EVEN TURING MACHINES CAN COMPUTE UNCOMPUTABLE FUNCTIONS

B. Jack Copeland

ABSTRACT

Accelerated Turing machines are Turing machines that perform tasks commonly regarded as impossible, such as computing the halting function. The existence of these notional machines has obvious implications concerning the theoretical limits of computability.

1. Introduction

Neither Turing nor Post, in their descriptions of the devices we now call Turing machines, made much mention of time (Turing 1936, Post 1936).[1] They listed the primitive operations that their devices perform - read a square of the tape, write a single symbol on a square of the tape (first deleting any symbol already present), move one square to the right, and so forth - but they made no mention of the *duration* of each primitive operation. The crucial concept is that of whether or not the machine halts after a finite *number* of operations. Temporal considerations are not relevant to the functioning of the devices as described, nor - so we are clearly supposed to believe - to the soundness of the proofs that Turing gave concerning them. Things are very different in the case of the boolean networks of McCulloch and Pitts (1943) and Turing (1948), where the duration of each primitive operation is a critical factor. Turing's boolean networks were synchronised by a central clock and each primitive operation took one 'moment' of clock time (1948: 10).[2] When working with Turing machines it is no doubt intuitive to imagine each primitive operation to be similarly fixed in duration, but this is no part of the

---

[1]My thanks to the audience of a talk I gave at King's College London in October 1995, whose vigorous questioning brought me to think seriously about (what I now call) accelerated Turing machines, and in particular to Richard Sorabji and Mark Sainsbury. Thanks also to Diane Proudfoot, Peter Farleigh, Philip Catton, Chris Bullsmith and Neil Tennant for valuable comments and discussion.

[2]Turing's boolean networks and his connectionist project involving them are described in Copeland and Proudfoot 1996.

original conception. No conditions were placed on the temporal patterning of the sequences of primitive operations.

Bertrand Russell, Ralph Blake and Hermann Weyl independently described one extreme form of temporal patterning. Weyl considered a machine (of unspecified architecture) that is capable of completing

> an infinite sequence of distinct acts of decision within a finite time; say, by supplying the first result after 1/2 minute, the second after another 1/4 minute, the third 1/8 minute later than the second, etc. In this way it would be possible ... to achieve a traversal of all natural numbers and thereby a sure yes-or-no decision regarding any existential question about natural numbers. (Weyl 1927: 34; English translation from Weyl 1949: 42.)

It seems this temporal patterning was first described by Russell, in a lecture given in Boston in 1914. In a discussion of Zeno's paradox of the race-course Russell said 'If half the course takes half a minute, and the next quarter takes a quarter of a minute, and so on, the whole course will take a minute' (Russell 1915: 172-3). Later, in a discussion of a paper by Alice Ambrose (Ambrose 1935), he wrote:

> Miss Ambrose says it is *logically* impossible [for a man] to run through the whole expansion of π. I should have said it was *medically* impossible. ... The opinion that the phrase 'after an infinite number of operations' is self-contradictory, seems scarcely correct. Might not a man's skill increase so fast that he performed each operation in half the time required for its predecessor? In that case, the whole infinite series would take only twice as long as the first operation. (1936: 143-4.)

Blake, too, argued for the possibility of completing an infinite series of acts in a finite time:

> A process is perfectly conceivable, for example, such that at each stage of the process the addition of the next increment in the series 1/2, 1/4, 1/8, etc., should take just half as long as the addition of the previous increment. But ... then the addition of all the increments each to each shows no sign whatever of taking forever. On the contrary, it becomes evident that it will all be accomplished within a certain definitely limited duration. ... If, e.g., the first act ... takes 1/2 second, the next 1/4 second, etc., the [process] will ... be accomplished in precisely one second. (1926: 650-51.)

Imposing the Russell-Blake-Weyl temporal patterning upon a Turing machine produces an *accelerated* Turing machine. These are Turing machines that perform the second primitive operation called for by the program in half the time taken to perform the first, the third in half the time taken to perform the second, and so on.[3] Let the time taken to

---

[3]Stewart (1991: 664-5) gives a cameo discussion of such machines. Related to accelerated Turing machines are the anti de Sitter machines of Hogarth (1994) and the Zeus machines of Boolos and Jeffrey (1980: 14-15). The term 'anti de Sitter machine' is from Copeland and Sylvan 1997. Hogarth's own term for his machines, 'non-Turing computers', is inappropriate in view of the discussion given in the present paper. Copeland and Sylvan 1997 discusses the relationship between Hogarth's machines and Turing's O-machines (the latter are described in section 3 below). Boolos and Jeffrey envisage Zeus being able to act so as to exhibit the Russell-Blake-Weyl temporal patterning (1980: 14). By an extension of terminology (which Boolos and Jeffrey do not make) a Zeus *machine* is any machine exhibiting the Russell-Blake-Weyl temporal patterning. All accelerated Turing machines are Zeus machines, but not vice versa. For example, an O-machine that exhibits

perform the first operation called for by the program be one 'moment'. Since

$$1/2 + 1/4 + 1/8 + ... + 1/2^n + 1/2^{n+1} + ...$$

is less than 1, an accelerated Turing machine can perform unboundedly many primitive operations before two moments of operating time have elapsed.

