THE BULLETIN OF SYMBOLIC LOGIC
Volume 17, Number 3, Sept. 2011

ALAN TURING AND THE FOUNDATIONS OF COMPUTABLE ANALYSIS

GUIDO GHERARDI

Abstract. We investigate Turing's contributions to computability theory for real numbers and real functions presented in [22, 24, 26]. In particular, it is shown how two fundamental approaches to computable analysis, the so-called 'Type-2 Theory of Effectivity' (TTE) and the 'realRAM machine' model, have their foundations in Turing's work, in spite of the two incompatible notions of computability they involve. It is also shown, by contrast, how the modern conceptual tools provided by these two paradigms allow a systematic interpretation of Turing's pioneering work in the subject.

§1. Introduction. The relevance of Alan Turing's work for the rigorous treatment of the intuitive notion of 'effective calculability' is well-known. Nevertheless, the fundamental results obtained in the characterization of the computable functions of an integer variable have perhaps obscured his important achievements in computability theory for real numbers and real functions. In particular, this is the case with his famous paper '*On computable numbers with an application to the 'Entscheidungsproblem*'' [22], where the notions of a computable real number and of a computable function of a real variable play a crucial role. This is clear from the title itself: since all natural numbers (and rational numbers) are trivially computable, it is manifest that Turing's interest pertained to real numbers in a peculiar way. The characterization of the computable real numbers within the indistinct background of the generic real numbers was in fact a primary motivation for the introduction of the machines named after him as 'Turing machines', which made [22] so popular. In particular, Turing used these machines to elaborate a rigorous method to prove that the real numbers which are commonly encountered in mathematics are computable:

Received June 15, 2010.
This work was presented at the seminar 'Modi della Logica e Forme della Scienza' held at the Philosophy Department, University of Bologna, in the year 2009. I am deeply indebted to Professor Giovanna Corsi for her valuable suggestions and her continuing support for my work.

1079-8986/11/1703-0003/$4.70

> (...) I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of the zeroes of the Bessel functions, the numbers π, e, etc. (...) Although the class of computable numbers is so great, and in many ways similar to the class of real numbers,[1] it is nevertheless enumerable [22, p. 230].[2]

In the short appendix of [22], Turing outlined a proof of the coincidence of 'computable real numbers' (that is to say, 'computable' by some Turing machines) with the 'λ-definable' ones according to the notion of 'λ-definability' given by Alonzo Church. This topic was analyzed in more detail in [23], where the notion of a computable function becomes fundamental:

> (...) the computable functions introduced by the author are identical with the λ-definable functions of Church and the general recursive functions due to Herbrand and Gödel and developed by Kleene [23, p. 153].

This equivalence is the foundation of computability theory for functions of an integer variable: the different definitions studied in recursion theory are actually equivalent. The well-known Church-Turing thesis extends this identical nature to all integer functions which are 'effectively calculable' according to every possible reasonable meaning ascribable to this expression. In fact, the intuitive and natural character of Turing machines provides a convincing motivation for such an extension. At the same it provides a justification for those approaches which sound less intuitive and more artificial, as Turing himself observes:

> The identification of 'effectively calculable' functions with computable functions[3] is possibly more convincing than an identification with the λ-definable or general recursive functions. For those who take this view the formal proof of the equivalence provides a justification for Church's calculus [23, p. 153].

The situation is different for computable real functions, for which there is no universally accepted definition. In fact, the one given by Turing (which we will discuss in Section 4) is nowadays no longer in use. Many approaches to computable analysis which are based on Turing machines (such as the 'Type-2 Theory of Effectivity' and the different paradigms introduced by Markov, Banach and Mazur, Pour-El and Richards, and others) provide

[1] $\mathbb{R}$ and the class of the computable numbers are similar since both are real closed fields.

[2] The cardinality of the class of the computable real numbers cannot exceed that of the programs that compute them. As finite strings of symbols in a fixed (finite) alphabet, these programs constitute an enumerable class.

[3] That is to say, 'computable by some Turing machine'.

non-equivalent (although strictly related) definitions. There are moreover alternative approaches which are not based on (standard) Turing machines, such as the 'realRAM machine' model, which provide other completely different definitions. Nevertheless, all these approaches develop ideas and methods introduced by Turing. In this paper we will mainly focus on the connections of Turing's work with the Type-2 Theory of Effectivity (TTE), on the one hand, and the realRAM machine model, on the other hand, although our discussion might concern other related approaches as well. Some of these connections have explicitly been recognized by the founders of these paradigms. In particular, it has been claimed that [22] and [24] provide a foundation for TTE, whereas the techniques developed in [26] are more related to the real-RAM model. The purpose of this paper is to analyze in detail the reasons for these claims. Actually, both [22] and [26] deal with computability of real numbers and of real functions, but they do it from two radically different perspectives: while [26] deals with a particular computational problem (the search of solutions for linear equation systems), [22] is a theoretical investigation on the general notion of computability.

§**2. Real numbers as 'infinite' objects.** One of the fundamental goals of [22] is the rigorous treatment of the computational processes producing the decimal expansions of real numbers.

In the physical world calculations are performed by individual human beings or by different mechanical devices, and therefore the single operations executed during any computational process might depend on several factors, such as the learned calculation method, the individual attitudes, or the characteristic of the selected computing machine. In order to determine a general notion of computability, some sort of 'standardization' of the computing activity is required. Turing found a solution in introducing the abstract and simple devices that are known after him as 'Turing machines'. Turing explained the way in which these machines work in detail, and argued that they can reliably simulate the human activity of computing:

> We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \dots, q_R$ which will be called 'm-configurations'. The machine is supplied with a tape (the analogue of paper) running through it, and divided into sections (called squares) each capable of bearing a 'symbol'. At any moment there is just one square, say the r-th, bearing the symbol $S(r)$ which is 'in the machine'. We may call this square the 'scanned square'. The symbol on the scanned square may be called the 'scanned symbol'. The 'scanned symbol' is the only one of which the machine is, so to speak, 'directly aware'. However, by altering its m-configuration the machine can effectively remember some of the symbols which

> it has seen (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration q_n and the scanned symbol $S(r)$. This pair q_n, $S(r)$ will be called the 'configuration': thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e., bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed (...) It is my contention that these operations include all those which are used in the computation of a number [22, pp. 231–232].

Turing's machines have tapes of infinite length in order to contain the infinitely long decimal expansions of (computable) real numbers. In particular, these machines, as originally conceived, can write two different kinds of symbols on the tape,

> (...) of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind) [22, p. 232].

The symbols of the first kind are those which are relevant for Turing as constituting the binary expansion of real numbers:

> If the machine is supplied with a blank tape and set in motion (...) the subsequence of the symbols printed by it which are of the first kind will be called the *sequence computed by the machine*. The real number whose expression as a binary decimal is obtained by prefixing this sequence by a decimal point is called the *number computed by the machine* [22, p. 232].

The binary expansions of real numbers have, without loss of generality, infinite length.[4] Moreover, finite sequences (and therefore all rational dyadic numbers) are trivial objects in terms of computability. Therefore Turing's definition of 'sequence computed by a machine' essentially concerns the relevant case, *i.e.*, the infinite sequences, as he explicitly specifies further on in the text:

> If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called *circular*. Otherwise it is said to be *circle-free*.
>
> (...) A sequence is said to be computable if it can be computed by a circle-free machine [22, p. 233].

[4] One can in fact very easily extend every finite decimal expansion of a dyadic number to an infinitely long expansion of the same number (for example, by postponing infinite occurrences of 0 to its last digit).

According to currently established terminology, the sequences computed by Turing machines are the 'outputs' of the corresponding computations. In general, an output sequence can be produced either by using only the information contained in the program, or by giving some additional information to the program. In this context, Turing is assuming the first possibility, since the tape of a circle-free machine contains no symbol when starting the computation ('*the machine is supplied with a blank tape and set in motion*'). When additional information is needed, it is provided as a sequence of symbols written on the tape at the beginning of the computation. This sequence is the input sequence (this terminology is nowadays applied both to finite and infinite lists of symbols).

2.1. The 'non-halting' problem. A fundamental reason for the enormous success of Turing machines is the very intuitive characterization of computable functions of an integer variable that can be obtained by the use of these devices. As an approximation, the characterization of such functions is usually expressed in the following terms. Roughly speaking, a partial function f from $\mathbb{N}$ into $\mathbb{N}$ is computable if there is a Turing machine which receives any finite codification of a number n as input and:

- returns a finite codification of $f(n)$ as output and then terminates the computation, when $n \in \mathrm{dom}(f)$;
- never stops the computation otherwise

(for more precise definitions, see [17, p. 49], [21, p. 11]). This conception has become standard especially as a result of the work of Stephen Kleene [11, p. 360] and Martin Davis [8, pp. 10–11]. Turing's original treatment of computable integer functions as it appears in [22] was in fact slightly different. The definition he gave actually emerged from the mathematical background of computable real numbers (that is to say, computable infinite sequences). In fact, he used computable infinite sequences to determine computable integer functions:[5]

> If γ is a computable sequence in which 0 appears infinitely often, and n is an integer, then let us define $\xi(\gamma, n)$ to be the number of figures 1 between the n-th and the $(n+1)$-th figure 0 in γ. Then $\phi(n)$ is computable if, for all n and some γ, $\phi(n) = \xi(\gamma, n)$ [22, p. 254].[6]

One interesting observation that can be deduced from this definition is that it concerns only *total* computable integer functions (although a slight

[5]And, *vice versa*, he used the computable integer functions to produce computable sequences:

> If $\phi(n)$ is a computable function whose value is always 0 or 1, then the sequence whose n-th figure is $\phi(n)$ is computable [22, p. 255].

[6]Actually, Turing says that this definition is possibly the simplest one. He also gave another equivalent definition formulated in terms of provable formulas in the Hilbert functional calculus.

modification of the definition would allow us to include also (proper) partial functions: it would be enough, for example, to let $\phi(n) = \xi(\gamma, n) - 1$ for $\xi(\gamma, n) \geq 1$ and $\phi(n) \uparrow$ when $\xi(\gamma, n) = 0$). According to the customary treatment of computable *partial* functions, which we described at the beginning of the paragraph, the property of 'partiality' is strictly connected to the problem of the existence of a halting stage for a computation run on a given input. When the computation performed by a machine never stops with an input n, this number does not belong to the domain of the corresponding computable function, otherwise the number returned as output[7] would be the value of the function for the argument n. In contrast, in [22] the question about the existence of halting stages is not as important as is usually believed. It is well true that this notion is involved in some crucial problems investigated in this paper, nevertheless it usually remains on the background. For example, the notion of a halting stage is connected to the decidability problems for which Turing proved a negative solution, such as the following:[8]

> I propose, therefore, to show that there can be no general process for determining whether a given formula $\mathcal{U}$ of the functional calculus **K** is provable, *i.e.*, that there can be no machine which, supplied with any one $\mathcal{U}$ of these formulae, will eventually say whether $\mathcal{U}$ is provable [22, p. 259].

A machine of this kind would work as a typical machine used in the literature to compute the integer functions: it receives as input a finite string and returns as output another finite string (such as 'yes' or 'no', or 0 or 1). Nevertheless, in [22] these machines are investigated in relation to the solution of decidability problems and not to the (explicit) computation of integer functions. Moreover, the functions that they would compute are always trivially total in the following sense: for all those elements for which the question is meaningful, an answer always exists (*e.g.*, every well-formed formula of the Hilbert functional calculus **K** is either provable or not). The negative results proved by Turing demonstrate that the answer cannot always be found, nevertheless, in those cases, any possible connection to the absence of halting stages in the computation is not explicitly mentioned.

---

[7]For the sake of simplicity, we are supposing that when the computation stops, an output is always obtained independently from the final state of the machine. For example, a blank tape can be interpreted as the codification of 0.

[8]In [19], E. Post formulated some objections to Turing's treatment of some undecidability problems, due to a certain lack of precision in the original definition of a Turing machine. In this respect, he even came to say:

> (...) it seems to the writer that Turing's preoccupation with computable numbers has marred his entire development of the Turing machine [19, p. 11].

The success of computable analysis along the years has proved how Turing's preoccupation with computable (real) numbers was instead a very profitable activity!

