

# Computation and Hypercomputation

Mike Stannett

Dept of Computer Science, Regent Court, University of Sheffield

211 Portobello Street, Sheffield S1 4DP, United Kingdom

Email: *m.stannett@dcs.shef.ac.uk*

## ABSTRACT

Does Nature permit the implementation of behaviours that cannot be simulated computationally? We consider the meaning of physical computability in some detail, and present arguments in favour of physical hypercomputation: for example, modern scientific method does not allow the specification of any experiment capable of refuting hypercomputation. We consider the implications of relativistic algorithms capable of solving the (Turing) Halting Problem. We also reject as a fallacy the argument that hypercomputation has no relevance because non-computable values are indistinguishable from sufficiently close computable approximations.

In addition to considering the nature of computability relative to any given physical theory, we can consider the relationships between physical computationalities. Deutsch and Penrose have argued on mathematical grounds that quantum computation and Turing computation have equivalent formal power. We show this equivalence to be invalid when considered from the physical point of view, by exhibiting a quantum computational behaviour that cannot meaningfully be considered feasible in the classical universe.

---

Keywords: Hypercomputation, super-Turing machine, recursion, computability, Church-Turing Thesis, scientific method, mathematical physics.

---

## 1 Introduction

Does Nature permit the implementation of behaviours that cannot be simulated computationally? Such behaviours, if they exist, are said to be *hypercomputational*. A hypercomputational system is therefore one that “computes” non-computable behaviours. For a range of recent viewpoints, see (Bain & Johnson 2000, Copeland & Sylvan 1999, Davies 2001, Siegelmann 1999, Zhong & Weihrauch 2000).

This use of the term ‘computation’ in both a physical and a conceptual sense causes a great deal of confusion, so I shall always use the word *computation* to mean the formal functional behaviour of a conceptual computer program (unless otherwise stated a ‘computer’ is any conceptual device equivalent to a universal Turing machine (UTM)). Physical systems will instead be called *implementations*. Consequently, the word ‘computable’ means ‘computable by Turing-machine,’ while an ‘implementable function’ is one that can be implemented by a physical system. If an implementation happens to implement a computable function, I call it a *computational* system. Notice that *computation* and *computational* apply to different domains: physical systems can be computational, conceptual behaviours can be computations. The statement ‘Nature is computable’ means ‘all implementations are computational’ and the statement

[HC]  $\equiv_{\text{def}}$  ‘hypercomputation is feasible’

---

means ‘there exists at least one implementable behaviour that is not computational.’ I’ll sometimes write [¬HC] as an abbreviation for “Nature is computable.”

It is often taken for granted that physical implementations of formal computers can be built, but this is in fact by no means obvious. As part of this investigation, I therefore consider the status of the statement

[TC]  $\equiv_{\text{def}}$  “Nature permits the implementation of (universal) Turing computation”

Knowing that a single behaviour is hypercomputational is potentially useful, but the approach I discuss here is more properly described as the investigation of hypercomputational *strategy*; I want to construct hypercomputational systems *deliberately*. Accordingly, I’ll restrict the word *machine* to mean a deliberately constructible physical system capable of performing physical processes according to deliberate ‘intent.’ Implementations of computers tend to use discrete instruction sets, but there is no need to impose this as a general precondition; instructions need not be spatially or temporally discrete, nor need they be finitely specified – they simply need to be physically producible as required. They need not even be explicit. For example, I do not know how to specify the processes involved in radioactive decay (I do not even know whether radioactive decay is a finitely specifiable process), but provided radioactive decay can be *used* in a mechanistic way, then it can be used both to control, and as a component of, a machine. If a machine can behave in hypercomputational ways, I call it a *non-Turing machine*. If it can also simulate a universal Turing machine, I call it a *super-Turing<sup>1</sup> machine* (STM). An STM is *more powerful* than a universal Turing machine, whereas a non-Turing machine might simply exhibit *different* potential. My personal position of these concepts is somewhat stronger than [HC]; I hold to the feasibility of building super-Turing machines (which entails both [TC] and [HC]); formally:

[ST]  $\equiv_{\text{def}}$  “Nature permits the existence of super-Turing machines”

Throughout this paper, I shall present several independent arguments in favour of [HC] (and to a lesser extent [ST]), drawing on ideas from a variety of different disciplines.