Since accelerated Turing machines are Turing machines, the restricted quantifiers 'all Turing machines', 'some Turing machines' and 'no Turing machines' have accelerated Turing machines among their range.

2. Computing the halting function

Every Turing machine has a program of instructions 'hard wired' into its head (the read/write device that has the tape passing through it). By using some suitable system of coding conventions, the instructions of any given Turing machine can be represented by means of a single (large) binary number. The number is obtained by first coding each individual instruction into a sequence of binary digits n digits long (for a fixed n) and then running all these together into one long string, with the code for the first instruction at the far left and for the last instruction at the far right. Call this number the machine's *program number*. (Such systems of coding conventions are known as assembly languages.)

Before a Turing machine is set in motion some sequence of binary digits is inscribed on its tape.[4] This is the input, the data upon which the machine is to operate coded in binary form. Of course, this sequence of binary digits, as well as being a representation of the data (which may be non-numeric: sentences of English, for example), is also a representation of a number. Call this number the machine's *data number*. One may speak of the machine being set in motion *bearing* such-and-such a data number.

The famous 'halting function' H is a function taking a pair of integers as arguments and having either 0 or 1 as its value (Turing 1936). It may be defined as follows, for any pair of integers x and y: H(x,y)=1 if and only if x is the program number of a Turing machine that eventually halts if set in motion bearing data number y; H(x,y)=0 otherwise. Notice that if the integer x is *not* a program number of a Turing machine then H(x,y)=0 for every choice of y; and if x *is* a program number, say of Turing machine t, then H(x,y)=0 if and only if t fails to halt when set in motion bearing y.

A machine that could compute the values of the halting function would be able to inform us, concerning any given Turing machine, whether or not that machine would halt when set in motion bearing any given data number. Such a machine, however, could not itself be a Turing machine: Turing was able to prove that no Turing machine can compute the values

---

the Russell-Blake-Weyl patterning - such as the machine O_H of section 3 - is a Zeus machine but is not a Turing machine.

[4]We wish to reason concerning the set of all Turing machines, but without loss of generality we may consider in its stead the set of Turing machines employing the binary alphabet, mark (1) and blank (0).

of the halting function for all integers x and y (1936, section 11). This result is commonly known as the halting theorem.

Yet it seems that an accelerated Turing machine would be able to so inform us, in contradiction to Turing's result.

The accelerated Turing machine in question is an accelerated *universal* Turing machine. A universal Turing machine can simulate the behaviour of any other Turing machine. Let t be the machine that is to be simulated. The universal machine is set in motion with t's program number followed by t's data number inscribed on its tape (which is to say, is set in motion bearing the data number formed by writing out the digits of t's program number followed by the digits of t's data number).[5] Thereafter the universal machine will perform every operation that t will, in the same order as t (although interspersed with sequences of operations not performed by t), and will halt just in case t does. (This idea of programming a machine by entering symbolically encoded instructions into its rewritable memory, and the associated concept of universality, were Turing's greatest contributions to the development of the digital computer.[6])

To produce a machine that can compute the values of the halting function for all integers x and y one simply equips an accelerated universal Turing machine with a signalling device - a hooter, say - such that the hooter blows when and only when the machine halts.[7] To compute the

---

[5]Also included is some code to serve as punctuation between the two numbers.

[6]I defend this historical claim elsewhere (Copeland 1998).

[7] Turing and his colleagues enjoyed the possibilities afforded by the hooter of the Manchester Mark I computer (the world's first fully

value of the halting function for a given machine t, one writes out the digits of t's program number p followed by the digits of t's data number d on the accelerated universal machine's tape and sets it in motion. If t does halt then it does so having performed n primitive operations (where n is some integer), and no matter how large n is, the accelerated universal machine copies t's behaviour, blow by blow, up to and including the act of halting, within two moments of running time. That is to say, if the hooter blows within two moments of the start of the simulation then H(p,d)=1, and if it remains quiet then H(p,d)=0. Given any Turing machine bearing any data number, the accelerated universal machine can inform us whether or not that machine halts.[8] (This is so even where the given machine is the accelerated universal machine itself; it informs us whether or not it itself halts when set in motion bearing any data number by

---

electronic stored-program digital computer). Turing's programming manual for the Manchester machine describes the hooter as producing 'a steady note, rich in harmonics' (1951: 24). The machine's order-code contained an instruction to send a single pulse to the hooter; a train of pulses, timed correctly, would produce a note. Turing displays a loop of two instructions producing middle C, and a loop of three instructions 'which gives a slightly louder hoot a fifth lower in frequency' (ibid.). The first program of any significant size to run on the machine - written by Christopher Strachey at Turing's behest - brought its activity to a close by playing the National Anthem on the hooter.