The problem of deciding whether a certain computation run on a given input eventually stops (or, equivalently, whether a computable function is defined on a given input) is known as the 'halting problem' (see [17, p. 150], [20, p. 24], [21, p. 19]). In [22], this problem has a very similar formulation but essentially a different meaning, as it primarily concerns the *real numbers*. To understand Turing's original interpretation of this problem, as it appears in [22], some preliminary notions are needed. Let the 'standard description' (S.D.) of a machine be a finite string of letters that encodes unambiguously the program of that machine according to some conventional vocabulary (an example is given in [22, §5]). Turing proved then the following statement is false:

> (...) we can invent a machine $\mathcal{D}$ which, when supplied with the S.D. of any computing machine $\mathcal{M}$ will test this S.D. and if $\mathcal{M}$ is circular will mark the S.D. with the symbol 'u' and if it is circle-free will mark it with 's' [22, p. 247].

This statement sounds very 'familiar', as its formulation is closely related to the usual one of the halting problem. Nevertheless, in the context in which is dealt with by Turing its meaning should be almost 'overturned' with respect to the usual conception. The context is, of course, the one of real numbers. More precisely, he wanted to show that the following conjecture is false:

> It may be thought that arguments which prove that the real numbers are not enumerable would also prove that the computable numbers and sequences cannot be enumerable [22, p. 246].

In particular, Turing refers to Cantor's diagonal process:

> (...) we might apply the diagonal process. "If the computable sequences are enumerable, let α_n be the n-th computable sequence, and let $\phi_n(m)$ be the m-th figure in α_n. Let β be the sequence with $1 - \phi_n(n)$ as its n-th figure. Since β is computable, there exists a number K such that $1 - \phi_n(n) = \phi_K(n)$ for all n. Putting $n = K$, we have $1 = 2\phi_K(K)$, *i.e.*, 1 is even. This is impossible. The computable sequences are therefore not enumerable." [22, p. 246].

Nevertheless:

> The fallacy in this argument lies in the assumption that β is computable [22, p. 246].

In fact, it might appear at first sight that β is obtained by the application of an algorithm, but this is not true. The definition of β is algorithmic only in the second part, which determines its figures, given the values $\phi_n(m)$, *i.e.*, given the enumeration of computable real numbers. This enumeration is actually the non-effective part of the definition, since in reality there is no computable method to enumerate all computable real numbers. Turing expresses this fact by using the following terminology. Let a 'description number' (D.N.) of a machine be the natural number that is obtained by assigning to each letter of the S.D. of that machine a fixed natural number (see [22, §5] for more details). Moreover, let the number $n \in \mathbb{N}$ be 'satisfactory' when it is the D.N. of a circle-free machine. An alternative formulation of the negative result proved by Turing is then that:

> (...) there can be no general process for determining whether a given number is satisfactory or not [22, p. 241].

Therefore,

> (...) the problem of enumerating computable sequences is equivalent to the problem of finding out whether a given number is the D.N. of a circle-free machine, and we have no general process for doing this in a finite number of steps[9] [22, p. 246].

Hence, in [22], the halting problem is conceived as a sort of 'non-halting problem': to determine which machine produce an infinite sequence by a non-halting computation (rather than determining those which eventually terminate the computation).

Actually, Turing's argument based on the Cantor diagonal process that we have quoted above provides an immediate proof that there is no computable enumeration of all computable real numbers: if any such enumeration existed, we would computably find, through the diagonal process, a new computable real number β which is not in the given enumeration.[10] Arguments of this type give standard proofs of the impossibility of computably enumerating all computable real numbers, and they can easily be adapted to other important codifications of real numbers, such as the Cauchy representation (see [28, Theorem 4.2.6, p. 104]) that we will discuss in the next section.

A further difference with the usual interpretation of the halting problem, which shows immediately that the existence of halting stages is not a fundamental question in [22], is the fact that the distinction between circular and circle-free machines concerns the number of digits (of the first kind) written in the output, and not the presence or the absence of a halting stage. Circular free machines are defined as those machines which write only a finite amount of symbols of the first kind, but they are not supposed to stop:

---

[9] This statement does not seem to be completely true. In principle, there could be other possible methods to enumerate all computable sequences beside the one suggested by Turing (based on the supposed decidability of the set of circle-free machines).

[10] Rigorously speaking, Turing's argument applies to computable sequences rather than computable numbers. In fact, if a computable enumeration of the computable real numbers were possible, β, intended as a number and not simply as a sequence, could already be present in the list. This is because, according to the usual definition of the binary representation, any rational number $0.w10000\ldots$, for $w \in \{0,1\}^{<\mathbb{N}}$, coincides with $0.w01111\ldots$, hence it has two binary expansions. But in reality, Turing does not apparently accept the use of bit sequences of the latter kind as binary expansions of real numbers (we will discuss this fact later). Therefore the sequence β obtained by diagonal process could possibly represent *no* real number. Although Turing is interested in computational versions of the '*arguments which prove that the real numbers are not enumerable*', he simply proves in the end that the '*computable sequences are* (...) *not enumerable*, which is a weaker result.

> A machine will be circular if it reaches a configuration from which there is no possible move, or if it goes on moving, and possibly printing symbols of the second kind, but cannot print any more symbols of the first kind [22, p. 233].

2.2. Type 1 and type 2 computability. Computability with finite objects has become the standard theory of computability, and for this reason it has been designated in the literature as '*type 1 theory of effectivity*' [28, p. 12]. According to this approach, infinite output sequences are essentially considered as products of divergent computational processes. But type 1 theory has a natural extension in the so-called '*type 2 theory of effectivity*', in short '*TTE*', which extends computability theory to infinitely long sequences.

We refer the reader to [28] for a systematic presentation of the TTE approach to computable analysis. We recall here only few basic notions. Let X and Y be $\{0,1\}^{<\mathbb{N}}$ or $\{0,1\}^{\mathbb{N}}$ (that is respectively the set of all finite and the set of all infinite sequences of bits 0 and 1) and let $F\colon\ \subseteq X \to Y$ be a (partial) function (the inclusion symbol '$\subseteq$' means that F is not necessarily total on X). We say that F is *TTE-computable* if there is some TTE–Turing machine $\mathcal{M}$ that computes it. More precisely, $\mathcal{M}$ has an input tape, a working tape, and an output tape. The input and the output tape have type specification X and Y respectively. The machine $\mathcal{M}$ receives as input a type X object and runs a computation on such input which is said to *converge* in case a type Y object is delivered as output (when Y is $\{0,1\}^{<\mathbb{N}}$ an additional requirement is that the machine eventually must halt). In particular, $\mathcal{M}$ gives as output $F(x) \in Y$ whenever $x \in X \cap \mathrm{dom}(F)$ is given as input. The machine $\mathcal{M}$ is essentially programmed like an ordinary (type 1) Turing machine but it is not allowed to make corrections on the output tape: when a digit has been written on the output tape, such digit cannot be removed or replaced. This constraint is relevant only when the output space is $\{0,1\}^{\mathbb{N}}$, and therefore it can be assumed only in this case. This has both an intuitive and a topological motivation. When the output sequence is infinite, we will never be able to obtain it *as a whole*; we are then satisfied with calculating reliable finite portions of the infinite result which at least do not contain any mistake. The nice topological consequence of this fact is that all computable functions $F\colon\ \subseteq \{0,1\}^{\mathbb{N}} \to \{0,1\}^{\mathbb{N}}$ are *continuous* (with respect to the Cantor space, that is the countable product topology on $\{0,1\}^{\mathbb{N}}$). The arity of the function is not important, since we can reduce any enumerable arity to 1 via natural tupling functions. If $X = Y = \{0,1\}^{<\mathbb{N}}$ then $\mathcal{M}$ corresponds very naturally to an ordinary (type 1) Turing machine. The type 1 machines can be then viewed as particular cases of type 2 machines.

The distinction between a working tape, where the written symbols have only a temporary function which is instrumental to the production of the output, and an output tape, where no correction/cancellation is allowed, corresponds to Turing's use of two kind of symbols. In particular, Turing specifies that the 'erase' instruction is allowed only for the symbols which have been written as temporary notes, whereas the symbols of the decimal expansion cannot be canceled:

> Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to 'assist the memory'. It will only be these rough notes which will be liable to erasure [22, pp. 231–232].

This distinction is emphasized by the convention adopted by Turing of using alternate squares:

> The convention of writing the figures only on alternate squares is very useful: I shall always make use of it. I shall call the one sequence of alternate squares F-squares and the other sequence E-squares. The symbols on E-squares will be liable to erasure. The symbols on F-squares form a continuous sequence [22, p. 235].[11]

Hence, the unique tape of an original Turing machine can be viewed as obtained by pairing the working tape (given by the E-squares, for which the erasure instruction is allowed) and the output tape (the F-squares, where no correction is allowed).[12]

According to this natural correspondence, circle-free machines can be viewed as type 2 machines with a blank input tape (or with no input tape) and a type $\{0,1\}^{\mathbb{N}}$ output tape. The machines considered in the decidability problems discussed above have (via numerical codifications) type $\{0,1\}^{<\mathbb{N}}$ both for the input and the output tape (in a word, they are common type 1 machines), whereas the universal machine studied by Turing, which converts the D.N. of any given circle-free machine into its infinite output, performs a type 2 computation with type $\{0,1\}^{<\mathbb{N}}$ for the input and $\{0,1\}^{\mathbb{N}}$ for the output.

§3. How should a real number be represented? The representation of a real number via its n-ary expansion in a given base n is probably the most common. According to this representation, π is $3.14159\ldots$ in base 10, i.e.,

$$3\cdot 10^0 + 1\cdot 10^{-1} + 4\cdot 10^{-2} + 1\cdot 10^{-3} + 5\cdot 10^{-4} + 9\cdot 10^{-5} + \cdots$$

[11] In this context, Turing is working with machines that compute binary expansions of real numbers, but, according to E. Post, Turing tacitly assumed that this convention holds for generic Turing machines:

> This convention is very useful in practice. However the actual performance (...) of the universal computing machine (...) strongly suggests that Turing makes this convention part of the definition of an arbitrary machine [19, p. 8].

[12] [22, §9] also seems to admit the possibility of turning some symbols directly into others without the need of intermediate erasure steps. Although he does not specify which symbols are liable to such change, we can reasonably assume that they are again only those of second type.

or 11.00100100001 ... in base 2, i.e.,

$$1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 10^{-1} + 0 \cdot 10^{-2} + 1 \cdot 10^{-3} + 0 \cdot 10^{-4} + 0 \cdot 10^{-5} + \\ + 1 \cdot 10^{-6} + 0 \cdot 10^{-7} + 0 \cdot 10^{-8} + 0 \cdot 10^{-9} + 0 \cdot 10^{-10} + 1 \cdot 10^{-11} + \cdots$$

In [22] base 2 is used.

In general, a *representation* of a set X is a codification of the elements of X through infinite sequences of symbols from a fixed alphabet, say $\{0, 1\}$. In other words, we use infinite sequences from $\{0, 1\}^{\mathbb{N}}$ as '*names*' for the objects of X. Formally, a representation of a set X is a surjective (partial) function $\sigma\colon \subseteq \{0, 1\}^{\mathbb{N}} \to X$ [28]. For example, in the binary expansion representation the sequence $a_1 a_2 a_3 \cdots$, for $a_i \in \{0, 1\}$ denotes the real number

$$\alpha = \sum_{i \geq 1} a_i \cdot 2^{-i}. \tag{1}$$

As one can see from this equation, in this representation we get rid of the integer part of the real number rather than introducing auxiliary symbols such as the dot (.). In other words, it is assumed (as is usually done in the literature) that the sequence $p \in \{0, 1\}^{\mathbb{N}}$ denotes the real number $0.p$ in the interval [0,1]. As we have seen, Turing himself worked in this way with the binary expansion representation (although this is not true for other kinds of representation, as we will point out). This omission has no theoretical consequence in computability theory, since the integer part is encoded by a finite sequence of bits and all finite sequences are computably trivial objects.[13] In other words, a number $m.p$, where m is any integer, is computable if and only if $0.p$ is computable:

> A number is computable if it differs by an integer from the number computed by a circle-free machine [22, p. 233].