In *Section 2*, my analysis of physical computability begins with a discussion of the Church-Turing Thesis [CT]. Many papers that cite [CT] assume that its content and context are familiar to readers, but the evidence suggests otherwise – Copeland (1997, 2000) points out that many discussions of [CT] dramatically misrepresent the Thesis by turning it from a statement about a carefully delimited class of mechanistic human behaviours into a statement about anything that can ever be computed by any means whatever. I have therefore included a brief *Appendix* of technical concepts (the thesis itself is described in the main text). Arguments based on misunderstandings are inherently suspect, but this should not deter us from asking how far their proponents have succeeded in uncovering a basic truth, even if only by accident. To what extent *does* [CT] tell us about the limits of physical computation?

In *Section 3* I consider the nature of observation and scientific experiment, and argue that no experiment conducted in the Natural Science tradition could ever refute hypercomputation. From a logico-scientific viewpoint, then, [HC] is either true or experimentally undecidable; it cannot be

---

<sup>1</sup> Some authors object to the terms ‘non-Turing’ and ‘super-Turing’ because Turing never denied the feasibility of hypercomputation; they use instead the word ‘hypercomputer.’ I acknowledge the validity of their complaint, but nonetheless avoid the use of ‘hypercomputer’ because this already has an established, and rather different, meaning in the Computer Science literature (a hypercomputer is a kind of parallel distributed super-computer). The term ‘super-Turing machine’ should be understood to mean ‘super-(Turing machine),’ *i.e.* a system that is provably more powerful than a Turing machine.

---

refuted. By negation, the statement ‘Nature is computable’ is either false or experimentally undecidable.

In *Section 4*, I consider the status of [ST] as one moves from a classical to modern theories of physics. I summarise Hogarth’s work on relativistic algorithms, and their inherent hypercomputationality. I also consider whether quantum computation has anything to offer. I argue that quantum *machines* are indeed more powerful than classical machines, even though quantum computers are no more powerful than classical computers. This can be seen as evidence against [TC] (the feasibility of universal Turing computation) in a classical Newtonian universe.

In *Section 5* I counter the common argument that hypercomputation is essentially irrelevant, because even if hypercomputational values could be implemented we would be unable to distinguish them experimentally from sufficiently good computable approximations.

In *Section 6*, I conclude by summarising my observations, and suggest some questions that might benefit from future research.

## 2 Hypercomputation and the Church-Turing Thesis

What do we mean by computation? For mathematicians and computer scientists this question is approached by breaking computational behaviours down into a relatively small number of basic actions, and asking how these actions can be combined. Attempts to do this in the 1930s resulted in the principle now known as the Church-Turing Thesis [CT], that the numerical functions that can be evaluated by “human clerical labour, working to fixed rule, and without understanding”<sup>2</sup> are precisely those functions that can be evaluated by computer (*i.e.*, universal Turing-machine). This is an assertion that two sets of functions, one defined mathematically and one physically, are actually the same set of functions.

One of the key features of Turing’s machine-based model of computation is the ease with which arguments in support of [CT] can be stated, and (as we illustrate below) these arguments become even easier when we replace Turing’s low-level machines with the modern digital machines with which we are all familiar. It is surprising, therefore, that so many authors appeal to [CT] rather than simply adapting the original arguments to generate their own direct proofs. If one truly believes that [CT] justifies claims like “if a psychological science is possible at all, it must be capable of being expressed in computational terms”<sup>3</sup> or “a standard digital computer, given only the right program, a large enough memory and sufficient time, can compute *any* rule-governed input-output function”<sup>4</sup> then there should be little difficulty in adapting Turing’s arguments to demonstrate these claims convincingly from first principles. If a direct proof is not forthcoming, the matter is probably not so clear-cut after all.

Few practising computer scientists would consider the Turing machine an obvious device around which to base a discussion of modern programming practice, in part because Turing machines are unfamiliar to their audiences, and in part, because it is usually unhelpful to address matters from such a low-level perspective. In contrast, philosophical discussions of computation regularly invoke the Turing machine or some equivalent (and equally unfamiliar) 1930s formalism, and otherwise accurate authors outside the discipline can easily be drawn by this unfamiliarity into making unjustified statements about formal computation and its relationship to physical and biological

---

<sup>2</sup> Turing, 1946, p. 38-9.

<sup>3</sup> Boden, 1988, p. 259.

<sup>4</sup> Churchland, 1988, p. 105.