[8]The accelerated universal machine is coded to enter an infinite loop when x is not the program number of a Turing machine, thus correctly returning 0 as the value of H.

signalling to us if it does so (if there is to be a hoot it must come within the first two moments of running time).) So we appear to have an example of a Turing machine that can compute a function that no Turing machine can compute.

It would be easy to conclude from this apparent contradiction that an accelerated universal Turing machine is a logical impossibility. To do so would not be correct, however. There is in fact a mistake in the reasoning leading to the contradiction.

Let me call the machine just described ℍ (for 'halting-function machine'). ℍ is not in fact a Turing machine, for a reason that has nothing to do with its accelerated nature. ℍ is a system consisting of a Turing machine plus additional equipment (most conspicuously the hooter), and so is *more* than a Turing machine. In particular, ℍ's program of instructions is not a Turing machine program. Among ℍ's primitive operations is the act of blowing the hooter and ℍ's program of instructions includes provision for this operation to be performed if certain conditions are met, whereas no Turing machine program contains any such instruction. Since ℍ is not a Turing machine, the fact that this particular machine can compute the halting function in no way contradicts the proposition that no Turing machine can do so. (A close cousin of ℍ that *is* a Turing machine is discussed in section 5.)

The knowledge that ℍ is not a Turing machine is not quite sufficient to produce conviction that there is no paradox here. For might not the considerations used in the proof that a Turing machine cannot compute the halting function also apply to ℍ, enabling one to infer that ℍ both does and does not compute the halting function? A review of the crucial ideas underlying the proof of the halting theorem will show that this is not the case.

The proof takes the form of a reductio ad absurdum. Assume that there *is* a Turing machine that computes the halting function; call it h. That is to say, for any two integers m and n, if h is set in motion bearing the data number produced by writing out the (binary) digits of m followed by the (binary) digits of n, then h will halt with 1 under its head if and only if H(m,n)=1, and h will halt with 0 under its head if and only if H(m,n)=0. The argument proceeds by introducing a further machine, h_2, which can be derived from h by two simple modifications. Provably, if h exists then h_2 exists. But it is easily shown that h_2 cannot exist, for if it does then a certain contradiction is true. Therefore h cannot exist either.

The first modification of h produces a machine h_1 which, when set in motion bearing the same data number as h, halts with 0 under its head if and only if h does so, but which never halts with 1 under its head. The modification consists of adding some instructions to h's program in order to make the head shuffle endlessly back and forth between some pair of adjacent squares of the tape in the case where it would otherwise have halted with 1 beneath it.[9]

In order to summarise the key facts about h and h_1 it is useful to introduce some notation. I will use 'm↓0n' ['m↓1n'] to mean 'machine m halts with 0 [1] under its head when set in motion bearing data number n' (for short, 'm halts on 0 [1] given n'); 'm↓n' to mean 'm halts when set in motion bearing n'; and 'i^j' (pronounced 'i concatenated with j') to indicate the number formed by first writing out the digits of i and then the digits of j (placing the first digit of j to the right of the last digit of i).[10] '↔' is the

---

[9]Further details of this modification may be found in many textbooks, for example Minsky 1967: 149 and Boolos and Jeffrey 1980: 17.

[10]See note **five**.

biconditional, '$\vee$' is exclusive disjunction, '$\neg$' is negation; x and y are integers.

(1) for every x, y: $h{\downarrow}1x\hat{}y \leftrightarrow H(x,y)=1$

(2) for every x, y: $h{\downarrow}0x\hat{}y \leftrightarrow H(x,y)=0$

(3) for every x: $h{\downarrow}x \leftrightarrow h{\downarrow}1x \vee h{\downarrow}0x$

(4) for every x: $h_1{\downarrow}x \leftrightarrow h{\downarrow}0x$.

(1) and (2) are true in virtue of the hypothesis that h computes the halting function, (3) is true in virtue of the binary nature of h, and (4) in virtue of the way h_1 is constructed from h.

Next h_1 is modified to produce h_2. h_2 is identical to h_1 except that h_2 first writes out a copy of its data number, beginning on the square immediately following the last digit of the initial occurrence of the data number. Thereafter h_2 behaves exactly as h_1. So if h_2 is set in motion bearing the data number m then once the copying phase is completed, h_2 will behave exactly as h_1 behaves when set in motion bearing the data number $m\hat{}m$.

So

(5) for every x: $h_2{\downarrow}x \leftrightarrow h_1{\downarrow}x\hat{}x$.

Let us investigate the effect of setting h_2 in motion bearing its own program number, p, as data number.

