function g.

The last equation (4.8) is extremely useful because it allows us to transfer knowledge gained by experiment on the model wing to the target system (prototype). This is owing to what has

[45]See, e.g., Logan (2013), for formal details and proof.

[46]The example we will be discussing here can be found in Zohuri (2015, ch.2) but it originally comes from the learning modules of Cimbala (2005).

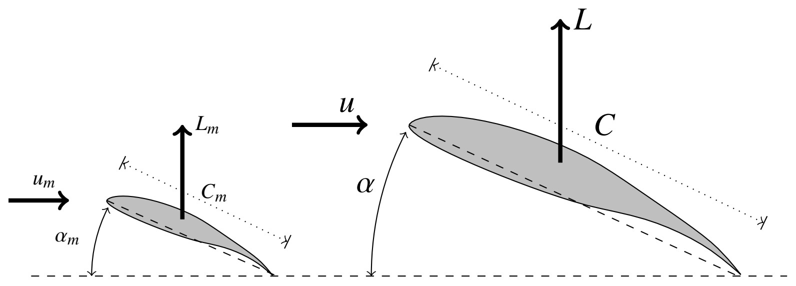


Figure 4.13: Scale model and prototype of an airfoil.

been called the *Principle of Dynamic Similarity*, which, given geometric and kinetic similarity, guarantees that *if each independent dimensionless parameter for the model is equal to the corresponding independent dimensionless parameter of the prototype, then the* dependent *dimensionless parameter for the prototype will be equal to corresponding* dependent *dimensionless parameter for the model.*[47]

So, for our case, the principle of dynamic similarity guarantees that if we build a geometrically similar model of the wing and put it in a wind tunnel at the same angle of attack as the target system and with the same Reynolds number, then the lift coefficient of the prototype (dependent dimensionless parameter) will equal the lift coefficient of the model.

That is, given that:

$$\alpha = \alpha_m$$

and that:

$$Re = Re_m \tag{4.9}$$

$$\text{i.e., } \frac{\rho u C}{\mu} = \frac{\rho_m u_m C_m}{\mu_m} \tag{4.10}$$

we have, by principle of dynamic similarity:

$$\frac{L}{\rho u^2 C^2} = \frac{L_m}{\rho_m u_m^2 C_m^2} \tag{4.11}$$

Solving (4.10) for u_m we get the required wind tunnel speed to achieve dynamical similarity (equality of angles of attack is obviously required as well). Then, (4.11) holds and we can solve

[47]See, Cimbala (2005) and Zohuri (2015, ch.2).

for L:

$$L = \frac{\rho}{\rho_m}\frac{u^2}{u_m^2}\frac{C^2}{C_m^2}L_m \qquad (4.12)$$

From (4.12), and by measuring L_m in the wind tunnel, we calculate the lift L of the real wing. The method works even for very different fluids between wind tunnel and reality (e.g., water and air), as long as the Reynolds numbers and angles of attack are held equal.

4.9 Comparison with analog computing

We are now ready to set off for the comparison between the two practices, analog computing and analogue modelling. The paradigmatic case of the former will be computing with electronic analog computers, implemented by arrays of operational amplifiers (OpAmps) (sec.4.2). For the latter, I will draw on the wing design example from above. I will be arguing that the two practices are rather orthogonal, fulfilling different epistemic purposes. Although the two practices do have features in common, my aim here is to show that they're distinct, by drawing attention to their differences.

Perhaps the easiest difference to notice from the analyses of both practices above relates to how much knowledge is required in advance, about the target system, in order to perform both methods. Crucially, the similitude[48] method does not require exact knowledge of the mathematical equations governing the modelled (or the modelling) system (Sterrett, 2002, 63). For example, we did not need knowledge of the exact functional dependence (4.7) above. Actually, this is one of the great advantages and purposes of scale modelling and similarity theory. Very often the problem is not that we know the differential equations and that they are too complicated (or impossible) to solve analytically.[49] Rather, we don't even need to know the exact form of them, in order to transfer knowledge from the scale model to the prototype. The necessary knowledge required is what parameters the phenomenon depends on (density, viscosity, etc.). Usually, these can be found based on general physical principles (often some conservation laws), fundamental assumptions (e.g., that the space is Euclidean, etc.), and perhaps some insightful use of past experience of the experimenter.

On the other hand, when it comes to analog computation, nothing less than exact knowledge of the differential equation governing the target system is adequate, in order to construct the

[48] Often called 'similarity' method by some authors as well.

[49] Although, in such a case, dimensional analysis and scaling methods (in the sense of perturbation theory) would be of crucial help as well.

circuit that computes (simulates) its solution (recall, the damped spring-mass system example, p.72).[50] This makes sense, of course: in order to 'program' a computer to compute the solution to a problem, we need to know the problem itself.

This brings us to the second main difference; namely, regarding the different epistemic purposes fulfilled by the two methods. The similitude method can be used to transfer knowledge —acquired by experiment or observation— between two systems that share some properties in common. That is, it is an instance of *analogical reasoning*. In the case of scale modelling, this knowledge (or confirmation) transfer is between two *physically ('dynamically') similar systems.* But there are cases where such transfer is between systems that are very different indeed, but they do share in common certain characteristics that are responsible for the behaviour of interest. For example, Sterrett (2017, 39.2.2) discusses the famous analogy between sound and light and how fruitful it has been in understanding the Doppler effect; in particular, how it helped Mach to identify the characteristics that are essential for the occurrence of Doppler effect and which are common to all oscillatory motion.[51] He verified his results by experimenting on analogue models with sound, where he was able to manipulate the relevant motion of the signal source and the observer. Based on the analogy, Mach transferred confirmation from the analogue experimentation to astronomical observations.