---

systems. I shall tentatively adopt the computer science approach, and attempt to express [CT] in terms that are familiar from everyday experience of digital computing machinery. This will, potentially, remove some of the confusion.

In the 1930s, a ‘computer’ was a person who performed calculations manually in a fairly mechanistic way, and Turing’s machine-based conceptualisation reflects this:

“A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.”<sup>5</sup>

These days a ‘computer’ is an electro-mechanical box of machinery, so to avoid confusion I shall use the term ‘clerk’ where Turing’s contemporaries might have said ‘computer.’ Thinking in terms of modern software systems, which are essentially Turing’s machines ‘made flesh,’ it is easy to see why [CT] ought to be true. The tasks involved in such chores as handling lengthy calculations manually are rather basic: clerks are given lists of numbers to be added or subtracted as instructed, and their results contribute to the lists of numbers handled by other designated clerks. The process of delegating sub-calculations to other clerks or groups or clerks is carefully controlled, and once a calculation is completed, it is possible to observe that this has happened. Each of the basic operations described here – adding and subtracting numbers, initiating delegated sub-calculations, writing results onto a designated list, and observing that a well-defined “I’ve finished and here’s my result” flag has been hoisted – can easily be simulated in software, so it’s reasonable to conclude that anything clerks can calculate, *given that they’re operating in this highly artificial way*, can also be calculated by a program with even fairly basic operations at its disposal. Conversely, no matter how high-level the software language used to program a numeric operation, it can be simulated by a lower-level program using only basic operations like addition and subtraction (see the *Appendix* for a detailed example), and such a program can in turn be simulated by human beings pretending to be the computer. This ‘play-acting’ process only requires the ability to operate in the clerks’ mechanistic way, following instructions, but using no intuition or intelligence – so anything the program can do, clerks can do. We call this sort of mechanistic human behaviour ‘effective,’ and [CT] asserts the intuition that when human beings work in an effective way they are neither capable of more nor less than a standard digital computer.<sup>6</sup> Therefore, putting the matter rather crudely, [CT] makes the at-first-sight unsurprising claim that

[CT]: when human beings pretend to be computers they behave like computers.

This version of [CT] is neither fair nor accurate, but it does show why the many published descriptions of [CT] as a claim to the effect that ‘any function that can be computed by any means whatsoever can be computed by a Turing-machine’ are nonsense (in fact Turing specifically described behaviours that *cannot* be simulated by his machines<sup>7</sup>). The reason our description is unfair is that it gives the impression that [CT] is simply an analytic statement defining what it means to ‘pretend to be a computer’ — in effect, that it says nothing at all. On the contrary, [CT] asserts two important alignments between the formal domain of conceptual mathematics and the experiential domain of experimental physics (see *Figure 1*), viz.,

- humans can, in principle, simulate computers; and
- the computational human behaviours are precisely the effective human behaviours.