3.1. The unreliability of the binary expansion representation. Unfortunately, the so 'popular' n-ary expansion representation, in any given base, is definitely unsatisfactory, as simple considerations on the basic operations over the reals show immediately. Suppose for example that there could be a Turing machine which receives binary expansions of x and y as input and returns a binary expansion of $x + y$ as output. We assume as usual that $x, y \in [0, 1]$. For integer numbers the sum is usually calculated by scanning the decimal expansions from right to left, but this is not possible for generic real numbers since the expansions of x and y are unlimited on the right hand

[13] At least, this has no theoretical consequence as long as computability of single real numbers is concerned. The situation is different when real functions (with domain or range exceeding [0, 1]) are considered. Obviously, two numbers differing in the integer parts can be mapped to two radically different real numbers. Not surprisingly, when dealing with real functions, Turing included integer parts in the codings, by using a strategy that we will discuss in the corresponding part of the paper.

side. Consequently the machine can only calculate the sum by scanning the names of x and y from left to right (all other methods would only be more complicated versions of this procedure). This is exactly the method adopted in TTE. At the same time, the machine cannot wait to scan these infinitely long names completely before writing the first digit of the output, and for this reason it is in general impossible to compute the correct result. Suppose in fact that the expansion of x begins with a 0 followed by many occurrences of 1, whereas in the corresponding positions within the expansion of y only occurrences of 0 are found:

$$\begin{array}{ll} x & 011111111111111\ldots \\ y & 000000000000000\ldots \end{array}$$

The machine continues to inspect the expansions of x and y but the situation does not change. At a certain stage s the machine must select the first digit of the output. As far as we know, x could be $1/2$ and y could be 0, therefore at this stage both 0 and 1 are legitimate candidates (since $1/2 + 0 = 1/2$ can be written as $10^{\mathbb{N}}$ or $01^{\mathbb{N}}$). Suppose therefore that at stage s the figure 0 is chosen and that it is written on the tape as the first digit of the output. Unfortunately there could be a large n, of which we are still unaware, such that the name of y has a first occurrence of 1 exactly in the n-th position, whereas the name of x is actually constantly 1 after the first position:

$$\begin{array}{ll} x & 01111111111111111111\ldots \\ y & 00000000000000000001\ldots \end{array}$$

In this case, the binary expansion of $x + y$ must begin with 1, since $x + y > 1/2$, so at stage s the machine made the wrong choice by selecting 0. Alternatively, at that stage the machine could write 1 as the first output digit. Unfortunately, this choice is wrong in the case where the name of y is constantly 0 as previously supposed but the name of x contains an unexpected second occurrence of 0 at a further position, which means $x + y < 1/2$:

$$\begin{array}{ll} x & 01111111111111111110\ldots \\ y & 00000000000000000000\ldots \end{array}$$

Since corrections are not allowed on the output tape, every choice might then produce an irreversible error.

This argument has a nice topological interpretation in TTE. Given two sets X and Y (represented by σ_X and σ_Y respectively) and a function $f\colon\subseteq X \to Y$, we say that f is *computable* (with respect to those representations) if f has a *computable realization*, i.e., if there is a TTE-computable function $F\colon\subseteq \{0,1\}^{\mathbb{N}} \to \{0,1\}^{\mathbb{N}}$ such that $F(p)$ is a name of $f(x) \in Y$ (with respect to σ_Y) whenever p is a name of $x \in X$ (with respect to σ_X):

$$\begin{tikzcd}
p \in \{0,1\}^{\mathbb{N}} \arrow[r, mapsto, "F"] \arrow[d, mapsto, "\sigma_X"'] & F(p) \in \{0,1\}^{\mathbb{N}} \arrow[d, mapsto, "\sigma_Y"] \\
x \in X \arrow[r, mapsto, "f"'] & f(x) \in Y
\end{tikzcd}$$

Since every (TTE-)computable function $F\colon \subseteq \{0,1\}^{\mathbb{N}} \to \{0,1\}^{\mathbb{N}}$ is continuous (with respect to the relative topologies on $\mathrm{dom}(F)$ and $\mathrm{range}(F)$ induced by the product topology $\{0,1\}^{\mathbb{N}}$), f cannot be computable when all its realizations are discontinuous.

As we have pointed out already, the continuity of every computable $F\colon \subseteq \{0,1\}^{\mathbb{N}} \to \{0,1\}^{\mathbb{N}}$ is a direct consequence of the prohibition of doing corrections on the output. In short, continuity with respect to (subtopologies on) $\{0,1\}^{\mathbb{N}}$ corresponds to the property that any finite initial segment v of $F(p)$ is univocally determined by some initial segment w of p; whatever happens in p 'after' w must not affect the production of v. The sum, with respect to the binary representation, has no continuous realization, otherwise the first digit of the output $x + y$ would be univocally determined by some initial segments of the names of the two *addenda* x and y, which contradicts what we have proved above. Therefore the sum cannot be computable with respect to the binary expansion representation.

3.2. Turing's rejection of the binary expansion representation. Turing himself soon discovered that the n-ary expansion representation is not satisfactory and decided to replace it by another one. Nevertheless, the reason for its rejection was due not to the necessity of having a suitable representation to compute the basic real functions, but rather to the desire of having a better representation for denoting the real numbers themselves. In 1937 Turing published a short note, '*On computable numbers with an application to the 'Entscheidungsproblem'. A correction*' [24], in which he changed his definition of a computable real number, and, in doing so, he gave an example of what we now call an '*admissible representation*', as we will see. This very short but deep correction contains important ideas which were developed later by other authors in computable analysis. In spite of its importance, this note has been surprisingly underestimated, probably because it could not be adequately appreciated before the systematic foundation of computable analysis was completed.

Turing's intuition about the inadequacy of the (binary) decimal expansion representation was correct, nevertheless the argument he gave in [24] to reject it was not completely convincing. The ground of such argument is the 'sudden change of initial segments' in the binary expansions of real numbers. For example, the real numbers in the interval $[0, 1/2)$ have expansions necessarily beginning with 0, whereas the real numbers in $(1/2, 1]$ have only expansions beginning with 1. This fact implies that it is in general impossible to find an algorithm which computes a certain real number given an algorithm which computes rational approximations of that number within any determined precision. This limitation makes the decimal expansion really unsatisfactory. To prove it, Turing analyzed the following true statement, called statement '(A)':

> If we can give a rule which associates with each positive integer n two rationals a_n, b_n satisfying $a_n \leq a_{n+1} < b_{n+1} \leq b_n$, $b_n - a_n < 2^{-n}$, then there is a computable number α for which $a_n \leq \alpha \leq b_n$ for each n. (A) [24, p. 545].

Turing did not provide a complete proof for this statement, nevertheless he suggested how a proof can be obtained:

> A proof of this may be given, valid by ordinary mathematical standards, but involving an application of the principle of the excluded middle [24, p. 545].

This useful hint outlines a method that is still perfectly acceptable in contemporary computable analysis: the use of the whole power of classical logic to prove the existence or the soundness of the algorithms. In this respect, computable analysis is different from constructive mathematics. According to the latter, a proof of a mathematical statement is *constructive* when it does not use any non-constructive logical principle such as the law of the excluded middle or the *reductio ad absurdum*. In this sense, constructive proofs can be seen as mathematical 'constructions'. In computable analysis algorithms developed to compute certain mathematical objects also provide mathematical constructions, as the objects they compute are constructed step by step during the execution of the algorithm by following a sequence of imperative instructions. Nevertheless, for the constructivist a mathematical theorem is proved to be constructive only when a constructive proof of that statement can be effectively found. In contrast, there are cases in which it is impossible to find effectively an algorithm which computes a certain output, but the existence of such an algorithm can be guaranteed by the principle of the excluded middle or by the *reductio ad absurdum*. For example, in the case of statement (A) it is classical logic that assures us, through the use of the excluded middle principle, that an algorithm computing α always exists. Indeed, for any number α, either α is rational or irrational.[14] If α is rational then it is trivially computable, since its binary expansion eventually obeys some predictable periodical law. Otherwise (if it is irrational), we can apply an (easy) algorithm to transform the computable sequences a_n, b_n into its binary expansion (we will return on this fact later). Since the input sequences a_n, b_n are computable, α itself is computable. Nevertheless, only in the second situation can the algorithm which computes a_n, b_n be extended to an algorithm which computes α. In contrast, if α is rational, there is surely some algorithm producing α, but this is completely *independent from* the one producing the sequences a_n, b_n. Turing showed this fact by negating the validity of the following statement (B):

[14] It is well known that this mathematical law does not hold constructively.

> There is a rule whereby, given the rule of formation of the sequences a_n, b_n in (A) we can obtain a D.N.[15] for a machine to compute a. (B) [24, p. 545].

Statement (B) is actually false, and Turing proved its falsehood *by contradiction*. Nevertheless, his proof is not fully satisfactory, since it holds only under the assumption that every rational dyadic number has a unique binary expansion, more precisely the one which is eventually constantly 0. This is a mathematically unjustified restriction. Turing was aware of this, as he observes:

> (...) (B) is false, at least if we adopt the convention that the decimals of numbers of the form $m/2^n$ shall always terminate with zeros (...) [24, p. 545].

Turing's proof is as follows. Take any circle-free machine $\mathcal{N}$ and define a sequence c_n such that $c_n = 1/2$ if $\mathcal{N}$ has not printed the symbol 0 after n steps of computation, otherwise $c_n = 1/2 - 1/2^{m+3}$ where $m \leq n$ is the least step in which $\mathcal{N}$ prints 0. Through the sequence c_n one can define two computable sequences of rational numbers satisfying the conditions required in statement (A):

$$a_n = c_n - 1/2^{n+2}, \qquad b_n = c_n + 1/2^{n+2}.$$

These two sequences converge, as specified by (A), to a rational number $\alpha = \lim_{n\to\infty} c_n$ such that:

- $\alpha = 1/2$ if $\mathcal{N}$ never writes 0;
- $\alpha < 1/2$ otherwise.

In terms of the 'restricted' binary expansion representation used by Turing, the binary expansion of α begins necessarily with 1 in the first case, and with 0 in the second case. Then if we knew how to compute it by the sequences a_n, b_n, we could determine whether the machine $\mathcal{N}$ (never) prints 0 by simply looking at the first digit written in the output. This is in contradiction with the following statement proved in [22]:

> (...) there can be no machine $\mathcal{E}$ which, when supplied with the standard description of an arbitrary machine $\mathcal{M}$, will determine whether $\mathcal{M}$ ever prints a given number (0 say) [22, p. 248].

If we remove the restriction on the allowed binary expansions of the numbers $m/2^n$, we will also re-introduce those expansions which eventually are constantly 1. The proof then no longer holds. In fact for $m \to \infty$, where m is the stage in which $\mathcal{N}$ writes the first 0, the binary expansion of α tends to the infinite sequence $01^{\mathbb{N}}$. This is exactly the binary name of $1/2$ that has now been re-introduced. A machine writing the binary expansion of α could then output initial segments of the sequence $01^{\mathbb{N}}$ for both possible

[15]We recall that the D.N. for a machine is its Gödel number, i.e., the numerical codification of its program.

values of α. Therefore the only way to distinguish the two cases would be that of knowing whether the binary expansion of α is actually $01^{\mathbb{N}}$ or it is different from it, in other words if it contains an occurrence of 0 after the first position. But this was essentially the original problem to solve!

In most applications contemporary computable analysis has replaced the decimal expansion representation by the two following representations of $\mathbb{R}$:

- *the standard representation* $\delta_{\mathbb{R}}$: $p \in \{0,1\}^{\mathbb{N}}$ is a standard name of $x \in \mathbb{R}$ if p encodes an enumeration of all open intervals (x_0, x_1), with $x_0, x_1 \in \mathbb{Q}$, such that $x \in (x_0, x_1)$;
- *the Cauchy representation* $\delta_{\mathbb{R}}^{C}$: $p \in \{0,1\}^{\mathbb{N}}$ is a Cauchy name of $x \in \mathbb{R}$ if p encodes a sequence of rational numbers $x_0, x_1, x_2, \dots$ such that $|x_n, x_m| \leq 2^{-n}$ for $n < m$ and $\lim_{n\to\infty} x_n = x$.

When a Cauchy sequence has an *effective* (computable) modulus of convergence (like that given in the definition of $\delta_{\mathbb{R}}^{C}$), it is said to *converge effectively*.