On the contrary, in the case of analog computation, we don't have transfer, per se, of newly acquired (through experiment) knowledge or confirmation. Rather, we *make use* of a certain system, for which we already have sufficient knowledge, to *compute* a solution, which also applies to another system that bears no other similarities but just obeying the same DEs. Of course, we can use this process for confirming certain hypotheses; we can use the analog computer to compute a solution and on the basis of the calculation to also confirm a certain hypothesis. Nevertheless, this is a separate, distinct, process. What we primarily do is to "ask" the analog computer to compute the solution in real time for us; further on, we might use the output for confirmation or otherwise. On the other hand, we don't use the scale model of the wing as a device to *compute* the lift force for us. Primarily, we experiment on the scale model to obtain quantities by measurement; further on, we use these results to transfer knowledge to the prototype, by means of calculations performed by *us* (eq.4.12) and not by the analogue model. It is also not easy to see how analog computing can help an agent identify the essential characteristics for the occurrence of a phenomenon, in the same way that analogue modelling helped Mach to do so about the occurrence of Doppler effect.

[50]Initial conditions are needed as well.

[51]Such characteristics were the existence of propagation in time with a finite speed, periodicity, and algebraic additivity. Crucially, the existence of a transmission medium, or properties thereof, was not one of them.

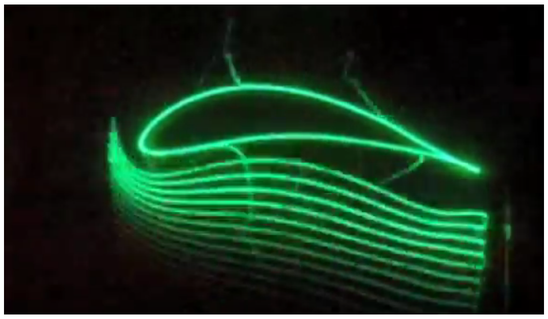


Figure 4.14: Simulation of airflow around a wing on an analog electronic computer.

I have argued in the previous sections that the apt way to look on the analog/digital distinction in computation is by means of the intended reading of the involved representation; dense or differentiated. This was in contrast to the view that regards the distinction as turning on the existence of some analogy or not. On the other hand, analogue modelling is '*analogue*' by virtue of the existence of analogies between the modelling and the modelled systems. This is the third difference between the methods. In analog computing, there is not really analogical reasoning per se; we rather exploit the fact that there is similarity in our mathematical description of both systems (same DE's), but we don't infer properties about a damped spring-mass system or an aeroplane wing based on properties of Operational Amplifiers or disc-and-wheel assemblies. Even more, a proponent of structural accounts of applied mathematics might want to claim that there are some structural similarities between a scale model wing and a real one, but it is difficult to ascribe structural similarities between an array of OpAmps simulating airflow around a wing (Fig.4.14) and a real wing in the sky.

I have also distinguished between two classes of physical models, those for demonstrative purposes only and those constructed based on similitude theory. The former can be read as analog or digital representations, since their role is usually just denotational. This is so because, as Goodman points out, we can understand such models as three-dimensional diagrams —and, hence, as symbolic systems— and read parts of them as inscriptions belonging to characters which denote parts of the prototype. On the other hand, we cannot simply regard parts of a scale model as denoting corresponding parts of the prototype, because ex hypothesi what the correspondence is really about is the *dimensionless parameters*; that is, specific *relations* of physical quantities and not just individual parameters. Since you cannot point with your finger at the Reynolds number of a scale wing, the latter cannot be read as a three-dimensional diagram, in the sense that the famous Watson and Crick's metal model of DNA can.

Finally, the two methods differ with respect to their foundations. Similitude method is based on the mathematics of dimensional analysis, such as the Buckingham Pi theorem, which, ultimately, is founded on the assumption/requirement that physics is independent of our choice of units. Generally, 'similarity' is regarded as a relative notion in philosophy, in the sense that any system might be seen as similar to any other with respect to some features. In similitude theory, the respect in which the two systems are similar is the dynamics. By keeping dimensionless parameters the same between the two systems, we force unit-independent features of the two systems (e.g., Reynolds number) to be similar. This allows us to transfer experimental results between two systems, yielded in (possibly) different units (because of the size difference), without the need for rescaling for the change of units, which would otherwise require exact knowledge of the formulas connecting the different quantities.

On the other hand, nothing of the sort exists in analog computing. Here, we might say that the counterpart of dimensional analysis and the Pi theorem is Shannon's GPAC mathematical model (or the later-improved versions thereof; see, sec.4.2). But, fundamentally, the successful implementation turns on the fact that the mathematical representations of the physical systems we employ are successful and convenient for our purposes; that is, representing the output voltage of an OpAmp as the continuous integral of the (sum of) input voltage(s) (eq.4.6) and not at a quantum level of analysis, or the air around a wing as a continuous fluid and not as a discrete medium as it really is. This is, then, the exact opposite from the similitude case. Whereas the success of similitude theory rests on the assumption that physics is independent of our representations, the success of analog computing and simulation rests on the fact (actually requirement) that we choose not the most accurate representation but the aptest one for the level of analysis we are after.