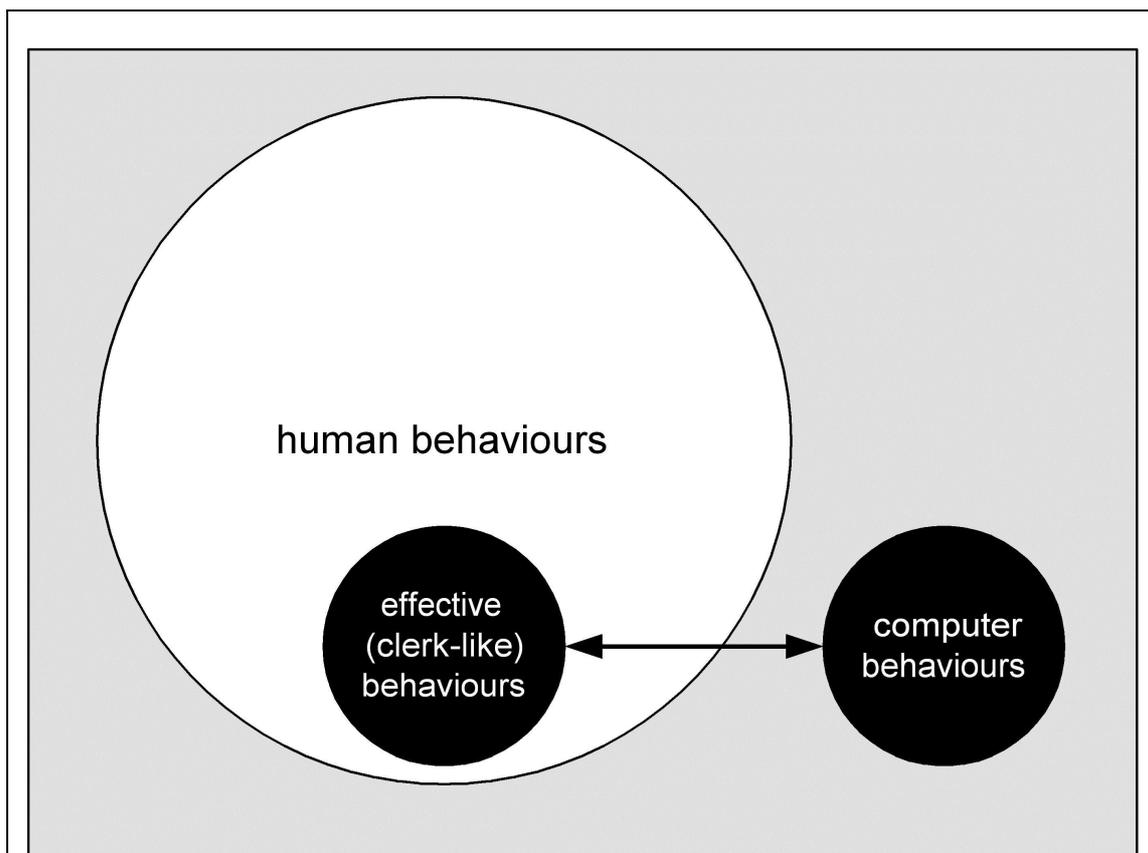
---

<sup>5</sup> Turing, 1948, p. 9.

<sup>6</sup> Assuming that both human and computer are given access to as many resources as are needed at any given time to continue their work, and that no limit is placed on the amount of time they’re allowed to take over completing the task.

<sup>7</sup> Turing’s ‘O’-machines explicitly use oracles whose behaviours cannot be simulated by Turing-machine (Copeland, 2000). See the *Appendix* for an explanation of oracles.

---



*Figure 1. The Church-Turing Thesis [CT]*

The fact that [CT] and ‘effectiveness’ are typically applied well beyond the domain of human behaviours is a testament to the power of Turing’s model. We’ve seen that the effective behaviours displayed by clerks are precisely the same (in idealised functional terms) as those displayed by modern software systems. Since these software systems can simulate everything from music synthesizers to washing machine control systems and automatic theorem provers, so can effective humans. According to [CT], effective humans behave identically to computers, so the behaviours of synthesizers, washing machine control systems and automatic theorem provers are all computable, and can all be described as ‘effective.’

Both aspects of [CT] are important. Computers (Turing’s ‘machines’) are conceptual devices, not physical ones, and the assertion that clerks and computers behave identically is a deep statement to the effect that a relationship can, does, and must exist between the effective physical behaviours of human beings on the one hand, and the mathematical behaviours that underpin formal arithmetic on the other. [CT] specifically tells us that computers can be implemented, in principle, by play-acting human beings, so it entails [TC].<sup>8</sup> The second aspect, that the set of computational behaviours is in a fundamental sense identical to the set of effective behaviours, is important because it is typically the only means available for demonstrating that a physical process is computational. For example, suppose two perfectly hard billiard balls of equal mass, each moving at exactly  $1 \text{ ms}^{-1}$ , collide with one another head-on in a Newtonian universe, on a frictionless flat table. The simplicity of this system ensures that we can design a software simulation of the collision. Since we’ve already seen

<sup>8</sup> More accurately, [CT] entails [TC] in any universe that satisfies the ‘unlimited resources’ premise of universal Turing computation.

that programs can be simulated, in principle, by effectively operating humans, we conclude that simple Newtonian collisions can also be simulated effectively. Invoking [CT] now allows us to deduce that the collision process is computational. The fundamental equivalence of effective behaviours and software behaviours also justifies calling today's software-driven digital machines 'computers' — it is because they are essentially equivalent to effective humans that [CT] can be invoked to declare their behaviours computational. Indeed, [CT] is usually described without reference to humans at all, with effectiveness *defined* to be 'computing-machine-like behaviour.'