In terms of the representations $\delta_{\mathbb{R}}$ and $\delta_{\mathbb{R}}^{C}$, the basic operations of real analysis (the sum, the product, the root square, the exponential, the main trigonometric functions ...) are computable. These two representations $\delta_{\mathbb{R}}$ and $\delta_{\mathbb{R}}^{C}$ may seem different at first sight but in reality they are *computably equivalent* ($\delta_{\mathbb{R}} \equiv \delta_{\mathbb{R}}^{C}$), i.e., there is a computable method to transform every $\delta_{\mathbb{R}}$-name of a point $x \in \mathbb{R}$ into a $\delta_{\mathbb{R}}^{C}$-name of the same point, and *vice versa*. In contrast, we can re-formulate the (true!) statement (B) in the language of the modern theory of representations by saying that the Cauchy representation $\delta_{\mathbb{R}}^{C}$ cannot be computably translated into the binary expansion representation, that is to say, there is no uniform method to transform a Cauchy name of a real α into a binary expansion of the same α.[16] Formally, $\delta_{\mathbb{R}}^{C} \not\leq \delta_{b,2}$, where $\delta_{b,2}$ is used to denote the binary expansion representation (see [28]). On the other hand, it is always trivially true that $\delta_{b,2}$ can be translated into (or 'reduced to') $\delta_{\mathbb{R}}^{C}$ ($\delta_{b,2} \leq \delta_{\mathbb{R}}^{C}$): in fact, for every n and $m > n$, the sequences made of the first n, m digits respectively of any binary expansion of the real α are the (finite) binary expansions of rational numbers x_n, x_m with $|x_n - x_m| \leq 2^{-n}$, and $\lim_{n\to\infty} x_n = \alpha$.

[16]'*Uniform*' means that there is an algorithm which can be applied to every input (of the desired kind) to produce a correct output. This does not hold in this case: although it is always possible to transform computably every $\delta_{\mathbb{R}}^{C}$-name of a number α into a binary expansion of the same number, different $\delta_{\mathbb{R}}^{C}$-names require in general the application of two distinct algorithms. For example, when α is rational, the existence of a computable translation is immediately deduced by the fact that the binary expansions of α are computable, hence they can be obtained by some machines without the need of any input. Therefore, in a certain sense, the translation is straightforward in this case. But in another sense, the translation is even impossible, since the information contained in the $\delta_{\mathbb{R}}^{C}$-name given as input is useless to select one of such machines. We know that such machines exist, but we are not able to determine them. In a word, the translation is 'non uniform'. The case in which α is irrational will be discussed below.

The two representations $\delta_{b,2}$ and $\delta_{\mathbb{R}}^{C}$ are computably equivalent for the *irrational numbers*, since in this case the reverse translation (reduction) also holds. In fact for any given rational Cauchy sequence $(x_0, x_1 x_2, \dots)$ *effectively* converging to an irrational number x so that $|x_i - x_j| \leq 2^{-i}$, for $i < j$, one can find for every $k \in \mathbb{N}$ a finite sequence $w_k \in \{0,1\}^{<\mathbb{N}}$ of length at least k such that $y_0^k = \delta_{b,2}(w_k 00^{\mathbb{N}})$, $y_1^k = \delta_{b,2}(w_k 10^{\mathbb{N}})$ and $y_0^k < x < y_1^k$. Then the (unique) binary name of x begins with $w_k 0$, more precisely $p = \sup\{w_i : i \in \mathbb{N}\} \in \{0,1\}^{\mathbb{N}}$ is the binary expansion of x. It is easy to see that this method does not work for rational numbers (and no other method can work, as proved in [28, Theorem 4.1.13, p. 93]).

3.3. The relationship with Brouwer. As we will see in the following, Turing had some knowledge of Brouwer's work, and maybe his proof of the inadequacy of the decimal representation is derived from an argument formulated by Brouwer:

> Sei d_ν die ν-te Ziffer hinter dem Komma der Dezimalbruchentwickelung von π und $m = k_n$, wenn es sich in der fortschreitenden Dezimalbruchentwickelung von π bei d_m zum n-ten Male ereignet, dass der Teil $d_m d_{m+1} \cdots d_{m+9}$ dieser Dezimalbruchentwickelung eine Sequenz 0123456789 bildet. Sei weiter $c_\nu = (-1/2)^{k_1}$, wenn $\nu \geq k_1$, sonst $c_\nu = (-1/2)^\nu$, dann definiert die unendliche Reihe $c_1, c_2, c_3, \dots$ eine reelle Zahl r, für welche weder $r = 0$, noch $r > 0$, noch $r < 0$ gilt [5, p. 3].[17]

In Brouwer's time the question about the occurrence of the sequence 0123456789 in the decimal expansion of π was still an open problem, therefore at that time it was impossible to determine which of $r < 0$, $r = 0$, $r > 0$ was valid.

Brouwer's and Turing's arguments are formally very similar and therefore an influence of the first on the second is plausible, but Turing's argument is formulated in a more general way and is not based, like Brouwer's inference, on a precise temporarily unsolved problem. The particular open question selected by Brouwer was in fact solved in 1997 and we are now able to assert that $r > 0$. Turing's merit is that of having rigorously proved what Brouwer could only perceive: the impossibility of determining whether a certain finite string of symbols will occur in a generic infinite *constructible* sequence or not. Unlike Brouwer, Turing did not simply rely on the plausible truth of this impossibility, but he proved it rigorously, by proving that it is in general impossible to determine whether a circle free machine $\mathcal{N}$ writes a 0 in the output or not.

[17] Let d_ν be the ν-th digit after the dot in the decimal expansion of π and let $m = k_n$ if at the position d_m in the decimal expansion of π it happens for the n-th time that the part $d_m d_{m+1} \cdots d_{m+9}$ of this decimal expansion is the sequence 0123456789. Let further $c_\nu = (-1/2)^{k_1}$, if $\nu \geq k_1$, otherwise $c_\nu = (-1/2)^\nu$, then the infinite sequence $c_1, c_2, c_3, \dots$ defines a real number r, for which neither $r = 0$, nor $r > 0$, nor $r < 0$ holds.

Although Turing did not refer explicitly to any possible influence by Brouwer on the formulation of his argumentation on the unreliability of the binary expansion representation, there is a specific case in which such influence is explicitly mentioned. In [24] Turing finds a possible solution to the failure of statement (B) by introducing a new representation for the real numbers.[18] In doing so, Turing admits openly that that idea is derived from Brouwer, as we will see. We notice here another feature still present in contemporary computable analysis: a constant attention to the methods and results of constructive mathematics (even if computable analysis is a part of classical mathematics).

Turing's new representation associates any sequence

$$\gamma = i\,\underbrace{1\cdots\cdots 1}_{n \text{ times}}0c_1c_2c_3\cdots \tag{2}$$

for $i, c_j \in \{0, 1\}$, with the real number:

$$\alpha = (2i-1)n + \sum_{r=1}^{\infty}(2c_r - 1)(2/3)^r. \tag{3}$$

$(2i-1)n$ is *approximately* the value of the integer part of the real: the exact value can differ by at most 2 units from it. In fact $\sum_{r=1}^{\infty} |(2c_r - 1)(2/3)^r| = 2$. Since, as we have already noticed, for every word $w \in \mathbb{N}^{<\mathbb{N}}$ and for every infinite sequence $p \in \mathbb{N}^{\mathbb{N}}$, $w * p$ is computable if and only if p is computable, the relevant part for computability of α is the subsequence $c_1c_2c_3\cdots$. Hence, we will ignore the initial segment $i1\cdots\cdots 10$, and simply consider the infinite subsequence $c_1c_2c_3\cdots$ as denoting the point

$$\alpha = \sum_{r=1}^{\infty}(2c_r - 1)(2/3)^r. \tag{4}$$

in the interval $[-2, 2]$. This convention also allows us to treat this representation in an analogous way with respect to the binary expansion representation, where, according to equation (1), only fractional quantities are considered in the sum.

We will denote this new representation for real numbers by $\beta_{\mathbb{R}}$. The representation $\beta_{\mathbb{R}}$ is perfectly appropriate for computability theory, being in fact equivalent to the Cauchy representation $\delta_{\mathbb{R}}^{C}$. Nevertheless, it has a more complicated definition. It is strange that Turing did not arrive at the formulation of the easier Cauchy representation, although he had explicitly dealt with *effectively converging* Cauchy sequences already in [22], as we will see in detail in the next section. With the Cauchy representation, the statement (B) would have become a trivial truth, since each of the two rational sequences a_n, b_n is a Cauchy name of the real α. In any case, it

[18]In this context Turing himself employs the term 'representation', which has become standard in computable analysis.

is true that the failure of statement (B) can be avoided by replacing the decimal representation by *any* representation equivalent to $\delta_{\mathbb{R}}^{C}$. Therefore, $\beta_{\mathbb{R}}$ is perfectly suitable for this purpose, as Turing points out:

> This disagreeable situation can be avoided by modifying the manner in which computable numbers are associated with computable sequences, the totality of computable numbers being left unaltered [24, p. 546].

and:

> If the machine which computes γ[19] is regarded as computing also this real number[20] then (B) holds [24, p. 546].

The equivalence between $\beta_{\mathbb{R}}$ and $\delta_{\mathbb{R}}^{C}$ is underlined in [28, Exercise 7.2.7, p. 217] (see also [16, p. 555]). A proof that $\beta_{\mathbb{R}} \leq \delta_{\mathbb{R}}^{C}$ can be obtained in the following way. Take any $\beta_{\mathbb{R}}$-name $c_1c_2c_3\cdots$ of a real number α, so that $\alpha = \sum_{i=1}^{\infty}(2c_i - 1)(2/3)^i$ according to equation (4). Let now $\alpha_n = \sum_{i=1}^{n}(2c_i - 1)(2/3)^i$. Then clearly $|\alpha_{n+1} - \alpha_n| = (2/3)^{n+1}$, so that for any $m > n$

$$|\alpha_m - \alpha_n| \leq \sum_{i=n+1}^{m}\left(\frac{2}{3}\right)^i \leq \sum_{i=n+1}^{\infty}\left(\frac{2}{3}\right)^i = \frac{2^{n+1}}{3^n}.$$

To prove the converse, it is more convenient to consider the *signed digit representation* that we will examine shortly. The definition of this representation is in fact very close to that of $\beta_{\mathbb{R}}$, but simpler.

But how did it happen that Brouwer influenced Turing's work? According to [18], Turing was not very familiar with Brouwer's work while writing [22], but the situation changed suddenly immediately after. Turing met in fact Alonzo Church during his doctorate at Princeton University and Church knew Brouwer personally, having visited him in Amsterdam some years earlier. S. Kleene, who was a student of Church, was also very interested in intuitionism and wrote a book on the subject [12]. Whatever the real connection was, Turing gives credit to Brouwer in the following way:

> This use of overlapping intervals for the definition of real numbers is due originally to Brouwer [24, p. 546].

The core of the representation $\beta_{\mathbb{R}}$ is then the use of overlapping intervals for the codification of real numbers:

> In his famous famous paper (...) on 'Turing machines', A. Turing used the binary representation $\delta_{b,2}$ (...) of the real numbers. Shortly after publication he recognized some difficulty arising from this representation. In his 'Correction' he proposes the representation (...) by overlapping intervals (...) [28, p. 217].

[19] γ is defined according to equation (2).
[20] 'This real number' is α as defined in equation 3.

Turing did not provide any detail about Brouwer's notion of overlapping, but we can find some hint in [3, 4]:[21]

> Wir wählen in der Ebene ein rechtwinkliges Koordinatensystem, und zerlegen die Ebene in Quadrate χ_1 mit der Seitenlänge 1, deren Eckpunkte ganzzahlige Koordinaten besitzen. Jedes der Quadrate χ_1 zerlegen wir in vier kongruente homothetische Teilquadrate χ_2, und definieren, in dieser Weise fortfahrend, Quadrate χ_3, χ_4, usw. Unter einem Quadrat χ verstehen wir ein Quadrat χ_ν mit willkürlichen ν [3, p. 8].[22]
>
> Unter einem *Quadrate* λ_ν verstehen wir das System der Eckpunkte eines die Vereinigung von vier (...) Quadraten $\chi_{\nu+1}$ bildenden Quadrats. (...) Unter einem *Punkte der Ebene* verstehen wir eine unbegrenzt fortgesetze Folge von Quadraten λ, deren jedes im Innengebiete des nächstvorangehenden enthalten ist.
>
> Wenn zwei Punkte P_1 und P_2 die Eigenschaft besitzen, dass in jedem Quadrate von P_1 ein Quadrat von P_2 und in jedem Quadrate von P_2 ein Quadrat von P_1 enthalten ist, so sagen wir, dass P_1 und P_2 *zusammenfallen*. (...) Wenn das Quadrat q_1 des Punktes P_1 und das Quadrat q_2 des Punktes P_2 ausserhalb voneinander liegen, so heissen P_1 und P_2 *örtlich verschieden* [4, p. 3].[23]

Brouwer's construction may obviously be adapted also to spaces of different dimensions, such as the straight line, as Brouwer himself did in [3]:

> (...), definieren wir auf der geraden Linie Intervalle χ in analoger Weise wie oben in der Ebene Quadrate χ, (und) verstehen unter einem intervalle λ_ν die Vereinigung von zwei aneinander grenzenden Intervallen $\chi_{\nu+1}$ (...) [3, p. 12].[24]

Therefore, Brouwer constructs points on the real line by using rational intervals and on the Euclidean space by rational squares. A similar method is used in contemporary computable analysis to define the standard represen-

[21]I thank Prof. Dirk van Dalen for his suggestions on the main biographical coordinates of this subject as they relate to Brouwer's work.