4.10 Summary and directions for further research

We distinguished between two kinds of physical models; models that serve only demonstrative purposes, such as the famous model of DNA constructed by Watson and Crick, and scale models used in similarity theory, such as scale wings in wind tunnels. We then examined the connection between analogue modelling and analog computing. To that end, we had to formulate an account of computing in general, and of analog and digital computing in particular. We argued that, generally, analog computing and analogue modelling are orthogonal practices in science, typically fulfilling different epistemic purposes. This view also underlies the use of different spelling for the two terms, throughout this work.

Analog computing in general requires more information about the target system, in order to implement it, than analogue modelling. Nevertheless, this does not necessarily mean that there is nothing to be learnt by analogue models if the exact equations governing the target system are known. There have been cases where the use of analogue models showed existence of phenomena that computational solutions had not (Sterrett, 2017, 39.4).

Furthermore, even though we argued that analog computing and analogue modelling are epistemically different, it does not follow that they always are distinct as processes. A question for further research might be the exact relation between analog computing devices and the class of physical models that we called 'demonstrative' and that we argued that can be seen as either analog or digital (or neither or both, since they in effect are representations). What is exactly the relation between the different epistemic functions —that of computation and that of modelling— in such cases? An interesting example might be the MONIAC, a hydraulic analog computer, built by Bill Philips at LSE in 1949, to model the British economy (see, also, fn.20). As a physical model, it seems to be in the class of demonstrative ones: it was meant to represent the dynamics of money flow and can be read as an analog representational system (syntactically and semantically dense). It does not seem to involve similitude theory of some sort. And it seems that in order to build one, more knowledge than just the parameters involved in some relation of the form $f(q_1, q_2, ..., q_m) = 0$ would be required; possibly the exact DEs describing the dynamics of the target system. It also seems that it can be used as a computing device as well —that is, for a different epistemic process from just a molecule replica made by sticks and balls. It can be let run and seen as solving in real time the DEs involved, in a manner similar to an electronic analog computer equipped with an oscilloscope. Systems with 'mixed' epistemic and semantic roles like that, then, might be a promising venue for further research on computational modelling and/or analogical reasoning in science.

Finally, some further related questions, worthy of investigation, are: Are *analog algorithms*, in the sense of circuits like those in sec.4.2, really 'algorithms'? How exactly do analog algorithms relate to their classical counterparts? And how do they stand with respect to foundational analyses of the classical ('digital') notion? (See, e,.g., Bournez et al. 2016). Do they exhibit different levels of abstraction as the classical ones? And how exactly do they relate with their '*quantum*' counterparts either? Can quantum algorithms be seen as instances of analog algorithms in some sense? Quantum algorithms look like circuits, similarly to the analog ones; but, there may be more similarities than that. There is no binary encoding in quantum computation and to perform a quantum simulation, the Hilbert space of the simulated system is to be mapped directly onto the Hilbert space of the (logical) qubits in the quantum computer (Kendon et al., 2010). This kind of direct mapping may seem closer to analo*gue* modes of simulation. Could then quantum computers simulating quantum systems be better thought of as analog(ue) computers (models)?

Chapter 5

Summary and Conclusions

We have examined aspects of the interplay between computing and scientific practice. As a foundational framework for this endeavour, real computability is more appropriate than ordinary computability theory, especially for considering issues of computation in scientific areas such as physics or computational modelling of physical systems. This is so because physical sciences, engineering, and applied mathematics mostly employ functions defined in (subsets of) $\mathbb{R}$. But, contrary to the case of ordinary computability, there is no universally accepted framework for real computation; rather there are two different approaches, both claiming to formalise algorithmic computation and to offer a foundation for scientific computing (computational science). We have presented the two frameworks, and the sense in which they are mutually inconsistent, in chapter 1.

In chapter 2, the underlying notion of 'algorithm' in each framework was examined. We proposed three possible interpretations of the problem that although virtually all mathematicians agree on the informal characterisation of the concept of 'algorithm', when they consider it with respect to uncountable domains of application, they explicate it in incompatible ways (sec.2.2.2). The first interpretation is that 'algorithm' is an informal but precise concept, and that only one of the two incompatible approaches captures it successfully. The second is that 'algorithm' is a concept whose pre-theoretical idea fails to pick out a single mathematical 'natural kind'; rather, the informal notion can legitimately be regimented in more than one way, consistent with the use of the concept in common-or-garden mathematics. The third is that the informal notion exhibits open texture; that is, the so far established use in the practice does not delimit it in all possible directions and there may still exist unforeseen situations in which certain routines may conform with the intuitive idea we have about it (cf., quantum or analog algorithms). We argued that the third interpretation is preferable. Then the claim was tested for its consistency with aspects of mathematical practice (certain methods that mathematicians call 'algorithms' which may not always be computable by a Turing machine; sec.2.3) as well as with key foundational analyses of the concept by Kolmogorov and Uspensky, Moschovakis, and Gurevich (sec.2.4).