The Church-Turing Thesis can be seen as strong evidence in favour of hypercomputation, because it tells us that implementing a computer is as easy as getting a human to play-act at having no intuition, no volition, no self-awareness, no intelligence – in short, none of the characteristics we normally associate with being a human being in the first place. In today's computer-centric society, it is normal to misread statements that compare humans with computers, and one standard misreading interprets [CT] to mean "if a human is like a computer, then humans must be fundamentally simple." This misreading is essential to many research areas, since there would otherwise be little prospect of simulating human behaviours computationally, and much modern AI and Philosophy of Mind would be based in fantasy. But this is exactly what [CT] does *not* say. It tells us that *computers* are fundamentally simple, because humans can do everything computers can, even with their minds turned off.<sup>9</sup> And if universal computation can be achieved by something as simple as a mindless human, it would be amazing if the wider Universe, with all the supernova explosions, gravity wells, vacuum fluctuations and as-yet-undiscovered complex behaviours at its disposal, were able to support nothing more powerful. Yet, this is precisely what people assert when they deny the feasibility of hypercomputation.

Of course, they might be right! — just because a small component of the universe can simulate computation, it does *not* follow that larger components must be capable of more. We can see this more clearly if we consider precisely what we *mean* by physical behaviours. The question, whether there really exists a universe 'out there' for us to observe, has exercised philosophers for centuries, but in the present context I contend that the physical properties we are interested in definitely *do not* have independent meaning — they acquire it instead by reference to the explanatory framework of a mathematico-physical theory. My claim here is not intended as a statement about all physical behaviours, but just the ones that are complex enough to have potential relevance to hypercomputation. In one sense this is obvious: the property we are interested in (computability) is *mathematical*, and as such cannot directly be said to apply to a *physical* system; it can only be said to apply to a *mathematical model* of that system, and the construction of such models entails the use of a physical theory. [CT] allows us to bridge the gap to some extent, and some physical processes seem in any event to be 'obviously' computable — for example, the simple Newtonian collision described above. However, when we come to *complex* systems the situation is very different. No one has seen a black hole, touched an exploding supernova, tasted a vacuum fluctuation, or made *any* direct observation of gravity or momentum or electrons. Rather, experimental observations of a much more straightforward nature have been made, and theories have been erected in an attempt to explain these observations. For example, we might *observe* the existence of dark lines in the coloured bands produced when sunlight passes through a prism and a vapour, and *explain* this observation by proposing that atoms in the vapour contain electrons which can absorb photons whilst simultaneously jumping from one well-defined energy state to another. However, strictly speaking, the most we can state with any certainty is that sunlight passing through a prism and a vapour generates bands of coloured light, which are interrupted in various places by dark lines. Our

---

<sup>9</sup> Indeed, an alternative reading of human-computation relationships – the socio-industrial statement that 'forcing humans to behave like computers is dehumanising' – logically depends on humans being hypercomputational.

---

observations have the potential to be things-in-themselves with direct physical relevance, but the electrons, atoms and photons invoked to explain these observations are simply artefacts of our explanatory theory and do not necessarily correspond to anything in the ‘real-world.’ They are, in effect, simply names we assign to statistical collections of behaviours that have a tendency to occur together; an electron may or may not be a thing-in-itself, but for the purposes of physical theory it is more useful to define *things* to be *behavioural tendencies*: an electron is anything that behaves in the way we expect an electron to behave. Even this behavioural style of definition is not absolute — when we say that light travels in straight lines, the meaning of this statement depends on the accepted semantics of ‘straight lines.’ In other words, the behavioural ensembles we regard as *things* acquire meaning only when they are *grounded* in some particular physical theory.

Though seemingly abstract, this state of affairs raises important objections to the existence of hypercomputational behaviours, because standard physical theories tend to express behaviours as mathematical equations, and these can be used to generate software simulations. For example, the physical equation stating that light travels in straight lines might be simulated by a graphics routine that turns all pixels in a certain collection yellow, where the collection is asserted to contain precisely those pixels whose coordinates are solutions  $(x, y)$  of some equation  $Line(x_{min}, x_{max}, slope, intercept) \equiv "(x_{min} \leq x \leq x_{max}) \ \& \ (y = slope \times x + intercept)"$ . Since these interpretations seem to ensure that the ‘graphical universe’ is as much a model of the physical theory as the ‘physical universe’ on which it is based, any physical behaviour that can be simulated graphically ought to be computational — whatever can be simulated computationally is computational.