(6) $h_2{\downarrow}p \leftrightarrow h_1{\downarrow}p\hat{}p$ (from 5)

(7) $h_1{\downarrow}p\hat{}p \leftrightarrow h{\downarrow}0p\hat{}p$ (from 4)

(8) $h{\downarrow}0p\hat{}p \leftrightarrow H(p,p)=0$ (from 2)

So (9) $h_2{\downarrow}p \leftrightarrow H(p,p)=0$ (from 6-8, by the transitivity of implication).

Now for the other arm of the contradiction. Since p is the program number of h_2 and h_2 is a Turing machine, the definition of the halting function gives

(10) $H(p,p)=0 \leftrightarrow \neg h_2{\downarrow}p$.

Combining (9) and (10) produces

(11) $h_2{\downarrow}p \leftrightarrow \neg h_2{\downarrow}p$.

This is a contradiction ($A \leftrightarrow \neg A$ and $A \& \neg A$ being interderivable in the propositional calculus). So the assumption leading to the contradiction - that some Turing machine can compute the halting function - is false.

Precisely where does this same train of reasoning fail if applied to $\mathbb{H}$? $\mathbb{H}$ does its work of simulating a given Turing machine t by inspecting t's program number and obeying each instruction written there in the order in which they are written, interpolating additional operations into the sequence whenever necessary. One operation that certainly must be interpolated is that of blowing the hooter. The table of instructions that characterises $\mathbb{H}$'s behaviour contains an entry telling $\mathbb{H}$ to perform this operation immediately before it halts (which it will if and only if t does). Thus if $\mathbb{H}$ is to be assigned a program number, the coding scheme that leads from tables of instructions to binary program numbers must be extended with a word designating this operation, and so $\mathbb{H}$'s program number, q say, will be distinctively different from the program number of any Turing machine. It is the fact that $\mathbb{H}$'s program number is not a program number of a Turing machine that stalls the foregoing train of reasoning as applied to $\mathbb{H}$.

In fact, $\mathbb{H}$ does not halt when set in motion bearing $q^\frown q$. This is because it is straightforwardly the case - from the definition of the halting function given earlier - that $H(q,q)=0$. Since $\mathbb{H}$ computes the halting function (indicating the value 1 by hooting and the value 0 by not hooting):

(12) $H(q,q)=0 \leftrightarrow \neg \mathbb{H}{\downarrow}q^\frown q$.

So we can infer, perfectly correctly, that $\neg \mathbb{H}{\downarrow}q^\frown q$.

The halting behaviour of h and its derivatives bears messages, spelt out in (1) and (2), concerning the halting behaviour of the Turing machine whose program number appears in the data number supplied to h (or derivative). Thus the means of tying the self-referential knot and producing the contradiction. The hooting/non-hooting behaviour of $\mathbb{H}$ - and of any suitable derivative of $\mathbb{H}$ - also bears such messages about Turing machines, but since $\mathbb{H}$ and its derivatives are not Turing machines the messages cannot come to be about the halting behaviour of these machines themselves. The absence of a hoot within the prescribed period when, for some integer x, $\mathbb{H}$ is fed x^x, bears a disjunctive message: either x is not the program number of a Turing machine or the Turing machine with this number does not halt when set in motion bearing x. No such disjunction can be pressed into making a statement about the halting behaviour of $\mathbb{H}$ itself nor of any derivative of it that is not a Turing machine.

3. Turing's O-machines

Let me render these considerations concrete by investigating a machine $\mathbb{H}_2$ that bears the same relationship to $\mathbb{H}$ as h_2 bears to h.

Turing introduced the concept of an O-machine in section 4 of his PhD thesis (1938). (The thesis, which was written at Princeton under the supervision of Church, was subsequently published as Turing (1939)). An O-machine is a Turing machine equipped with an additional device - a black box - that, when presented with arguments of some non Turing-machine-computable function, returns the corresponding values of this function. For example, the black box may respond to an input of a pair of integers, x and y, with the corresponding value of the halting function,

H(x,y), for every x and y. Turing called such black boxes 'oracles'. He remarked that an oracle works by 'unspecified means', saying that we need 'not go any further into the nature of [an] oracle' (1939: 173). A 'call to the oracle' is a primitive operation of an O-machine.

One way of specifying a mechanism by means of which an oracle can do its work is in terms of $\mathbb{H}$. $O_{\mathbb{H}}$ is an O-machine of which $\mathbb{H}$ itself serves as oracle. $O_{\mathbb{H}}$ consists of $\mathbb{H}$ operating in conjunction with a universal Turing machine T equipped with a clock and with resources for delivering a data number to $\mathbb{H}$'s tape, for setting $\mathbb{H}$ in motion, and for detecting and recording the presence or absence of a signal within two moments of $\mathbb{H}$'s work commencing. $O_{\mathbb{H}}$ will be described as being set in motion bearing data number x just in case T is so set in motion, and to halt (with 1 [0] under its head) just in case T halts (with 1 [0] under its head). $O_{\mathbb{H}}$ is able to compute numerous non Turing-machine-computable functions, for a wide assortment of such functions is *reducible* to the halting function, in the sense that any machine that is able to compute the halting function is able to compute them also.