Two main points have been suggested as the potential sources of the open-character of the intuitive concept (sec.2.6). First, although there is a long tradition in logic and mathematics which understands computation as a concrete process of "pushing symbols around", the way mathematicians develop algorithms is as if they are meant to describe computations over *abstract mathematical entities* (real numbers, matrix rows, points and straight lines of a plane, etc.). Although, one can make a case that such approaches illegitimately stretch the notion of computation out of its conceptual limits, the fact remains that the informal characterisation of the 'algorithm' (as presented in sec.2.2.3) does not have enough shape to rule out such views. Therefore, the dominant approach in logic and computer science, embodied in computable analysis and Turing computability (Type I or II), can be understood as capturing the view of algorithmic computation over *concrete entities* of any kind (mathematical symbols, abacus pebbles, etc.), whereas BSS/Real-RAM approaches can be seen as capturing the view of algorithmic computation as instructed operations over *abstract entities*. Our examination of Gurevich's axiomatic approach to defining the concept of 'algorithm' showed that it can encompass both views, including geometric algorithms, given Tarski's first-order axiomatisation of Euclidean geometry. Additionally, since the latter view of algorithms as routines over abstract entities may subsume methods that can not be computable by Turing machines, it is proposed (sec.2.7) that 'algorithms' be distinguished from 'effective procedures' on the basis of their possibility to be simulated by pen and paper in a finite amount of time. Thus, non Turing-implementable algorithms are not evidence against the CTT, which is a thesis about effective procedures only.

The second suggested source of the "openness" of 'algorithms' is the vagueness in the requirement of "small steps". This can give rise to two different senses of 'algorithm'; an *absolute* sense, capturing the long-standing concept in mathematics of mechanical clerk-like calculation, and a *relative* sense, according to which an algorithm is any stepwise process which is characterized as such within a certain *model of computation*, exemplified in what operations are stipulated as primitive. Again, the one tradition of computation (Effective Approximation) can be seen as capturing the former sense, whereas the other (BSS) can be seen as capturing the latter.

Crucially, it is submitted that both distinctions —*algorithm* vs. *effective procedure*, and *absolute* vs. *relative* algorithm— become apparent only when the concept is considered with respect to uncountable domains (sec.2.6–2.7). The fact that every integer can be symbolically represented on paper in a finite manner means that there is no way to distinguish between algorithms over abstract or concrete entities, and any algorithm over the integers is also an effective procedure (the reverse always holds, in any domain). Also, the fact that the domain of the integers exhibits only one level of infinity underlies a robust notion of step-by-step computation, no matter how the requirement of "small step" is actually formalised, since between any two steps of a computational process there will at most intervene a *finite* number of actions. This satisfies the implicit assumptions we identified as the constitutive ones of any conceptual framework about symbolic computation (sec.2.4.3), and so is responsible for the facts that (i) models which differ considerably on how big their admitted steps are (e.g., Turing Machines vs. KU-machines) pick out the same class of functions over the integers, and (ii) the class of computable functions that are algorithmically computable in the absolute sense is the same as those computable with respect to any reasonable model of computation (e.g., Turing-computable vs. μ-recursive functions).

In chapter 3, the claim of both approaches to real computation that they constitute the preferable framework for founding scientific computing was investigated and evaluated. We examined the advantages and disadvantages of each (sec.3.2), and argued for a pragmatic approach according to which the choice of the appropriate model is dependent on the nature of the problem we are interested in. We also discussed an account of the conditions on which the BSS/Real-RAM approach provides reliable results for analyses of algorithms, despite being an unrealistic model of computation (sec.3.3). An analogy was drawn between idealisations in computational and in mathematical models, and the required conditions were identified for such strategies in both areas to yield meaningful results.

Additionally, we concluded that BSS/Real-RAM are suitable models for analysing algorithms and problem complexities under most circumstances (sec.3.4). Nevertheless, analyses of ill-conditioned problems as well as questions about what is in principle computable by an idealised agent (machine or otherwise) —e.g., decidability of physical processes— can be addressed only within the framework of Effective Approximation (3.4).

Finally, in chapter 4, we examined the relation between analog computational models and analogue (physical) models. We distinguished between two kinds of the latter: models that serve only demonstrative purposes, such as the famous model of DNA constructed by Watson and Crick, and scale models used in similarity theory, such as scale wings in wind tunnels (sec.4.8.1).

We then examined the connection between analogue modelling and analog computing. To that end, we had to formulate an account of computing in general, and of analog and digital computing in particular. We proposed (sec.4.4) that computing is an epistemic notion, always relative to individual agents or epistemic communities. It is carried out by an epistemic agent who is trying to obtain an answer with respect to a certain state-of-affairs that the agent interprets as a problem. Representation is an essential part of the computational process, whose function is the mapping of physical states of the computing machinery to corresponding states, or values, of the (physical or mathematical) problem, to be computed.

To the extent that computing involves instrumental aid and that it is a process in time, we concluded that two, conceptually distinct, representation mappings are employed by the computing agent; one, mapping initial and final states to inputs and outputs, and one, concerning how the transformation of the internal states of the computing machinery is represented, in accordance with the level of analysis relevant to the computing process. We then put forward an account of the distinction between analog and digital computation (sec.4.7), based on the nature of the representations involved. Whenever both representations involved in a particular computation employ symbolic systems that are *read* as syntactically and semantically dense —that is, analog—, the computation is analog. Whenever both are read as syntactically and semantically finitely differentiated —that is, digital—, the computation is digital. Since, however, the two representation mappings are conceptually independent processes, "hybrid" kinds of computation can exist as well.