Does this mean that physical hypercomputation is necessarily infeasible? Not necessarily, because we have made various unstated assumptions. It is only if a physical theory is consistent and grounded in computable axioms, and fully explains the given physical behaviour, that our argument applies. By implication, therefore, a physical behaviour can potentially be hypercomputational if one or more of the following assertions holds for the physical theory relative to which it is described:

- the theory is not grounded in computable relationships
- the theory does not explain the behaviour in question
- the theory is logically inconsistent

Which of these situations can arise in practice?

- The idea that fundamental physical laws might not be computable is problematic but probably untenable in practice. It is certainly true that physical equations can involve uncomputable operators like general integration, but this has little significance because physicists typically use numerical approximation methods whenever computational difficulties arise. In effect, the *true* physical theory (the theory actually used by physicists) is the one involving the computational approximations; the idealised version is simply a convenient shorthand description.
  - There are many important processes for which no complete explanation is known. No currently established physical theory (I know of) can explain why a radioactive atom should decay at one particular moment rather than another. Perhaps explanations of this particular behaviour will one day be forthcoming, but in any event, the claim that physical theories cannot fully explain behaviours is logically irrefutable. This is a consequence of indistinguishability (*see below*) — because we cannot distinguish the ‘true’ process from any sufficiently close approximation, we have no way of deciding whether the physical model we’re proposing is exactly accurate, or just a good approximation.
-

- The third possibility, that physical theories might be inconsistent, is particularly interesting, because one can argue that this is in fact the case for the current Standard Model of physics.<sup>10</sup> In general, however, it is probably safe to assume that the tendency towards parsimony in choosing fundamental physical laws ensures that physical theories are generally consistent.

In summary, behaviours that can be explained by physical theory are potentially computational, but there are behaviours that cannot be explained, and in any event, indistinguishability means that we cannot necessarily regard explicability as evidence that Nature is computational.

### 3 Hypercomputation is experimentally irrefutable

At the heart of the Standard Model lies the axiomatic assumption that experimental observation is the only meaningful evidence of existence, where observation is itself meaningful only if it satisfies the constraints of “scientific method.” In particular, an experimental result is only considered truly valid if it can be replicated by independent parties. By definition, replication of an experiment requires that it be recursively defined – independent scientists should be able to repeat the same experimental steps in the same order, starting from the same carefully prepared initial conditions, and so generate the same results, a concept that is only meaningful if the experiment is viewed as a recursive operation to be conducted on a recursively recreated environment. Since the constraints of scientific method amount to a requirement of recursive replicability, the basic framework of physics places implicit faith in the completeness of recursive methods for determining physical reality. The Standard Model is therefore an intrinsically recursive representation of Nature, and it is not surprising that physical hypercomputation should be so difficult to demonstrate – the very language of modern physics precludes direct analysis of hypercomputational processes. It does not follow, however, that physics has nothing to say about hypercomputation. It means simply that any such statement is likely to rely on subtle arguments, and will probably require a deal of lateral thinking.

I have suggested elsewhere<sup>11</sup> that a hypercomputational system might be constructed using a radioactive sample. Underpinning the system’s behaviour is the assumption that, given any radioactive sample, there must eventually come a time when half of that sample has decayed – the average time required is just the half-life of the sample in question. If the sample decays by  $\alpha$ -radiation, the process causes it to lose mass as Helium nuclei are ejected, and we can count the number of seconds required for a given observable threshold in mass loss to be exceeded. Because decay is assumed in the Standard Model to occur stochastically, the number of seconds must be random, so this system can be regarded as an example of a “true random number generator.” We mean by this a system which is guaranteed to generate an output, and which can potentially generate any positive integer, but for which the eventual outcome cannot be determined *a priori*.

---

<sup>10</sup> I argue as follows: the Standard Model is partly based on equations that involve differentiation with respect to temporal coordinates. For such differentiation to be well defined, it must be possible to consider events that are arbitrarily close together in time. However, it is generally accepted as a consequence of the Standard Model that there is a shortest meaningful non-zero interval of time. Consequently, the Standard Model undermines its own mathematical premisses. It is interesting that this inconsistency seems not to matter; the Standard Model has remarkably good predictive power. This suggests the instrumentalist conclusion that the mathematical structures usually considered to constitute the Standard Model do not in fact constitute the model at all, but are simply a scaffold used by theorists to help guide their own internalised efforts. Just as scaffolding can be defective without the enclosed building collapsing, so the mathematics used to enunciate the Standard Model can be defective without the internalised model being undermined. This raises important issues, but they are not directly relevant to this paper — I intend discussing them at length elsewhere.