To make matters specific, let $O_{\mathbb{H}}$ be arranged so that its halting behaviour when set in motion bearing a data number $x^\frown y$ (which is to say, the halting behaviour of its component machine T when set in motion bearing this data number) is as follows: $O_{\mathbb{H}}$ halts with 1 under its head just in case the oracle $\mathbb{H}$ delivers a signal when fed $x^\frown y$ and $O_{\mathbb{H}}$ halts with 0 under its head just in case the oracle delivers no signal when fed $x^\frown y$. So:

(1') for every x, y: $O_{\mathbb{H}}\downarrow 1x^\frown y \leftrightarrow H(x,y)=1$

(2') for every x, y: $O_{\mathbb{H}}\downarrow 0x^\frown y \leftrightarrow H(x,y)=0$.

$\mathbb{H}_1$ is obtained from $O_{\mathbb{H}}$ by exactly the moves that yield h_1 from h. So

(4') for every x: $\mathbb{H}_1\downarrow x \leftrightarrow O_{\mathbb{H}}\downarrow 0x$.

$\mathbb{H}_1$ is modified to produce $\mathbb{H}_2$ in just the way that h_1 is modified to produce h_2. So

(5') for every x: $\mathbb{H}_2{\downarrow}x \leftrightarrow \mathbb{H}_1{\downarrow}x\hat{}x$.

As with h_2, let us consider the effect of setting $\mathbb{H}_2$ in motion bearing its own program number, r, as data number. We obtain

(6') $\mathbb{H}_2{\downarrow}r \leftrightarrow \mathbb{H}_1{\downarrow}r\hat{}r$ (from 5')

(7') $\mathbb{H}_1{\downarrow}r\hat{}r \leftrightarrow O_{\mathbb{H}}{\downarrow}0r\hat{}r$ (from 4')

(8') $O_{\mathbb{H}}{\downarrow}0r\hat{}r \leftrightarrow H(r,r)=0$ (from 2')

(9') $\mathbb{H}_2{\downarrow}r \leftrightarrow H(r,r)=0$ (6'-8', transitivity).

Since r is not the program number of a Turing machine, H(r,r) is in fact 0. Thus we can conclude that $\mathbb{H}_2$ does halt if set in motion bearing its own program number as data number.

There is no hope of also showing that $\mathbb{H}_2$ does *not* halt if set in motion bearing its own program number as data number. The inference to (10) requires the knowledge that p (h_2's program number) is the program number of a Turing machine, for it is only if this is so that H(p,p) being 0 entails that the Turing machine so numbered does not halt when fed p. It is precisely this inference that is blocked in the case of $\mathbb{H}_2$. h_2's halting when fed p bears a message about the behaviour of the Turing machine whose program number is p, namely that it does not halt when fed p, but $\mathbb{H}_2$'s halting when fed r bears no message about the halting behaviour of the machine whose program number is r.

O-machines form an infinite hierarchy. At the bottom are the first-order O-machines: the O-machines which, like $O_{\mathbb{H}}$, can compute all Turing-machine-computable functions and all functions reducible to the halting function H(x,y) but no other functions. The halting function for first-order O-machines, $H_1(x,y)$, is defined as follows (for all integers x and y): $H_1(x,y)=1$ if and only if x is the program number of a first-order O-machine

that halts if set in motion bearing data number y; $H_1(x,y)=0$ otherwise. That no first-order O-machine can compute the halting function for first-order O-machines is shown by much the same argument as that which establishes that no Turing machine can compute the halting function for Turing machines. It is the inability of $\mathbb{H}$ and $O_{\mathbb{H}}$ to compute the halting function for first-order O-machines - while nevertheless being able to compute the halting function for Turing machines - that preserves them from inconsistency.

4. Some myths about computability

That Turing was the first to consider computing devices which compute more than the machines of his 1936 paper is less widely known than it should be. Indeed, the so-called Church-Turing thesis maintains that Turing machines form a *maximal class* of idealised computing devices, in the sense that the functions computable by Turing machines allegedly exhaust the functions computable by idealised computing devices of any form (which is to say, exhaust the functions that are computable both in practice and in principle). Thus, for example, Minsky maintains that the operations of a Turing machine 'give rise to the full range of possible computations' (1967: 112), and Dennett that 'anything computable is Turing-machine computable' (1978: 83). Fodor restricts the claim to computations on discrete symbols: 'Although the elementary operations of the Turing machine are restricted, iterations of the operations enable the machine to carry out any well-defined computation on discrete symbols (1981: 130; see also 1983: 38-9). McArthur writes: 'The limits of Turing machines, according to the Church-Turing thesis, also describe the theoretical limits of all computers' (1991: 401). Since there can

presumably be no question but that O_H *computes* (and, moreover, on discrete symbols), O_H forms a clear counterexample to this thesis. Despite a seemingly widespread belief to the contrary, neither Church nor Turing advanced this thesis. (The relevant texts by Church and, particularly, Turing are examined in Copeland 1996.)