Based on these proposed frameworks, we were able to account for a number of paradigmatic cases of computing which are pre-theoretically characterised as analog or digital (from slide rules to modern PCs), as well as explain the property of universality which appears only in those systems that are commonly characterised as 'digital' (sec.4.7). We also drew some connections between the proposed frameworks and computable analysis (4.7).

Finally, we examined foundational aspects of both analog computing and analogue modelling (the latter as grounded in dimensional analysis and similitude theory), and argued that, generally, the two practices are orthogonal, typically fulfilling different epistemic purposes (sec.4.8–4.9).

Bibliography

Aberth, O. (1980). *Computable Anlysis*. McGraw-Hill.

Avigad, J. and V. Brattka (2014). Computability and analysis: the legacy of Alan Turing. In R. Downey (Ed.), *Turing's Legacy: Developments from Turing's Ideas in Logic*, pp. 1–47. Cambridge University Press.

Babbage, C. (1826). *On the Influence of Signs in Mathematical Reasoning*. Cambridge: Printed by J. Smith.

Beebe, C. (2016). Model-based computation. In M. Amos and A. Condon (Eds.), *Unconventional Computation and Natural Computation*, pp. 75–86. Springer International Publishing.

Blass, A. and Y. Gurevich (2003). Algorithms: A quest for absolute definitions. *Bulletin of the European Association for Theoretical Computer Science*.

Blum, L. (2004). Computing over the reals: Where Turing meets Newton. *Notices of the AMS 51*(9), 1024–1034.

Blum, L. (2012). Alan Turing and the other theory of computation. In S. Cooper and J. van Leeuwen (Eds.), *Alan Turing – His Work and Impact*, pp. 383–423. Elsevier.

Blum, L., F. Cucker, M. Shub, and S. Smale (1997). *Complexity and Real Computation*. Springer Science.

Borel, É. (1912). Le calcul des intégrales définies. *Journal de Mathématiques pures et appliquées 8*, 159–210.

Bournez, O. and M. L. Campagnolo (2008). A survey on continuous time computations. In S. Cooper, B. Löwe, and A. Sorbi (Eds.), *New Computational Paradigms. Changing Conceptions of What is Computable.*, pp. 383–423. Springer.

Bournez, O., M. L. Campagnolo, D. S. Graça, and E. Hainry (2006). The general purpose analog computer and computable analysis are two equivalent paradigms of analog computation. In

International Conference on Theory and Applications of Models of Computation, pp. 631–643. Springer.

Bournez, O., N. Dershowitz, and P. Néron (2016). Axiomatizing analog algorithms. *CoRR abs/1604.04295*.

Bournez, O., D. S. Graça, and A. Pouly (2012). On the complexity of solving initial value problems. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, pp. 115–121. ACM.

Bournez, O., D. S. Graça, and A. Pouly (2013). Turing machines can be efficiently simulated by the general purpose analog computer. In *International Conference on Theory and Applications of Models of Computation*, pp. 169–180. Springer.

Bournez, O. and E. Hainry (2006). Recursive analysis characterized as a class of real recursive functions. *Fundamenta Informaticae 74*(4), 409–433.

Braverman, M. (2008). On the complexity of real functions. arXiv:cs/0502066 [cs.CC].

Braverman, M. and S. Cook (2006). Computing over the reals: Foundations for scientific computing. *Notices of the AMS 53*(3), 318–329.

Bürgisser, P., M. Clausen, and M. A. Shokrollahi (2013). *Algebraic Complexity Theory*, Volume 315. Springer Science & Business Media.

Carnap, R. (1962). *Logical Foundations of Probability* (2nd ed.). The University of Chicago Press.

Cengel, Y. A. and J. M. Cimbala (2018). *Fluid Mechanics: Fundamentals and Applications* (4th ed.). McGraw-Hill Education.

Chabert, J.-L. (Ed.) (1999). *A History of Algorithms: From the Pebble to the Microchip*. Springer-Verlag.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*(2), 345–363.

Cimbala, J. M. (2005). Dimensional analysis and similarity. `http://www.mne.psu.edu/cimbala/Learning/Fluid/Dim_anal/dim_anal.htm`. [Accessed: 2018-05-27].

Cockshott, P., L. M. Mackenzie, and G. Michaelson (2012). *Computation and Its Limits*. New York, NY, USA: Oxford University Press, Inc.

Copeland, B. J. (2008). The modern history of computing. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2008 ed.).

Copeland, B. J. (2015). The Church-Turing Thesis. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2015 ed.).

Copeland, J. B. (2004). Computable numbers: A guide. In B. J. Copeland (Ed.), *The Essential Turing*, pp. 5–57. Clarendon Press.

Corless, R. M. and N. Fillion (2013). *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*. Springer-Verlag New York.

Cubitt, T. S., D. Perez-Garcia, and M. M. Wolf (2015). Undecidability of the spectral gap. *Nature 528*(7581), 207–211.

Demopoulos, W. (2000). On the origin and status of our conception of number. *Notre Dame Journal of Formal Logic 41*(3), 210–226.

Dershowitz, N. and Y. Gurevich (2008). A natural axiomatization of computability and proof of Church's thesis. *Bulletin of Symbolic Logic 14*(3), 299350.