[22]We choose in the plane a system of straight coordinates, and divide the plane into squares χ_1 of side length 1, with vertices of integer coordinates. We divide then each of the squares χ_1 into four congruent squares χ_2 and define, proceeding in the same way, the squares χ_3, χ_4, and so on. By *squares* we mean a square χ_ν with arbitrary ν.

[23]By *square* λ_ν we mean the system originated by the vertices of a square made of a union of four squares $\chi_{\nu+1}$ (...) By *point of the plane* we mean an unlimited convergent sequence of squares λ, such that each square is contained in the interior of the predecessor. When two points P_1 and P_2 have the property that in each square of P_1 is contained a square of P_2 and vice versa, we say that P_1 and P_2 *collapse*. (...) When the square q_1 of the point P_1 and the square q_2 of the point P_2 lie separately, P_1 and P_2 are said to be *locally distinguishable*.

[24](...) we define the intervals χ on the straight line similarly to the squares χ on the plane, (and) conceive the interval λ_ν as the union of two adjacent intervals $\chi_{\nu+1}$ (...).

tations $\delta_{\mathbb{R}}$ and $\delta_{\mathbb{R}^2}$ of $\mathbb{R}$ and $\mathbb{R}^2$, respectively. With respect to these representations, two points P_1, P_2 are equal or different depending on whether they 'collapse' ('zusammenfallen') or they are 'locally distinguishable' ('örtlich verschieden').

Turing's system of overlapping intervals used in $\beta_{\mathbb{R}}$ is obtained by dividing intervals into three congruent parts, and iterating this division operation *ad infinitum*. The construction can be better explained in the following terms. At stage 0, all intervals $[z, z+1]$, for $z \in \mathbb{Z}$, are considered and we call all numbers $z \in \mathbb{Z}$ 'type 0 points'. At stage 1 each interval $[z, z+1]$ is decomposed into three congruent parts by drawing the points $z + (2/3)$ and $(z+1) - (2/3) = z + (1/3)$. We call all the points considered at this stage 'type 1 points' (hence, type 1 points include type 0 points). In this way, the segment $[z, z+1]$ is decomposed into the two overlapping intervals $[z, z+(2/3)]$ and $[z+(1/3), z+1]$. On the other hand, by analogy with Brouwer's construction, each χ_1 interval $[z, z+1]$ is the union of the three adjacent χ_2 intervals $[z, z+(1/3)]$, $[z+(1/3), z+(2/3)]$, $[z+(2/3), z+1]$. At the next stage, we divide each of these χ_2 intervals into three congruent parts in the same way, by drawing the points $i + (2/3)^2$ and $i - (2/3)^2$, for every type 1 point i. All the points considered at this stage are 'type 2 points'. We proceed analogously at the following stages by considering quantities $(2/3)^r$ for increasing r. For example, the interval $[0, 1]$ is firstly decomposed into the two overlapping intervals $[0, 2/3]$ and $[1/3, 1]$ at stage 1, and then into the overlapping interval system $[0, 2/9]$, $[1/9, 3/9]$, $\ldots$, $[7/9, 1]$ at stage 2, and so on. According to (the restriction of) the $\beta_{\mathbb{R}}$ representation (as defined in equation (4)), the information contained in the finite word 00 encodes the type 2 point $-2/3 - 4/9 = -10/9$. Analogously, the words 01, 10 and 11 give us the type 2 points $-2/9$, $2/9$, and $10/9$, respectively. Since $\sum_{r \geq 3}(2c_r - 1)(2/3)^r$, for $c_r \in \{0, 1\}$, can range between $-8/9$ and $8/9$, these four words, as initial segments of $\beta_{\mathbb{R}}$-names, determine the overlapping interval system $[-18/9, -2/9]$, $[-10/9, 6/9]$, $[-6/9, 10/9]$, $[2/9, 18/9]$.

3.4. Admissible representations. The theory of admissible representations is outlined in [13] and systematically investigated in [28].

The use of admissible representations is crucial in computability theory, and it is then not surprising that Turing himself was forced to introduce an example of admissible representation to deal with the reals. Nevertheless, Turing did not arrive at the formulation of a general theory of admissible representations, not even for the case of the real numbers.

When is a representation admissible? Given a second countable T_0-space X, a representation σ_X of X is admissible when every continuous representation of X (with respect to the product topology on $\{0, 1\}^{\mathbb{N}}$ and the given topology on X) is reducible to σ_X via continuous reduction.[25] For the case

[25] We have already met the notion of computable reducibility (or 'translation') between representations. The continuous counterpart is obtained by requiring that the reduction (or

of real numbers this coincides to saying that σ_X is admissible if and only if it continuously equivalent to the Cauchy representation $\delta^C_{\mathbb{R}}$. And in fact $\beta_{\mathbb{R}}$ is admissible, since it is computably equivalent to $\delta^C_{\mathbb{R}}$.

Turing observes that $\beta_{\mathbb{R}}$ is not injective (in contrast with *his* restricted binary expansion representation):

> The uniqueness of representation of real numbers by sequences of figures is now lost, but this is of little theoretical importance (...) [24, p. 546].

A simple topological consideration shows that there is no possible admissible injective representation of $\mathbb{R}$ (see [28, Theorem 4.1.15.2, p. 96]). Therefore the lack of the uniqueness is not only of '*little theoretical importance*', but it is an unavoidable consequence. It is then remarkable how Turing did not miss to point out this fact.

We have already noticed that Turing was very close to the definition of Cauchy representation already in [22]. Moreover Brouwer's notion of overlapping by which Turing defined the representation $\beta_{\mathbb{R}}$ is very close to the standard representation. Nevertheless, although these representations are very simple and intuitive, their explicit introduction would have required a deeper comprehension of the concept of representation than that achieved by Turing. After all, what Turing probably intended to do was simply to *repair* the (binary) decimal representation. Like everyone else he had been used to this representation since childhood and he was only trying to modify it just a little in order to make it work correctly.

The fundamental problem with the (binary) decimal representation is the fact that, according to equation (1), every occurrence of 1 in a decimal expansion increases the value of the corresponding real by a certain amount, whereas there is no way to remove any required quantity from this value. More precisely, if an occurrence of 1 is written at the n-th position of a decimal expansion, the value of the real is increased by 2^{-n}, and this increment is irreversible: the final value obtained by the computation is necessarily equal to or larger than 2^{-n}. This fact produces irreversible errors, as the following informal reasoning can intuitively show. Suppose we are given a Cauchy sequence (effectively) converging to a number α in some little neighborhood of $1/2$ and suppose that this Cauchy sequence must be converted into a binary expansion of α. The members of the sequence are inspected and at some stage the first digit of the output must be written. If 0 is selected, this forces α to be in the interval $[0, 1/2]$, which is obviously a mistake when in reality

'translation') function is simply continuous (and not necessarily computable). Formally, given two representations σ_X and γ_X of a set X, we say that γ_X is *continuously reducible* to σ_X if there is a continuous function $F :\ \subseteq \{0,1\}^{\mathbb{N}} \to \{0,1\}^{\mathbb{N}}$ such that $F(p)$ denotes via σ_X the same element in X denoted by p via γ_X, for every γ_X-name $p \in \{0,1\}^{\mathbb{N}}$. The representations σ_X and γ_X are *continuously equivalent* when they are both continuously reducible to each other.

$\alpha > 1/2$, as later could possibly be discovered. The other strategy would be that of writing 1 as the first digit. This is equivalent to forcing α to be in the interval $[1/2, 1]$, which could be false as well, but if it were possible to remove quantities exactly as they have been added, then the possible mistake could be corrected without any problem. The ability to remove type 2^{-i} quantities for every $i \in \mathbb{N}$, in addition to the ability to add them, would in fact still allow us to obtain any number in $[0, 1]$ even if 1 has been selected as the first output digit. We must then use representations which allow the subtractions of quantities. In this way it is possible to correct the final result in spite of the prohibition of using the instruction 'erase' in the output tape. A simple method to do this is the one suggested by Turing in equation (3): the quantity $(2/3)^r$ is added or subtracted depending on whether $(2c_r - 1)$ is positive or negative. If it is subtracted, then the value of the real *decreases* without any symbol needing to be erased. It can be convenient to add the new symbol -1 to the alphabet $\{0, 1\}$: in this way any given quantity can be added, subtracted or ignored according to whether it is multiplied by $1, -1, 0$, respectively. It is also probably more convenient to consider usual type 2^{-i} quantities. We then obtain the so-called *signed digit representation* δ_{sd}: a real number α is again denoted according to equation (1), but a_i in that equation ranges now on $\{-1, 0, 1\}$ (observe that in this way α lies in the interval $[-1, 1]$; to denote points outside this interval we would need some further symbol or convention to encode integers parts, but we prefer to ignore this complication once again). The core of Turing's representation $\beta_{\mathbb{R}}$ is therefore the same as the signed digit representation, apart from small differences of negligible theoretical importance. Beside the fact that the possibility of ignoring certain quantities through their multiplication by 0 is not assumed by Turing (which may complicate the calculations somewhat), $\beta_{\mathbb{R}}$ is based on the overlapping system obtained by dividing intervals into 3 congruent parts, whereas δ_{sd} corresponds to Brouwer's division of intervals into 2 congruent parts. More precisely, in the construction associated with δ_{sd} type 0 points coincide again with integer numbers, whereas type $n + 1$ points are now obtained respectively by adding and subtracting to each type n point the quantity $2^{-(n+1)}$. Using Brouwer's notation, every $\chi_{\nu+1}$ interval $[i, j]$ (where i, j are adjacent type ν points in our terminology) is decomposed into the two $\chi_{\nu+2}$ intervals $[i, i + (1/2)^{\nu+1}]$, $[j - (1/2)^{\nu+1}, j]$. According to the signed digit representation, the information contained in the word $-1-1$ encodes the type 2 point $= -(1/2) - (1/4) = -3/4$. Analogously, the words -10, -11, $0-1$, 00, 01, $1-1$, 10, 11 give us the points $-1/2$, $-1/4$, $-1/4$, 0, $1/4$, $1/4$, $1/2$, $3/4$. Since $\sum_{i \geq 3} a_i(1/2)^i$, for $a_i \in \{-1, 0, 1\}$, can range between $-1/4$ and $1/4$, these nine words altogether, as initial segments of δ_{sd}-names, determine the overlapping interval system $[-1, -1/2]$, $[-3/4, -1/4]$, $[-1/2, 0]$, $[-1/2, 0]$, $[-1/4, 1/4]$, $[0, 1/2]$, $[0, 1/2]$, $[1/4, 3/4]$, $[1/2, 1]$.