The erroneous thought that, in his paper of 1936, Turing showed something fundamental concerning the limits of what can be computed by machine enjoys a wide currency. Thus, for example, the 'Oxford Companion to the Mind' states: 'Turing showed that his very simple machine ... can specify the steps required for the solution of any problem that can be solved by instructions, explicitly stated rules, or procedures' (Gregory 1987: 784). Dennett maintains that 'Turing had proven - and this is probably his greatest contribution - that his Universal Turing machine can compute any function that any computer, with any architecture, can compute' (1991: 215). Sterelny asserts 'Astonishingly, Turing was able to show that any procedure that can be computed at all can be computed by a Turing machine. ... Despite their simple organisation, Turing machines are, in principle, as powerful as any other mode of organizing computing systems' (1990: 37, 238). (Astonishingly, indeed!) In similar vein, Paul Churchland writes: 'The interesting thing about a universal Turing machine is that, *for any well-defined computational procedure whatever, a universal Turing machine is capable of simulating a machine that will execute those procedures*. It does this by reproducing exactly the input/output behaviour of the machine being simulated' (1988: 105; Churchland's italics). Also: Turing's 'results entail something remarkable, namely that a standard digital computer, given only the right program, a large enough memory and sufficient time, can compute *any* rule-governed input-output function. That is, it can display any systematic pattern of

responses to the environment whatsoever' (Paul and Patricia Churchland 1990, p.26). Phillips, writing in the 'Handbook of Logic in Computer Science', avers that 'Turing's analysis of what is involved in computation ... seems so general that it is hard to imagine some other method which falls outside the scope of his description ... so ... anything which can be computed can be computed by a Turing machine' (Abramsky, Gabbay and Maibaum 1992: 123).

Of course, Turing neither proved nor asserted that a universal Turing machine ‘can compute any function that any computer, with any architecture, can compute’. Nor did he have a result entailing that a Turing machine can display any systematic pattern of responses to the environment whatsoever. Indeed, he had a result entailing the opposite. The halting theorem entails that there are possible patterns of responses to the environment, perfectly systematic patterns, that no Turing machine can display.

Turing offered the Turing machine as an analysis of the activity of an (idealised) human mathematician engaged in the process of computing a real number unaided by any machinery (1936: 231). His concern in 1936 was with the theoretical limits of what an unaided human mathematician can compute, the whole project being directed toward showing, in answer to a question famously raised by Hilbert, that there are classes of mathematical problems whose solutions cannot be discovered by a mathematician working mechanically. Turing’s actual thesis, the Church-Turing thesis properly so-called, that the limits of what an ideal human mathematician can compute coincide with the limits of what a universal

Turing machine can compute, is a thesis that carries no implication concerning the limits of what a *machine* can compute.[11]

Naturally, the crucial question is: Are there real physical processes that can be harnessed to do the work of Turing's 'oracles'? If so, computing machines that are forbidden by the thesis improperly known as the 'Church-Turing thesis' can in principle be constructed. Leaving the issue of harnessability to one side, I think it would be profoundly surprising if the physics of our world can fully be given without departing from the set of Turing-machine-computable functions. These functions have been the focus of intense interest during the brief six decades since Turing delineated them, but the explanation of this is surely their extreme tractability - together, of course, with the fact that they have made many people a lot of money - rather than that some inherent suitability for exhaustively describing the structure and properties of matter is discernible in them. These functions are the fruit of Turing's analysis of the activity of an idealised human mathematician working mechanically with pencil and paper. It is simple anthropomorphism to expect such functions to suffice also for characterising the behaviour of the rest of the universe. It would be - or should be - one of the greatest astonishments of science if the activity of Mother Nature were never to stray beyond the bounds of the Turing-machine-computable functions.

5. Internal and External Computability

There exists an accelerated Turing machine $\mathbb{H}_T$ that is capable of mimicking $\mathbb{H}$. As with $\mathbb{H}$, $\mathbb{H}_T$ is set in motion bearing the concatenation of a

---

[11] See further my 1996, 1997a, 1997b.

pair of integers. $\mathbb{H}_T$'s first actions are to position the head over the first square to the left of the input string - call this the designated square - and print 0 there. The remainder of $\mathbb{H}_T$'s program is identical to $\mathbb{H}$'s except that the instruction to blow the hooter is replaced by a block of instructions that cause the head to move back to the designated square and change the 0 to 1 before halting. Since $\mathbb{H}$ computes the halting function for Turing machines and $\mathbb{H}_T$ mimics $\mathbb{H}$ exactly (save for writing 1 on the designated square instead of hooting), it follows that $\mathbb{H}_T$ computes this function too. For any x and y, if $\mathbb{H}_T$ is set in motion bearing the data number x^y, the value H(x,y) can be read from the designated square at the end of the second moment of operating time. But according to the halting theorem, the halting function is not Turing-machine-computable. A contradiction, it seems.