DiSalle, R. (2012). Analysis and interpretation in the philosophy of modern physics. In *Analysis and Interpretation in the Exact Sciences*, pp. 1–18. Springer.

Epstein, R. L. and W. A. Carnielli (2008). *Computability: Computable Functions, Logic, and the Foundations of Mathematics* (3rd ed.). Advanced Reasoning Forum.

Feferman, S. (2013). About and around computing over the reals. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 55–76. The MIT Press.

Gandy, R. (1980). Church's thesis and principles for mechanisms. In J. Barwise, H. J. Keisler, and K. Kunen (Eds.), *The Kleene Symposium*, Volume 101 of *Studies in Logic and the Foundations of Mathematics*, pp. 123 – 148. Elsevier.

Gherardi, G. (2008). Computability and incomputability of differential equations. In R. Lupacchini and G. Corsi (Eds.), *Deduction, Computation, Experiment*, pp. 223–242. Springer, Milano.

Goldstine, H. H. (1973). *The Computer from Pascal to von Neumann*. Princeton University Press.

Goodman, N. (1976). *Languages of Art* (2nd ed.). Hackett Publishing.

Graça, D. S. (2004). Some recent developments on Shannon's general purpose analog computer. *Mathematical Logic Quarterly 50*(4-5), 473–485.

Graça, D. S. and J. F. Costa (2003). Analog computers and recursive functions over the reals. *Journal of Complexity 19*(5), 644–664.

Grzegorczyk, A. (1955). Computable functionals. *Fund. Math 42*(19553), 168–202.

Gurevich, Y. (1993). On Kolmogorov machines and related issues. In *Current Trends In Theoretical Computer Science: Essays and Tutorials*, pp. 225–234. World Scientific.

Gurevich, Y. (2012). What is an algorithm? In M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán (Eds.), *SOFSEM: Theory and Practice of Computer Science 7147*, pp. 31–42. Berlin: Springer.

Gurevich, Y. (2014). What is an algorithm? (revised). In A. Olszewski, B. Bartosz, and P. Urbanczyk (Eds.), *Church's Thesis: Logic, Mind and Nature*, pp. 165–186. Copernicus Center Press.

Gurevich, Y. (2015). Semantics-to-syntax analyses of algorithms. In G. Sommaruga and T. Strahm (Eds.), *Turing's Revolution: The Impact of His Ideas about Computability*, pp. 187–206. Cham: Springer International Publishing.

Haugeland, J. (1981). Analog and analog. *Philosophical Topics 12*(1), 213–226.

Hermes, H. (1969). *Enumerability, Decidability, Computability: An introduction to the theory of recursive functions* (2nd. ed.). Springer-Verlag.

Howe, R. M. (2005, June). Fundamentals of the analog computer: circuits, technology, and simulation. *IEEE Control Systems 25*(3), 29–36.

Hromkovič, J. (2003). *Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography*. Springer.

Jackson, A. S. (1960). *Analog Computation*. McGraw-Hill.

Katz, M. (2008). Analog and digital representation. *Minds and Machines 18*(3), 403–408.

Kendon, V. M., K. Nemoto, and W. J. Munro (2010). Quantum analogue computing. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences 368*(1924), 3609–3620.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Ko, K.-I. (1991). *Complexity Theory of Real Functions*. Cambridge, MA, USA: Birkhauser Boston Inc.

Kolmogorov, A. N. and V. A. Uspenskii (1963). On the definition of an algorithm. *American Mathematical Society Translations 29*, 217–245.

Kripke, S. A. (1982). *Wittgenstein on Rules and Private Language*. Harvard University Press.

Lacombe, D. (1955). Extension de la notion de fonction recursive aux fonctions dune ou plusieurs variables reelles. *Comptes Rendus Hebdomadaires Des Seances De L' Academie Des Sciences 240*(26), 2478–2480.

Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press.

Lemoine, É. (1902). *Géométrographie: ou Art des constructions géometriques*, Volume 18 of *Physico-Mathématique*. C. Naud.

Lewis, D. (1971). Analog and digital. *Noûs 5*(3), 321–327.

Logan, J. D. (2013). *Applied Mathematics* (4th ed.). John Wiley & Sons.

MacLennan, B. J. (2012). Analog computation. In R. A. Meyers (Ed.), *Computational Complexity: Theory, Techniques, and Applications*, pp. 161–184. New York, NY: Springer New York.

Maddy, P. (1992). Indispensability and practice. *The Journal of Philosophy 89*(6), 275–289.

Maddy, P. (1997). *Naturalism in Mathematics*. Oxford University Press.

Malc'ev, A. I. (1970). *Algorithms and Recursive Functions*. Wolters-Noordhoff Pub. Co Groningen. Translated from the Russian ed. by Leo F. Boron, with the collaboration of Luis E. Sanchis, John Stillwell and Kiyoshi Iseki.

Maley, C. J. (2011). Analog and digital, continuous and discrete. *Philosophical Studies 155*(1), 117–131.

Maxwell, J. C. (2011). Description of a new form of the platometer, an instrument for measuring the areas of plane figures drawn on paper. In W. D. Niven (Ed.), *The Scientific Papers of James Clerk Maxwell*, Volume 1. Dover.

Mazur, S. (1963). *Computable Analysis*. Instytut Matematyczny Polskiej Akademi Nauk.