We are now ready to see that $\beta_{\mathbb{R}}$ is actually an admissible representation, i.e., $\beta_{\mathbb{R}} \equiv \delta^C_{\mathbb{R}}$. We have already seen that $\beta_{\mathbb{R}} \leq \delta^C_{\mathbb{R}}$. For the other direction, it is more convenient to replace $\beta_{\mathbb{R}}$ by δ_{sd}, since the reduction strategy is essentially the same but the calculations are easier. We show then that $\delta^C_{\mathbb{R}} \leq \delta_{sd}$. Take any Cauchy sequence of rational numbers effectively converging to a real x. Through this Cauchy sequence, we can obtain closed intervals $\overline{I}_i$ such that $\mathrm{diam}(\overline{I}_i) = 2^{-i}$, $\overline{I}_i \supseteq \overline{I}_{i+1}$ and $\{x\} = \bigcap_{i \in \mathbb{N}} \overline{I}_i$. As noticed above, with the signed digit representation we work without loss of generality in the interval $[-1, 1]$, hence we can assume that $\overline{I}_i \subseteq [0, 1]$ for every i. At stage 0 select a type 1 point x_1 such that $x_1 \in \overline{I}_1$. There can be at most two such points, and at least one exists; furthermore, we can select it such that $x_1 = a_1 \cdot 2^{-1}$ for $a_1 \in \{-1, 0, 1\}$ (hence $x_1 \neq -1, 1$). Then a_1 is the first digit of the δ_{sd}-name $p \in \{0, 1\}^{\mathbb{N}}$ of x. At stage $n \geq 1$ the initial segment $a_1 \cdots a_{n+1}$ of p, for $a_1, \ldots, a_{n+1} \in \{-1, 0, 1\}$, has been obtained and the type $n + 1$ point $x_{n+1} = \sum_{i=1}^{n+1} a_i \cdot 2^{-i}$ is reached. Now take a type $n + 2$ point $x_{n+2} \in \overline{I}_{n+2}$ which is different from -1 and 1 (observe that it always exists). If there is more than one such point, select the one which is closer to x_{n+1}. Then $x_{n+2} = \sum_{i=1}^{n+2} a_i \cdot 2^{-i}$, where $a_{n+2} \in \{-1, 0, 1\}$ (more precisely $a_{n+2} = -1$ if $x_{n+2} < x_{n+1}$, $a_{n+1} = 1$ if $x_{n+2} > x_{n+1}$, and $a_{n+1} = 0$ if $x_{n+2} = x_{n+1}$). The initial segment of p is extended to $a_1 \cdots a_{n+1} a_{n+2}$.

The signed digit representation is very important in complexity theory, therefore it is remarkable that the admissible representation $\beta_{\mathbb{R}}$ defined by Turing is conceptually so similar to δ_{sd}.

§**4. Computable real functions.** Early notions of computable real function are due to Banach and Mazur [15], Markov, Grzegorczyk [9, 10] and Lacombe [14]. In particular, the definitions given by the last two authors are coherent with the TTE approach. What is maybe less known is that a previous notion of computable real function was already introduced in [22]:

> We cannot define general computable functions of a real variable, since there is no general method of describing a real number, but we can define a computable function of a computable variable [22, p. 254].

In [22] Turing's interest was in computable numbers, and for this reason he did not consider possible real variables other than the computable ones. In fact, the use of 'oracles' to store non-computable infinite inputs was not taken into account by Turing in that paper.

The goal of [22] was to rigorously determine the concept of a computable real number rather than that of a computable real function, but Turing also intended to write a whole paper on the latter subject, as he points out at the beginning of [22]:

> Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable (...) The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers [22, p. 230].

Unfortunately, this announced paper was never written. Nevertheless, a computability theory for real functions is already sketched out in [22, §10], where some fundamental results (numbered (i)–(x)) are given. But the fundamental reason for the introduction of the computable real functions in [22] is that of proving that certain numbers '*which would naturally be regarded as computable*' [22, p. 249] are actually computable according to his rigorous concept of computability.[26] This is witnessed by the fact that the computable real functions appear at the beginning of §10 which is significantly called '*Examples of large classes of numbers which are computable*', and they are mainly treated as useful and instrumental concepts:

> It will be useful to begin with definitions of a computable function of an integral variable and of a computable variable [22, p. 254].

Most of the results (i)–(x) concern computable analysis and are in fact fundamental theorems of the subject. Unfortunately, these results are given without proofs, and this prevent us from gaining a deep understanding of Turing's way of working on the topic.

4.1. A problematic definition. Turing defined the computable real functions in the following way [22]. First of all, for every satisfactory n, let γ_n be the (real) number computed by the machine $\mathcal{M}(n)$ encoded by n, and then

[26]The whole §9 in [22] is designed to '*show that the 'computable' numbers include all numbers which would naturally be regarded as computable*' [22, p. 249], where the first occurrence of the word 'computable' means of course 'computable by a Turing machine'. The '*arguments*' given by Turing '*are of three kinds*':

> (a) A direct appeal to intuition.
> (b) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).
> (c) Giving examples of large classes of numbers which are computable.
> [22, p. 249].

Argument (c) is developed in §10, and consequently this paragraph can be viewed as an extension of §9. In argument (b) Turing compares his notion of computability with the Hilbert functional calculus. More precisely, he discusses the existence of machines proving the theorems of the Hilbert functional calculus, in particular those statements asserting that the x-th digit of the decimal expansion of a certain number y is i, for $i \in \{0, 1\}$.

define

$$\alpha_n = \tan(\pi(\gamma_n - 1/2)) \tag{5}$$

unless $\gamma_n = 0$ or $\gamma_n = 1$, in either of which cases $\alpha_n = 0$.

When dealing with the computable real functions, Turing wanted to work with all (computable) real numbers, but the infinite sequences in $\{0,1\}^{\mathbb{N}}$ were seen as the binary expansions of the reals in the interval [0,1]. Turing used then equation (5) to associate every computable sequence in $\{0,1\}^{\mathbb{N}}$ with a computable real number possibly outside [0,1] in such a way that all computable real numbers are covered. In other words, he was implicitly working with another representation of the reals according to which $p \in \{0,1\}^{\mathbb{N}}$ denotes the real number $\tan(\pi(0.p - 1/2))$. In this way, Turing used a method to codify the integer parts of real numbers which is alternative to the one used in the definition of $\beta_{\mathbb{R}}$. Obviously, this particular representation is not essential, as he points out:

> A function α_n may be defined in many other ways so as to run through the computable numbers [22, p. 255].

Consider now any computable function $\phi\colon \mathbb{N} \to \mathbb{N}$

> (...) which can be shown to be such that for any satisfactory argument its value is satisfactory [22, p. 255].

Then

> (...) the function f, defined by
>
> $$f(\alpha_n) = \alpha_{\phi(n)}$$
>
> is a computable function and all computable functions of a computable variable are expressible in this form [22, p. 255].

We now point out that Turing's definition of a computable real function causes some problems of interpretation.

For example, for some $i, j \in \mathbb{N}$ it may happen that $\alpha_i = \alpha_j$ but $\phi(i) \neq \phi(j)$, therefore f is not well-defined. This is a consequence of the fact that the numbers $m/2^n$ have two different binary expansions. Turing could have solved this problem through the same convention adopted in [24] of rejecting one of the two binary expansions, but unfortunately he was postulating that every satisfactory n defines a real number γ_n. That means that every infinite computable sequence must be the binary expansion of a real number.

Moreover, in order to meet Turing's requirements, the function ϕ must satisfy a very strong condition about the satisfactory numbers. Turing requires the existence of a method to show that $\phi(n)$ is satisfactory for every satisfactory n. As a first consequence, f is always a total function, since ϕ is supposed to be defined for every satisfactory n. We have seen already that, according to Turing's approach, all computable functions of integer variable are total, and this is just the same property extended to the set of the computable real numbers. Nevertheless, the required condition on ϕ is very strong, since Turing had proved that it is in general not possible to determine the satisfactory numbers. Therefore he specifies:

> Although it is not possible to find a general process for determining whether a given number is satisfactory, it is often possible to show that certain classes of numbers are satisfactory [22, p. 255].

This observation may suggest that the computable function f could in reality be partial, as its domain could be restricted only to the numbers n for which there is a (uniform) method to prove that such numbers as well as their images $\phi(n)$ are satisfactory. But Turing is not clear on this fact, and in any case this condition also sounds very strong. A reasonable solution would be to simply let f be a partial function whose domain is restricted to the set of the numbers α_n such that $\phi(n)$ is satisfactory (even if there is no method to prove it).

Another fundamental problem is that the Turing-computable real functions f can be discontinuous, which is unacceptable for TTE.[27] Consider the following example. We recall that for any satisfactory n, γ_n is the real number computed by $\mathcal{M}(n)$. Let then $k_0, k_1, k_2, \ldots \in \mathbb{N}$ be satisfactory such that $\gamma_{k_0} = 1/2$ and $\gamma_{k_n} = (1 + 2^n)/2^{(n+1)}$ for every $n > 0$.[28] Consider now the computable function $\phi \colon \mathbb{N} \to \mathbb{N}$ such that $\phi(k_0) = k_1$, $\phi(n) = n$ for $n \neq k_0$. The function ϕ satisfies Turing's requirements, since the k_n's (which are all satisfactory) are mapped to themselves, with the exception of k_0 which is mapped to (the satisfactory) k_1. Moreover, according to equation (5), $\alpha_{k_n} \to 0 = \alpha_{k_0}$ but $f(\alpha_{k_n}) \nrightarrow f(\alpha_{k_0})$, since $f(\alpha_{k_n}) \to 0$ and $f(\alpha_{k_0}) = \tan(\pi/4) \neq 0$. Hence, f is Turing-computable but discontinuous.

4.2. Turing's results. In spite of these essential differences, the results (i)–(x) achieved by Turing are perfectly valid in the TTE framework. We now analyze in detail some of these results that are particularly relevant to our discussion:[29]

> (i) A computable function of a computable function of an integral or computable variable is computable [22, p. 255].

In statement (i) Turing says that computable functions are closed under composition: this is the computational version of the basic topological result concerning continuity.

---

[27] Actually, this is impossible in terms of admissible representations, but not in terms of all representations. Some important cases have been studied in the literature [29]. For example, if the admissible Cauchy representation is replaced by the non-admissible '*naive* Cauchy representation' (according to which a real number is denoted by any possible rational sequence converging to it) many simple discontinuous functions become computable. In this context Turing works with the binary expansion representation, which is not admissible, but the argument given below to prove the existence of computable discontinuous real functions, according to his definition, also applies to the admissible representations of the reals.

[28] It is easy to write programs to compute the numbers γ_{k_n}, and therefore corresponding satisfactory numbers k_n can effectively be found.

[29] Another review of the results (i)–(x) can be found in [18].

> (vi) If α and β are computable and $\alpha < \beta$ and $\phi(\alpha) < 0 < \phi(\beta)$, where $\phi(\alpha)$ is a computable increasing continuous function, then there is a unique computable number γ satisfying $\alpha < \gamma < \beta$ and $\phi(\gamma) = 0$ [22, p. 256].

Statement (vi) is a computable version of the intermediate value theorem. Probably, this is the first explicit example of a computational interpretation of a theorem of classical analysis. Providing computable versions of classical mathematical results is nowadays one of the main goals of computable analysis, and this shows the anticipatory value of Turing's work in that regard.

We point out that in (vi) Turing explicitly requires the continuity of the function, since computability and continuity are not related in his definition, as just mentioned.

> (viii) The limit of a computably convergent sequence is computable [22, p. 256].

In formulating point (viii), Turing came very close to the definition of Cauchy representation. His definition of computable convergence is the following:

> We shall say that a sequence β_n of computable numbers converges computably if there is a computable integral valued function $N(\epsilon)$ of the computable variable ϵ, such that we can show that, if $\epsilon > 0$ and $n > N(\epsilon)$ and $m > N(\epsilon)$ then $|b_n - b_m| < \epsilon$ [22, p. 256].[30]

The expression '*we can show*' suggests that the sequence as a whole, and not only its members, are computable: in fact the sequence β_n probably satisfies some predictable construction law if it is effectively possible to prove that $|\beta_n - \beta_m| < \epsilon$ for all $m, n > N(\epsilon)$. Previously, Turing declared this on page 246:

> It might, for instance, be thought that the limit of a sequence of computable numbers must be computable. This is clearly only true if the sequence of computable numbers is defined by some rule [22, p. 246].

It is well-known that without the assumption of a computable convergence rate the limit of the sequence could be non-computable: a well-known example is the limit $x = \sum_{i \in K} 2^{-i}$ of the computable sequence of the reals $x_s = \sum_{i \in K_s} 2^{-i}$, where K and K_s are respectively the halting set and the partial enumeration of this set at stage s. While the sequence x_s is computable, its limit x is non-computable.

[30] We can assume that $\epsilon \in \mathbb{N}$ rather than $\epsilon \in \mathbb{Q}^+$ by identifying $\mathbb{N}$ and $\mathbb{Q}^+$ via any computable enumeration of the latter set. In this way, only the notion of a computable function of an integer variable is required.