At bottom, the reason that this contradiction cannot validly be derived is that the halting theorem is in fact a somewhat weaker proposition than is often supposed. It is time to be more precise in stating the halting theorem.

Standardly, to say that a function f(x) each of whose values is 0 or 1 is Turing-machine-computable is to say that there is a Turing machine which, if set in motion bearing x as data number (for any x in the function's domain), will eventually halt with its head resting on the corresponding value of the function. (Similarly for functions of more than one argument and for functions whose values include integers other than 0 and 1. In the latter case it is stipulated that the head should halt resting on, say, the last

digit of the function's value.) The two pioneering papers, Turing 1936 and Post 1936, are both clear on this crucial matter.[12]

The halting theorem is this: *in the sense just given* the halting function for Turing machines is not Turing-machine-computable. Only this much was proved by the reductio ad absurdum presented earlier, for it is crucial to the construction employed in the reductio that the machine h should halt with the value of the function, 0 or 1, beneath its head. Unless this is so the recipe for obtaining h_1 from h is inapplicable.

It is essential to distinguish between two senses in which a function may be said to be computable by a given machine, which I will refer to as the *internal* sense and the *external* sense, respectively. A function is computable by a machine in the internal sense just in case the machine can produce values from arguments (for all arguments in the domain), 'halting' once any value has been produced, where what counts as 'halting' can be specified in terms of features internal to the machine and without reference to the behaviour of some device or system - e.g. a clock - that is external to the machine. (This condition on the nature of 'halting' behaviour will be referred to as the internalist condition.) Numerous behaviours on the part of a machine can satisfy this condition, for example complete cessation of activity, or playing the National Anthem[13], or writing any sequence of digits in a certain location. A function is computable by a machine in the external sense just in case the machine can produce values from arguments (for all arguments in the domain), displaying each value at

---

[12]Post gave an explicit statement to that effect (1936: 103). Turing offered no explicit statement but that such was his intention is evident (see especially 1936: 247).

[13]See note **seven**.

a designated location some pre-specified number of moments after the corresponding argument is presented. The machine may or may not 'halt', in the internalist sense, once a value has been displayed.

For example, it is in this latter sense that a given function may be computable by a logic circuit (one of Turing's own boolean networks, for instance). The value of the function is displayed at some designated node n moments after the argument is presented at the input nodes (mutatis mutandis for functions of more than one argument and functions whose values require more than one binary node for their expression). Before and after that critical time, the activity of the output node may afford no clue as to the desired value.

Even where the logic circuit never stabilises (in the sense of eventually producing an output signal that remains constant until such time as the input signal alters) it nevertheless computes values of a function in the external sense if it displays them at the designated location at the prespecified times. The same is true of neural networks. A particular network may compute the values of a certain function in the external sense even though the network never stabilises (a network stabilises, or 'halts', if and only if after some point there is no further change in the activity level of any of its units).

In some cases, a machine may compute the values of a function in both senses, there being an n such that whenever an argument is presented, the machine 'halts' displaying the corresponding value within n moments, such 'halting' satisfying the internalist condition. A machine that computes a function in the external sense can readily be converted into one that computes the function in both senses by the addition of a bell triggered by an internal clock. The bell rings when the value (or, as appropriate, the last digit of the value) is produced, and the machine's

ringing the bell constitutes its 'halting'. Of course, adding a clock to a machine may result in a machine not of the same type. A Turing machine plus a clock is not a Turing machine.

The halting theorem speaks only of computability by a Turing machine in the internal sense, not of computability in the external sense. This is what I meant when I said earlier that the halting theorem is weaker than is often supposed. A Turing machine cannot compute the halting function for Turing machines in the internal sense - as was proved earlier - but this same function is computable by a Turing machine in the external sense, as witnessed by $\mathbb{H}_T$. A machine derived from $\mathbb{H}_T$ by the addition of a clock and bell computes the function in the internal sense. The function is computable by $\mathbb{H}$ in the external sense but not in the internal sense. It is computable by an O-machine in the internal sense, as witnessed by $O_{\mathbb{H}}$. There are functions that can be computed by higher-order O-machines in the internal sense that cannot be computed by a first-order O-machine - such as $O_{\mathbb{H}}$ - in either the internal or the external sense.

While it may grate on one's ear to say so, it is nevertheless perfectly true that (even) a Turing machine can compute functions (in the external sense) that are not Turing-machine-computable (in the standard, internal sense). Again, so much the worse for the Church-Turing thesis improperly so-called, the claim that if a function can be computed at all then it is Turing-machine-computable (in the standard, internal sense).