Moor, J. H. (1978). Three myths of computer science. *The British Journal for the Philosophy of Science 29*(3), 213–222.

Moschovakis, Y. N. (1984). Abstract recursion as a foundation for the theory of algorithms. In E. Börger, W. Oberschelp, M. M. Richter, B. Schinzel, and W. Thomas (Eds.), *Computation and Proof Theory: Proceedings of the Logic Colloquium held in Aachen, July 18–23, 1983 Part II*, pp. 289–364. Springer-Verlag, Berlin.

Moschovakis, Y. N. (1997). Interview with G. Evangelopoulos (in Greek). *Quantum 4*(4). Accessed [Feb, 2018] from Moschovakis's webpage: http://www.math.ucla.edu/˜ynm/.

Moschovakis, Y. N. (1998). On founding the theory of algorithms. In H. G. Dales and G. Oliveri (Eds.), *Truth in Mathematics*, pp. 71–104. Clarendon Press, Oxford.

Moschovakis, Y. N. (2001). What is an algorithm? In B. Engquist and W. Schmid (Eds.), *Mathematics Unlimited — 2001 and Beyond*, pp. 929–936. Springer.

Moschovakis, Y. N. and V. Paschalis (2008). Elementary algorithms and their implementations. In S. B. Cooper, B. Löwe, and A. Sorbi (Eds.), *New Computational Paradigms: Changing Conceptions of What is Computable*, pp. 87–118. New York, NY: Springer New York.

Mostowski, A. (1979). Thirty years of foundational studies - Lectures on the development of mathematical logic and the study of the foundations of mathematics in 1930–1964. In A. Mostowski (Ed.), *Foundational Studies Selected Works*, Volume 93 of *Studies in Logic and the Foundations of Mathematics*, pp. 1 – 176. Elsevier.

Mycka, J. (2006). Analog computation and Church's thesis. In A. Olszewski, J. Wolenski, and R. Janusz (Eds.), *Church's Thesis After 70 Years*, pp. 331–352. Ontos Verlag.

Myrvold, W. C. (1994). *Constructivism, Computability, and Physical Theories*. Ph. D. thesis, Boston University.

Myrvold, W. C. (1995). Computability in quantum mechanics. In W. D. Pauli-Schimanovich, E. Köhler, and F. Stadler (Eds.), *Vienna Circle Institute Yearbook*, pp. 33–46. Kluwer Academic Publishers.

Norton, J. D. (2012). Approximation and idealization: Why the difference matters. *Philosophy of Science 79*(2), 207–232.

Pégny, M. (2016). How to make a meaningful comparison of models: The Church–Turing thesis over the reals. *Minds and Machines 26*(4), 359–388.

Piccinini, G. (2015). *Physical Computation: A Mechanistic Account*. Oxford University Press, USA.

Piccinini, G. (2017). Computation in physical systems. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2017 ed.).

Pincock, C. (2012). *Mathematics and Scientific Representation*. Oxford University Press.

Pincock, C. (2014). How to avoid inconsistent idealizations. *Synthese 191*(13), 2957–2972.

Plouffe, S. (1998). The computation of certain numbers using a ruler and compass. *Journal of Integer Sequences 1*(98.1).

Poincaré, H. (1905). *Science and Hypothesis*. Science Press.

Post, E. L. (1936). Finite combinatory processes - formulation. *Journal of Symbolic Logic 1*(3), 103–105.

Pour-El, M. B. (1974). Abstract computability and its relation to the general purpose analog computer (some connections between logic, differential equations and analog computers). *Transactions of the American Mathematical Society 199*, 1–28.

Pour-El, M. B. and J. Caldwell (1975). On a simple definition of computable function of a real variable-with applications to functions of a complex variable. *Mathematical Logic Quarterly 21*(1), 1–19.

Pour-El, M. B. and I. Richards (1989). *Computability in Analysis and Physics*. Berlin: Springer-Verlag.

Preparata, F. P. and M. I. Shamos (1985). *Computational Geometry: An Introduction*. New York, NY, USA: Springer-Verlag New York, Inc.

Rogers, Jr., H. (1987). *Theory of Recursive Functions and Effective Computability*. Cambridge, MA, USA: MIT Press.

Rubel, L. A. (1989). Digital simulation of analog computation and Church's thesis. *Journal of Symbolic Logic 54*(3), 1011–1017.

Shannon, C. E. (1941). Mathematical theory of the differential analyzer. *Journal of Mathematics and Physics 20*(1-4), 337–354.

Shapiro, S. (2006). Computability, proof, and open-texture. In A. Olszewski, J. Wolenski, and R. Janusz (Eds.), *Church's Thesis After 70 Years*, pp. 420–455. Ontos Verlag.

Shapiro, S. (2013). The open texture of computability. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 153–181. The MIT Press.

Smith, B. C. (1991). The owl and the electric encyclopedia. *Artificial Intelligence 47*(1), 251–288.

Smith, P. (2013). *An Introduction to Gödel's Theorems* (2nd. ed.). Cambridge University Press.

Sterrett, S. G. (2002). Physical models and fundamental laws: Using one piece of the world to tell about another. *Mind and Society 3*(1), 51–66.

Sterrett, S. G. (2017). Experimentation on analogue models. In L. Magnani and T. Bertolotti (Eds.), *Springer Handbook of Model-Based Science*, pp. 857–878. Cham: Springer International Publishing.