<sup>11</sup> Most recently at the UCL Workshop on Hypercomputation, May 2000.

---

It's well known that true *infinite* randomness of this type cannot be produced by a computer<sup>12</sup>. Therefore, radioactive decay ought to enable the construction of super-Turing machines.

Further reflection shows, however, that the assumption that half the sample will eventually decay is extremely subtle. In trying to express the assumption logically I find that two radically different interpretations are possible, and comparing the two interpretations allows us to deduce that:

- *either* the assumption is correct, and we can build a true infinite random-number generator; so [ST], and hence [HC], are both correct.
- *or* our assumption was flawed, but in this case, we can show that hypercomputation cannot be refuted by any experiment conducted according to the rules of the Standard Model (*i.e.*, the two concepts are logically independent).

Although radioactive decay is a familiar concept in modern physics, I have found it to have remarkably ambiguous semantics. This is disquieting because decay is of great importance to modern physics, as this popular description of the proton illustrates (but see Loez (1996) for a more formal overview):

Protons are essential parts of ordinary matter and are stable over periods of billions and even trillions of years. Particle physicists are nevertheless interested in learning whether protons eventually decay, on a time scale of  $10^{33}$  years or more. This interest derives from current attempts at grand unification theories that would combine all four fundamental interactions of matter in a single scheme. Many of these attempts entail the ultimate instability of the proton, so research groups at a number of accelerator facilities are conducting tests to detect such decays. No clear evidence has yet been found; possible indications thus far can be interpreted in other ways. ["Proton," *Microsoft® Encarta® 98 Encyclopedia*. ©1993-1997 Microsoft Corporation. All rights reserved.]

Given that protons are key components of every atomic nucleus, it is clear that many grand unification theories should entail the instability of *all* atomic species.

At the present time several hundred elementary particles are known, with mean lifespan ranging from the very long (protons  $> 10^{33}$  years), through the everyday (free neutron  $\sim 15$  minutes), to the vanishingly short ( $Z^0 \sim 10^{-25}$  seconds). However, what does "mean lifespan" actually mean in this context? The standard definition is straightforward. For example, if we start with one million free neutrons, we would expect that after roughly 15 minutes (the half-life of a free neutron) only about 500,000 would remain, the rest having decayed. Another 15 minutes later we would have only 250,000, and 15 minutes after that only 125,000. This notion of "half-life" is so familiar that it is commonly accepted as an unquestioned basis for experimentation. For example, one standard experiment in support of the relativistic notion that time slows down for fast moving objects involves measuring the half-life of cosmic rays. These fast-moving particles rain constantly down upon the Earth, and it is possible to measure how their number changes with height. The lower we take our measurement, the longer the particles will have been in transit and undergoing decay, and the lower their apparent flux will become. Experimental evidence confirms that the flux drops off much more slowly than would be expected given the amount of time involved (as measured by an

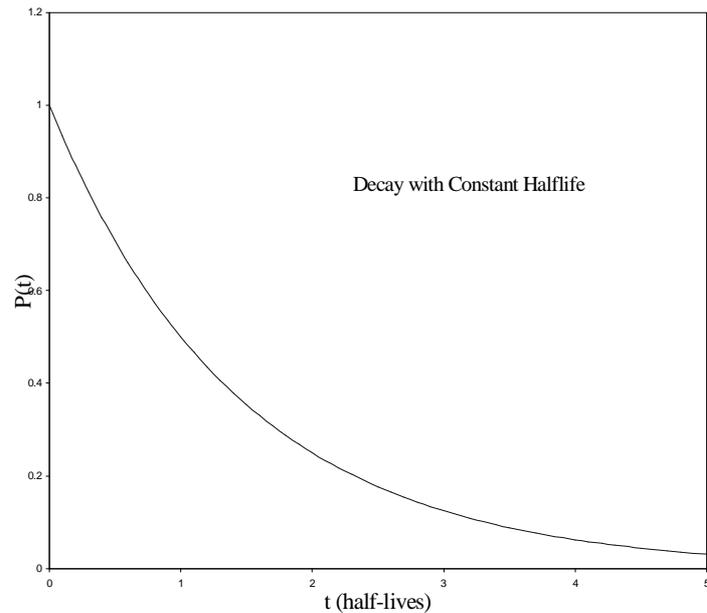
---

<sup>12</sup> König's Lemma says that the only way a finitely branching tree can contain infinitely many leaves is if it also contains an infinite path, a ZFC result known to be equivalent to the Axiom of Choice. Because the IF statements in a computer program only ever involve finitely many choices, the set of possible program behaviour traces constitutes a finitely branching tree. For a program to be a true random number generator, it must be guaranteed to terminate, and be capable of doing so in any one of infinitely many different ways. These give infinitely many leaves of the behaviour-trace tree, so by König's Lemma there is at least one behaviour that corresponds to an infinite path through the tree, *i.e.* the program is *not* guaranteed to terminate after all. Consequently, it is impossible for a computer program to simulate a true random number generator.