REFERENCES

Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds) 1992. *Handbook of Logic in Computer Science*. Vol.1. Oxford: Clarendon Press.

Ambrose, A. 1935. 'Finitism in Mathematics (I and II)'. *Mind*, 35, 186-203 and 317-40.

Blake, R.M. 1926. 'The Paradox of Temporal Process'. *Journal of Philosophy*, 23, 645-54.

Boolos, G.S., Jeffrey, R.C. 1980. *Computability and Logic*. 2nd edition. Cambridge: Cambridge University Press.

Churchland, P.M. 1988. *Matter and Consciousness*. Cambridge, Mass.: MIT Press.

Churchland, P.M., Churchland, P.S. 1990. 'Could a Machine Think?'. *Scientific American*, 262 (Jan.), pp.26-31.

Copeland, B.J. 1996. 'The Church-Turing Thesis'. In Perry, J., Zalta, E. (eds) *The Stanford Encyclopaedia of Philosophy*, World Wide Web: Stanford University, 25 screens. Available from http://plato.stanford.edu/.

Copeland, B.J. 1997a. 'The Broad Conception of Computation'. *American Behavioral Scientist*, forthcoming.

Copeland, B.J. 1997b. 'Turing's O-machines, Searle, Penrose and the Brain'. *Analysis*, forthcoming.

Copeland, B.J. 1998. *Turing's Machines*. Oxford: Oxford University Press, forthcoming.

Copeland, B.J., Proudfoot, D. 1996. 'On Alan Turing's Anticipation of Connectionism'. *Synthese*; 108, 361-377.

Copeland, B.J., Sylvan, R. 1997. 'Computability: A Heretical Approach'. Forthcoming.

Dennett, D.C. 1978. *Brainstorms: Philosophical Essays on Mind and Psychology*. Brighton: Harvester.

Dennett, D.C. 1991. *Consciousness Explained*. Boston: Little, Brown.

Fodor, J.A. 1981. 'The Mind-Body Problem'. *Scientific American*, 244 (Jan.), 124-32.

Fodor, J.A. 1983. *The Modularity of Mind*. Cambridge, Mass.: MIT Press.

Franklin, S., Garzon, M. 1991. 'Neural Computability'. In O. Omidvar (ed.) 1991, *Progress in Neural Networks*, vol. 1, Norwood, N.J.: Ablex.

Gregory, R.L. 1987. *The Oxford Companion to the Mind*. Oxford: Oxford University Press.

Hogarth, M.L. 1994. 'Non-Turing Computers and Non-Turing Computability'. *PSA 1994*, 1, 126-38.

McArthur, R.P. 1991. *From Logic to Computing*. Belmont, Calif.: Wadsworth.

McCulloch, W.S., Pitts, W. 1943. 'A Logical Calculus of the Ideas Immanent in Nervous Activity'. *Bulletin of Mathematical Biophysics*, 5, 115-33.

Minsky, M.L. 1967. *Computation: Finite and Infinite Machines*. Englewood Cliffs, N.J.: Prentice-Hall.

Post, E.L. 1936. 'Finite Combinatory Processes - Formulation 1'. *Journal of Symbolic Logic*, 1, 103-5.

Russell, B.A.W. 1915. *Our Knowledge of the External World as a Field for Scientific Method in Philosophy*. Chicago: Open Court.

Russell, B.A.W. 1936. 'The Limits of Empiricism'. *Proceedings of the Aristotelian Society*, 36, 131-50.

Sterelny, K. 1990. *The Representational Theory of Mind*. Oxford: Basil Blackwell.

Stewart, I. 1991. 'Deciding the Undecidable'. *Nature*, 352, 664-5.

Turing, A.M. 1936.'On Computable Numbers, with an Application to the Entscheidungsproblem'. *Proceedings of the London Mathematical Society*, Series 2, 42 (1936-37), 230-265.

Turing, A.M. 1938. 'Systems of Logic based on Ordinals'. A dissertation presented to the faculty of Princeton University in candidacy for the degree of Doctor of Philosophy.

Turing, A.M. 1939. 'Systems of Logic Based on Ordinals'. *Proceeding of the London Mathematical Society*, 45, 161-228.

Turing, A.M. 1948. 'Intelligent Machinery'. National Physical Laboratory Report. In B. Meltzer, D. Michie (eds) 1969, *Machine Intelligence 5*, Edinburgh: Edinburgh University Press.

Turing, A.M. 1951. 'Programmers' Handbook for the Manchester Electronic Computer'. University of Manchester Computing Laboratory.

Weyl, H. 1927. *Philosophie der Mathematik und Naturwissenschaft*. Munich: R. Oldenbourg.

Weyl, H. 1949. *Philosophy of Mathematics and Natural Science*. Princeton: Princeton University Press.