Tarski, A. (1959). What is elementary geometry? In L. Henkin, P. Suppes, and A. Tarski (Eds.), *The Axiomatic Method*, Volume 27 of *Studies in Logic and the Foundations of Mathematics*, pp. 16 – 29. Elsevier.

Thomas, W. (2015). Algorithms: From Al-Khwarizmi to Turing and beyond. In G. Sommaruga and T. Strahm (Eds.), *Turing's Revolution: The Impact of His Ideas about Computability*, pp. 29–42. Springer International Publishing.

Tisza, L. (1963). The conceptual structure of physics. *Rev. Mod. Phys. 35*, 151–184.

Traub, J. F., H. Woźniakowski, and G. W. Wasilkowski (1988). *Information-based Complexity*. Academic Press.

Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 42*(1), 230–265.

Turing, A. M. (1948). Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics 1*(1), 287–308.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind 59*(236), 433–60.

Ulmann, B. (2006). Analog and hybrid computing. `http://www.analogmuseum.org/library/anhyb.pdf`. [Accessed: 2018-05-27].

Ulmann, B. (2013). *Analog Computing*. Walter de Gruyter.

Uspensky, V. and A. Semenov (1993). *Algorithms: Main Ideas and Applications*, Volume 251. Kluwer Academic Publishers. Translated from the Russian by A. Shen.

von Neumann, J. (1986). *The Computer and the Brain* (3rd ed.). Yale University Press.

Waismann, F. (1945). Verifiability. In *Proceedings of the Aristotelian Society*, Volume 19.

Wang, H. (1997). *A Logical Journey: From Gödel to Philosophy*. The MIT Press.

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

Woźniakowski, H. (1999). Why does information-based complexity use the real number model? *Theoretical Computer Science 219*(1), 451 – 465.

Zohuri, B. (2015). *Dimensional Analysis and Self-Similarity Methods for Engineers and Scientists*. Springer, Cham.

Curriculum Vitae

Filippos Papagiannopoulos

EDUCATION

2018: Ph.D. in Philosophy
University of Western Ontario, London, Canada.

2012: M.A. in History and Philosophy of Science and Technology (two years, inter-university program)
National and Kapodistrian University of Athens, Department of History and Philosophy of Science, and National Technical University of Athens, School of Applied Mathematics and Physics, Greece.

2010: BSc/MSc in Applied Mathematics and Physics (five-year program, equivalent to M.Sc. degree)
National Technical University of Athens, School of Applied Mathematics and Physics, Greece.
Specialization: Applied Physics (Optoelectronics and Advanced Technological Materials)

PUBLICATIONS

Editorial work
(with M. Cuffaro) (Eds.) *Proceedings of the 9th International Workshop on Physics and Computation, 2018, EPTCS,* Vol. 273

SCHOLARSHIPS AND AWARDS

2018: Faculty of Arts and Humanities Graduate Thesis Research Award
2017: Faculty of Arts and Humanities Graduate Thesis Research Award
2016: Province of Ontario Graduate Scholarship
2014: Rotman Institute of Philosophy Doctoral Scholarship
2014: Western University Graduate Research Scholarship

CONFERENCE ACTIVITY/PARTICIPATION

Conferences organised
2018: 9th International Workshop on Physics and Computation (with M. Cuffaro). IUT de Fontainebleau, Fontainebleau, France, June 25-29 (Satellite to the 17th International Conference on Unconventional Computation and Natural Computation).

2015-2018: Philosophy of Logic, Mathematics, and Physics Graduate Conference (LMP), University of Western Ontario (Reviewer & co-organiser)

Contributed talks (peer-reviewed)
2018: “Algorithms and Real Computation: A Quest for Foundations”, Twenty-Sixth Biennial Meeting of the Philosophy of Science Association, November 1-4, Seattle, USA

2018: “Remarks on the Incompatibility of Foundational Analyses of Scientific Computing”, British Society for the Philosophy of Science Annual Conference, July 4-6, Oxford, UK

2017: "Incompatible Models of Computation over the Reals and Their Importance for Scientific Computing", European Philosophy of Science Association Biennial Meeting. September 6-9, Exeter, UK

2017: "Rival Models of Computation over the Reals and Their Importance for Scientific Computing", British Society for the Philosophy of Science Annual Conference, July 13-14, Edinburgh, UK

2017: "Real Number Algorithms: Incompatible Foundations" Triennial International Conference of the Italian Society for Logic and Philosophy of Science, June 20-23, Bologna, Italy

2017: "The Open Texture of 'Real Number Algorithms'", Canadian Society for the History and Philosophy of Mathematics, Annual Meeting, May 28-30, Toronto, Canada

TEACHING EXPERIENCE

Teaching Assistant
2014-2018: Western University

Other (Secondary education)
2004-2014: Private tutor in physics & mathematics, Greece

PROFESSIONAL AFFILIATIONS AND ACADEMIC SERVICE

Professional Experience
2016-2017: Assistant Editor in General Philosophy of Science, *Centre for Digital Philosophy, PhilPapers Foundation*, Western University
2016-2017: Rotman Institute of Philosophy Media and Outreach Assistant, Western University

2017: Index creation of the volume: M. E. Cuffaro and S. C. Fletcher (eds.) *Physical Perspectives on Computation, Computational Perspectives on Physics*, 2018, Cambridge University Press