(ix) The limit of a uniformly computably convergent computable sequence of computable functions is a computable function [22, p. 256].

Turing formulates this result for the '*obvious definition*' [22, p. 256] of 'uniformly computable convergence', that can reasonably be expressed in the following way: a (computable) sequence of (computable) real functions $(f_0, f_1, f_2, \dots)$ *converges computably* if there is a computable integral valued function N such that ('*we can show that*') for any computable ϵ and $n, m > N(\epsilon)$:

$$|f_n(x) - f_m(x)| < \epsilon$$

for every $x \in \mathbb{R}$. Point (ix) is the computable version of the classical result according to which every function that is the limit of a uniformly convergent sequence of continuous functions is continuous. It is well-know that without the uniform convergence condition the limit function may be discontinuous, therefore TTE-incomputable, as one can see for the function

$$f(x) = \begin{cases} 0 & \text{if } x \in (-1, 1), \\ 1 & \text{if } x \in (-\infty, -1] \cup [1, \infty). \end{cases}$$

which is the limit of the (computable) sequence of continuous (and computable) functions

$$f_n(x) = \begin{cases} |x^n| & \text{if if } x \in (-1, 1), \\ 1 & \text{if } x \in (-\infty, -1] \cup [1, \infty). \end{cases}$$

Turing's result is then the computable counterpart of the classical theorem relating continuity and uniform convergence, although according to *his* approach the computable real functions are not necessarily continuous.

(x) The sum of a power series whose coefficients form a computable sequence is a computable function in the interior of its intervals of convergence [22, p. 256].

As pointed out already, Turing provided the results (i)–(x) primarily to prove that certain numbers are 'computable' according to his rigorous and specific notion of computability. More precisely, he wanted to show that such notion is *adequate*, since it applies successfully to the *intuitively computable* real numbers. By point (viii) he stated in fact that

$$\pi = 4(1 - 1/3 + 1/5 - \cdots)$$

and

$$e = 1 + 1 + 1/2! + 1/3! + \cdots$$

are computable in his framework. From (vi) and (x) he infered the computability of real algebraic numbers (see [6] for more details) and of the zeroes of real Bessel functions.

§**5. Turing and the realRAM model.** An approach to computable analysis alternative to the one considered so far is the 'realRAM machine' model (or 'BSS-model', from the names of its founders). It was initiated by Leonore Blum, Felipe Cucker, Michael Schub and Steve Smale and it does not accept Turing machines as a foundation for real number algorithms. The realRAM model has instead roots in the methods employed in numerical analysis, more precisely its original motivation was to give a computability model closer to the daily practice of numerical analysts.

If [22, 24] are often seen as a foundation for the TTE approach, L. Blum and F. Cucker have acknowledged how Turing's 1948 paper on the solution of linear equation systems provides a justification for the realRAM paradigm.

Turing wrote a few papers on the computability of specific mathematical functions in which, apparently, he did not work with the abstract definition of a computable real function given in 1936. For example, no mention of it is made in the papers on the computability of the Riemann Zeta-function [25, 27], or in [26]. The very abstract and sophisticated computability theory developed by Turing to solve the logical and theoretical problem of decidability (Hilbertian 'Entscheidungsproblem') does not play an *explicit* role in his work on numerical analysis.

This '*double-face*' approach to computability corresponds to a fundamental dichotomy that has become deeply rooted in mathematics: the development of theoretical computer science and numerical analysis as two radically separated subjects. As Blum, Cucker, Schub and Smale explain:

> There is a substantial conflict between theoretical computer science and numerical analysis. These two subjects with common goal have grown apart. For example, computer scientists are uneasy with calculus, whereas numerical analysis thrives on it. On the other hand numerical analysis see no use for the Turing machine. The conflict has at its roots another age-old conflict, that between the continuous and the discrete. Computer science is oriented by the digital nature of machines and by its discrete foundations given by Turing machines. For numerical analysis, systems of equations and differential equations are central and this discipline depends heavily on the continuous nature of the real numbers (...) A major obstacle to reconciling scientific computation and computer science is the present view of the machine, that is the digital computer. As long as the computer is seen simply as a finite or 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 [2, p. 23].

5.1. Real numbers as atomic objects. According to the realRAM model, real numbers are atomic objects, and the elementary operations on them also are atomic: comparison, sum, difference, multiplication, division over the reals can be executed in a single atomic step. If TTE estimates that the time needed to compute $a + b$, for $a, b \in \mathbb{R}$, is infinite, for the realRAM approach this operation requires only a finite time to be executed, say one single step. Cucker [7] writes:

> Early computers in the 1940s and 1950s represented real numbers as either fixed points or floating points numbers (...) A distinctive feature of all these machine implementations of real numbers arithmetic is that there is no notion of *varying size* of a real number and that, consequently, all real numbers are assumed to have a *unit* size. This assumption somehow involves the fact that a real number is considered as an indivisible entity, much as it is in everyday mathematical thought (...) Another consequence of the assumption that all real numbers have unit size is that arithmetic operations between real numbers have a *fixed* (which we can set to unit) cost [7, pp. 106–107].

This approach may seem at first sight to be very abstract and without actual applications. Nevertheless it defines a natural notion of complexity, according to which the complexity of a computational process depends on the number of elementary operations performed in it. This notion of complexity can be already found in [26]:

> It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then give them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recordings of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. For this purpose, a reciprocation will count as a multiplication. This is purely formal. A division will then count as two multiplications; this seems a little too much, and there may be other anomalies, but on the whole substantial justice should be done [26, p. 288].

Cucker comments on the above quotation by saying:

> One may wonder whether in 1948, with the Turing machine firmly established as a formal model of computation, the consideration of unit cost arithmetic was a sensible choice. It is then perhaps reassuring to know that the quote above was written by Turing himself [7, p. 107].

Speaking of [26], Blum says on the same line:

> Written by Alan Turing while he was preoccupied with solving large systems of equations, this paper is quite well known to numerical analysts but almost unknown by logicians and computer scientists. Its implicit model of computation is more closely related to the former than the latter.
>
> In the first section of the paper, Turing considers the 'measures of work' in a process [1, p. 1028].
>
> From this point of view, it is again natural to start with a model of computation in which real numbers are viewed as entities, and algebraic operations and comparisons, as well as simple accessing, are each counted as a unit of work [1, p. 1028].

[26] contains an overview of the classical resolution methods used to solve linear equation systems and investigates their reliability with respect to *round-off errors*. In particular Turing analyzed the Gaussian elimination process, the Jordan method and the Cholesky method. Turing was interested in the complexity of the computations on the one hand, and in the approximation errors accumulated during their executions on the other hand. The complexity is estimated considering real numbers as '*exact*' objects, but these exact values need then to be substituted by rational approximations when computations are run by electronic devices. This is the unavoidable compromise responsible for the approximation errors investigated in the paper.

The realRAM approach is characterized by the same two-stage procedure: firstly, algorithms working with real numbers as atomic objects are designed and their complexity is evaluated. Secondly, abstract real numbers are substituted by rational approximations and the reliability of this substitution is evaluated:

> The way we solve a computational problem over the reals passes normally through two different stages. First, we design an algorithm which solves the problem. At this stage, we may evaluate its cost and compare it with those of other algorithms solving the same problem. Second, we study how 'stable' the algorithm is, i.e., roughly speaking, how the accuracy of the final result is affected by round-off errors (...) [7, pp. 109–110].
>
> For the first stage of this two-stage procedure (...) the consideration of round-off errors is unnecessary. So, one may perfectly use the exact BSS model. In this way, in addition, the machine models the way we *think* about the algorithm (and not only the way the computer performs it) [7, p. 110].

5.2. Turing's evaluation of complexity. Turing dealt with the solutions of linear equation systems $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ and $\mathbf{b}$ are a square matrix of dimension n and a vector of the same dimension, respectively. He observed that the solution can be found in two ways: either by finding $\mathbf{x}$ directly, or by finding the inverse matrix of $\mathbf{A}$ so to calculate $\mathbf{x}$ as $\mathbf{A}^{-1}\mathbf{b}$. He suggested the use of the first method to solve a single set of equations, since

> (...) the former approach has the advantage of involving less work (about one-third the number of multiplications by almost all methods) [26, p. 288].

But when the solutions of several equation systems with the same matrix $\mathbf{A}$ are needed it is more convenient to search for the inverse and then calculate $\mathbf{A}^{-1}\mathbf{b}$. In this case Turing obtained the following estimate of complexity:

> This involves, in addition, n^2 multiplications and n recordings for each vector, compared with a total of about $(1/3)n^3$ multiplications in an independent solution [26, p. 288].

The result '$(1/3)n^3$ *multiplications in an independent solution*' concerns the Gauss elimination method and it is easily obtained by transforming the equation system $\mathbf{Ax} = \mathbf{b}$ step by step in the following way. At stage 0 one puts $\mathbf{A^0} = \mathbf{A}$ and $\mathbf{b^0} = \mathbf{b}$. At each stage i, for $1 \leq i \leq n-1$, the system $\mathbf{A^i x} = \mathbf{b^i}$ is obtained from $\mathbf{A}^{i-1}\mathbf{x} = \mathbf{b^{i-1}}$ by adding to the j-th equation, for $i < j \leq n$, a multiple of the i-th equation such that the variable x_i disappears in the j-th line. At the end of the process, the i-th equation of $\mathbf{A^{n-1}x} = \mathbf{b^{n-1}}$, for $1 \leq i \leq n$, is of the form $\sum_{j=i}^{n} v_j x_j = c_i$. In particular, the n-th equation is of the form $v_n x_n = c_n$ and this gives the exact value of x_n. Then one moves backwards to evaluate $x_{n-1}, x_{n-2}, \ldots, x_1$ by iterated substitutions. More precisely, by the value of x_n the value of x_{n-1} in the $n-1$-th equation is found; then the values of x_n, x_{n-1} are used to obtain the value of x_{n-2} in the $n-2$-th equation, and so on. Turing estimated that the Gauss solving process can be executed in '$(1/3)n^3 + O(n^2)$ *multiplications and recordings*[31] *of which* $(1/2)n^2 + O(n)$ *recordings involve the vector* $\mathbf{b}$' [26, p. 291]. Probably, these two estimates were obtained by Turing in the following way.

For the latter result $(1/2)n^2 + O(n)$, observe that during the elimination process the vector $\mathbf{b}$ changes $n-1$ times. At stage 0 one records the whole vector $\mathbf{b}$ (n recordings). At every further stage r, for $1 \leq r \leq n-1$, one does not need to record the whole vector $\mathbf{b^r}$. In fact, let $\mathbf{A^{r-1}x} = \mathbf{b^{r-1}}$ coincide with the system of equations $e_1^{r-1}, \ldots, e_n^{r-1}$. To obtain $\mathbf{A^r x} = \mathbf{b^r}$ (which is the system $e_1^r, \ldots, e_n^r$) from $\mathbf{A^{r-1}x} = \mathbf{b^{r-1}}$, one modifies in the latter only $e_{r+1}^{r-1}, \ldots, e_n^{r-1}$, the equations $e_1^{r-1}, \ldots, e_r^{r-1}$ remaining unchanged. Therefore, if the vector $\mathbf{b^{r-1}}$ is $(b_1^{r-1}, b_2^{r-1}, \ldots, b_n^{r-1})$, the vector $\mathbf{b^r}$ will be $(b_1^{r-1}, \ldots, b_r^{r-1}, b_{r+1}^r, \ldots, b_n^r)$, where possibly $b_k^{r-1} \neq b_k^r$ for $r+1 \leq k \leq n$. Consequently, one needs to record only the values $b_{r+1}^r, \ldots, b_n^r$, thus $n-r$

[31]Turing often uses the two expressions 'multiplications' and 'recordings' as interchangeable terms. Apparently he considered multiplication as the fundamental '*atomic*' operation, but in reality the whole process of calculating a number was viewed by him as a single operation that coincides with the last step of recording the final result.

recordings are enough. All together, the number of recordings then amounts to $n + (n-1) + (n-2) + \cdots + 1 = (n^2 + n)/2 = n^2/2 + O(n)$.

For the first result $(1/3)n^3 + O(n^2)$, observe that $\mathbf{A}$ also changes $n-1$ times during the elimination process. At stage 0 one records the whole matrix $\mathbf{A}$ (n^2 recordings). Consider again that $\mathbf{A^r x} = \mathbf{b^r}$ is the system of equations $e_1^r, \ldots, e_n^r$, where we let e_i^r be the equation $a_{i1}^r x_1 + a_{i2}^r x_2 + \cdots + a_{in}^r x_n = b_i^r$ for $1 \le i \le n$. We have already noticed that $e_i^{r-1} = e_i^r$ for $i \le r$. For $i > r$, $a_{ij}^r = 0$ for all $j \le r$. Therefore one does not need to record all values a_{ij}^r, but only those for which $i, j > r$, thus $(n-r)^2$ recordings are enough. Altogether the number of recordings amounts to $n^2 + (n-1)^2 + (n-2)^2 + (n-3)^2 + \cdots + 1 = (2n^3 + 3n^2 + n)/6 = n^3/3 + (3n^2 + n)/6 = n^3/3 + O(n^2)$.

But Turing observes that the method can be simplified. In fact this method corresponds to calculating the coefficients of some appropriate $n \times n$ matrices $\mathbf{J_1}, \ldots, \mathbf{J_{n-1}}$ that give us the matrices $\mathbf{A^r}, \mathbf{b^r}$ by left multiplication. But it is not necessary to calculate all the coefficients of the matrices $\mathbf{J_i}$. It is enough to determine those coefficients of $\mathbf{J_i}$ that are necessary to obtain $\mathbf{J_{i+1}}$. More precisely, $\mathbf{A^r} = \mathbf{J_r J_{r-1}} \cdots \mathbf{J_2 J_1 A}$ and $\mathbf{b^r} = \mathbf{J_r J_{r-1}} \cdots \mathbf{J_2 J_1 b}$, for $1 \le r \le n-1$, where the matrix $\mathbf{J_r}$ is almost equal to the $n \times n$ identity matrix, except for column r, where for $i > r$ the element[32] $(\mathbf{J_r})_{ir}$ is possibly different from 0 and satisfies

$$(\mathbf{J_r})_{ir} = -\mathbf{A}_{ir}^{r-1} / \mathbf{A}_{rr}^{r-1}.$$

Then, for $i > r$,

$$\mathbf{A}_{ij}^{\mathbf{r}} = \mathbf{A}_{ij}^{\mathbf{r-1}} + (\mathbf{J_r})_{ir} \mathbf{A}_{rj}^{\mathbf{r-1}}$$

and for $i \le r$,

$$\mathbf{A}_{ij}^{\mathbf{r}} = \mathbf{A}_{ij}^{\mathbf{r-1}}.$$

Turing estimates that this method is more efficient, as it requires only $n^2 + O(n)$ registrations (it is enough to record the numbers $(\mathbf{J_r})_{ir}$).

Turing then analyzes other methods to solve the equation system $\mathbf{Ax} = \mathbf{b}$ and tries to determine the complexity of those procedures by evaluating in each case the number of recordings which are needed. In this way he estimates the work involved in order to establish which method is preferable. Commenting on the efficiency of the Jordan's method, he observes:

> Although it can be used for the solution of equations, it is not very economical for that purpose. For hand work it has the serious disadvantage that the recording is very heavy (...) It may be the best method for use with electronic computing machinery [26, p. 294].

[32] $\mathbf{B}_{ij}$ is the element in the i-th line and j-th column in $\mathbf{B}$, for any given matrix $\mathbf{B}$.

5.3. Dealing with errors. Turing handled the approximation errors by the notions of '*condition number*' and '*ill-conditioning*'. He introduced the condition number as a coefficient expressing the dependence of the approximation error in the solution $\mathbf{x}$ on the approximation error of $\mathbf{A}$. Let $\Delta\mathbf{x}$ be the approximated solution obtained through the approximated input $\Delta\mathbf{A} = \mathbf{A} - \mathbf{S}$, for a (small) 'error matrix' $\mathbf{S}$. This dependence is formulated by Turing through the equation

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} = \frac{1}{n}\|\mathbf{A}\|\|\mathbf{A}^{-1}\|\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \tag{6}$$

(where $\|\mathbf{A}\| = \sqrt{\sum_{i,j=1}^{n}(\mathbf{A}_{ij})^2}$). Turing calls the quantity $\frac{1}{n}\|\mathbf{A}\|\|\mathbf{A}^{-1}\|$ '*condition number*' of $\mathbf{A}$ but he does not explain how he has deduced equation (6) in which it appears. Commenting on Turing's notion of a condition number, Cucker [7, p. 114] provides a proof of a similar formula, which shows the dependence of $\Delta\mathbf{x}$ on the approximation error relative to $\mathbf{b}$. Cucker obtains an inequality, due to the use of the standard norm $\|A\| = \max_{\|x\|=1}\|Ax\|$. Let $\Delta\mathbf{b}$ be the perturbed approximation of $\mathbf{b}$. Then $\mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{b}$ and

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} = \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{b}\|}\frac{\|\mathbf{b}\|}{\|\mathbf{x}\|} = \frac{\|\mathbf{A}^{-1}\Delta\mathbf{b}\|}{\|\mathbf{b}\|}\frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|\mathbf{A}\|\|\mathbf{A}^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

Here the condition number is $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$. This is a particular case of a general law given by Cucker in the same paper which shows the perturbations of $\mathbf{x}$ given the perturbations both in $\mathbf{A}$ and in $\mathbf{b}$:

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(\mathbf{A})}{1 - \kappa(\mathbf{A})\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}}\left(\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} + \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}\right)$$

(it is assumed here that $\kappa(\mathbf{A})\frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} < 1$).

L. Blum suggests that the notion of a condition number given by Turing provides a link between computer science and scientific computation:

> We now return to the Turing paper on rounding-off errors (...) It is here that the notion of condition (of a linear system) was originally introduced (...) Turing's notion of condition, clearly inspired by Newton's derivative, links both traditions, particularly when considering questions of complexity [1, p. 1032].

The other important notion discussed by Turing is that of 'ill-conditioning'. A set of equations is *ill-conditioned* when a small percentage of error in the input may lead to a large percentage of error in the solution. As an example, Turing considers the system

$$1.4x + 0.9y = 2.7,$$
$$-0.8x + 1.7y = -1.2.$$

If we add one-hundredth of the first equation to the second one, and we replace the first by the result, we obtain the new system

$$\begin{aligned} -0.786x + 1.709y &= -1.173, \\ -0.8x + 1.7y &= -1.2. \end{aligned}$$

Although the two systems are equivalent, we should classify the second system as ill-conditioned, '*or at any rate, as ill-conditioned compared with*' the first one. This is because the coefficients in the first equation of the second system have more significant figures. Working with many significant figures increases the percentage of errors obtained during a computation.

Turing was satisfied by his definition of a condition number as a reliable measure of ill-conditioning. For example:

$$\mathbf{A} = \begin{pmatrix} 0.8 & 0.6 \\ -0.6 & 0.8 \end{pmatrix}$$

has condition number 1, whereas the matrix

$$\mathbf{B} = \begin{pmatrix} 0.008 & 0.006 \\ -0.6 & 0.8 \end{pmatrix}$$

has condition number 50.005. Coherently, '*the small values of the first row of*' matrix **B** '*will prejudice the accuracy obtainable, because of the number of significant figures available*' [26, p. 298].

Turing concluded his investigation on the round-off errors in the Gauss elimination method by saying that errors are normally quite moderate. Only with particularly unlucky examples of ill-conditioned matrices can one obtain large errors. The growth of the error is not exponential, and Turing also arrived at similar conclusions for the other methods that he discusses.

REFERENCES

[1] Leonore Blum, *Computing over the reals: where Turing meets Newton*, ***Notices of the American Mathematical Society***, vol. 51 (2004), no. 9, pp. 1024–1034.

[2] Leonore Blum, Felipe Cucker, Michael Schub, and Stephen Smale, ***Complexity and real computation***, Springer, New York, 1998.

[3] Luitzen E. J. Brouwer, *Begründung der Mengenlehre unabhängig vom logischen Satz vom ausgeschlossenen Dritten. Erster Teil: allgemeine Mengenlehre*, ***Verhandelingen der Koninklijke Nederlandse Akademie van Wetenschappen***, vol. 1 sectie 12 (1918), no. 5, pp. 3–43.

[4] ———, *Begründung der Mengenlehre unabhängig vom logischen Satz vom ausgeschlossenen Dritten. Zweiter Teil: Theorie der Punktmengen*, ***Verhandelingen der Koninklijke Nederlandse Akademie van Wetenschappen***, vol. 1 sectie 12 (1919), no. 7, pp. 3–33.

[5] ———, *Über die Bedeutung des Satzes vom ausgeschlossenen Dritten in der Mathematik, insbesondere in der Funktiontheorie*, ***Journal für die reine und angewandte Mathematik***, vol. 154 (1924), pp. 1–7.

[6] Gregory Chaitin, ***Meta math! The quest for Omega***, Vintage Books, 2006.

[7] Felipe Cucker, *Real computations with fake numbers*, ***Journal of Complexity***, vol. 18 (2002), pp. 104–134.

[8] Martin Davis, ***Computability and unsolvability***, McGraw–Hill, 1958.

[9] Andrzej Grzegorczick, *Computable functionals*, ***Fundamenta Mathematicae***, vol. 42 (1955), pp. 168–202.

[10] ———, *On the definitions of computable real continuous functions*, ***Fundamenta Mathematicae***, vol. 44 (1957), pp. 61–71.

[11] Stephen C. Kleene, ***Introduction to metamathematics***, North–Holland Publishing Company, Amsterdam, 1952.

[12] Stephen C. Kleene and Richard E. Vesley, ***Intuitionistic mathematics especially in relation to recursive functions***, North–Holland Publishing Company, Amsterdam, 1965.

[13] Christoph Kreitz and K. Weihrauch, *Theory of representations*, ***Theoretical Computer Science***, vol. 38 (1985), pp. 35–53.

[14] Daniel Lacombe, *Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles III*, ***Comptes Rendus Académie des Sciences Paris***, vol. 241 (1955), pp. 151–153.

[15] Stanislav Mazur, ***Computational analysis***, Razprawy Matematyczne, Warsaw, 1963.

[16] Joseph S. Miller, *Degrees of unsolvability of continuous functions*, ***The Journal of Symbolic Logic***, vol. 69 (2004), no. 2, pp. 555–584.

[17] Piergiorgio Odifreddi, ***Classical recursion theory***, North–Holland Publishing Company, Amsterdam, 1989.

[18] Charles Petzold, ***The annotated Turing***, Wiley Publishing, Indianapolis, 2008.

[19] Emil L. Post, *Recursive unsolvability of a problem of Thue*, ***The Journal of Symbolic Logic***, vol. 12 (1947), pp. 503–513.

[20] Hartley Rogers, ***Theory of recursive functions and effective computability***, McGraw–Hill, New York, 1967.

[21] Robert I. Soare, ***Recursively enumerable sets and degrees***, Springer, Berlin, 1987.

[22] Alan M. Turing, *On computable numbers, with an application to the 'Entscheidungsproblem'*, ***Proceedings of the London Mathematical Society***, vol. 42 (1936), no. 2, pp. 230–265.

[23] ———, *Computability and λ–definability*, ***The Journal of Symbolic Logic***, vol. 2 (1937), no. 4, pp. 153–163.

[24] ———, *On computable numbers, with an application to the 'Entscheidungsproblem'. A correction*, ***Proceedings of the London Mathematical Society***, vol. 43 (1937), no. 2, pp. 544–546.

[25] ———, *A method for the calculation of the Zeta–function*, ***Proceedings of the London Mathematical Society***, vol. 48 (1943), no. 2, pp. 180–197.

[26] ———, *Rounding–off errors in matrix processes*, ***The Quarterly Journal of Mechanics and Applied Mathematics***, vol. 1 (1948), no. 1, pp. 287–308.

[27] ———, *Some calculations of the Riemann Zeta–function*, ***Proceedings of the London Mathematical Society***, vol. 3 (1953), no. 3, pp. 99–117.

[28] Klaus Weihrauch, ***Computable analysis***, Springer, Berlin, 2000.

[29] Martin Ziegler, *Revising type–2 computation and degrees of discontinuity*, ***Electronic Notes in Theoretical Computer Science***, vol. 167 (2007), pp. 255–274.

DIPARTIMENTO DI FILOSOFIA
UNIVERSITÀ DI BOLOGNA
VIA ZAMBONI 38
40126 BOLOGNA, ITALY
E-mail: guido.gherardi@unibo.it