---

observer on the Earth), because the fast-moving particles consider the journey to have taken considerably less time.

Their actual usage of the concept suggests that physicists consider decay to be not merely possible but *necessary*, but this is nowhere expressed in the standard statistical definitions. I (and clearly most modern physicists) would “expect” roughly half of a neutron sample to have decayed after 15 minutes, but no one can guarantee it. The possibility always remains, however low the probability, that *none* of the neutrons will have decayed, even several days later. However, if this actually happened in practice, it is highly unlikely that a physicist would say “aha, there’s Nature reminding us that decay is a purely statistical concept that need not happen at all.” They’re far more likely to say “your counter is obviously broken.” This suggests two distinct semantics for the decay of unstable particles, one based in formal description, the other in pragmatic usage.



The rest of this section, which is quite formal in places, is organised as follows.

1. I state the two different versions of what it means to say that “unstable particles eventually decay.” I show how these two versions can be expressed as logical formulae,  $\phi$  and  $\psi$ .
2. I show that  $\phi$  and  $\psi$  are so similar that no experiment can distinguish between them.
3. I show that  $\phi$  is a tautology of the Standard Model, and that [HC] (in fact, [ST]) is a logical consequence of  $\psi$ .
4. It follows that hypercomputation is experimentally irrefutable in the Standard Model, because:
  - suppose an experiment refutes [HC]
  - since [HC] is a consequence of  $\psi$ , the experimental system is a counter-example to  $\psi$
  - since  $\phi$  and  $\psi$  cannot be distinguished experimentally, the experimental system is also a counterexample to  $\phi$
  - since  $\phi$  is a tautology of the Standard Model, the experiment actually refutes the Standard Model itself.

Suppose, then, that observation of an as-yet-undecayed unstable particle  $p$  begins now, at time *zero*. Let’s write  $E(p,t)$  to denote the *a priori* probability that  $p$  will still not have decayed at a time  $t$  in the future. The concept of half-life gives one particular shape to the notion that  $E(p,t)$  tends towards zero the longer we wait, *i.e.*  $E(p,t) \rightarrow 0$  as  $t \rightarrow \infty$ . Nevertheless, we can express this asymptotic behaviour logically in two similar, but significantly distinct, ways. In the following statements, the value  $\epsilon$  is always assumed to be positive.

**First Interpretation of  $E(p,t) \rightarrow 0$  as  $t \rightarrow \infty$** 

This interpretation says: any given particle  $p$  may or may not eventually decay, but the chances of its not having done so become arbitrarily close to zero as time passes. Formally,  $p$  satisfies

$$(\forall \epsilon)(\exists T(\epsilon))(\forall t)[ (t > T(\epsilon)) \rightarrow (E(p,t) < \epsilon) ]$$

*i.e.*, given any positive probability,  $\epsilon$ , no matter how small, there will be some amount of time,  $T$  (the exact value may depend on your choice of  $\epsilon$ ), such that the chances of  $p$  still existing at any given time  $t$  later than  $T$  are less than  $\epsilon$ . ■

**Second Interpretation of  $E(p,t) \rightarrow 0$  as  $t \rightarrow \infty$** 

This interpretation says: any particle,  $p$ , must eventually decay after some finite time  $T(p)$ , but we do not know in advance what value  $T(p)$  will take. We can assume, however, that if we were to watch many particles the various values would be distributed as expected. Formally,  $p$  satisfies

$$(\exists T(p))(\forall t)[ (t > T(p)) \rightarrow (E(p,t) = 0) ]$$

*i.e.*, there definitely will be some future time,  $T$  (which may vary from particle to particle, and which we may not actually be able to calculate), by which the particle has decayed. ■

The symmetry between the two interpretations isn't immediately obvious, so I'll apply a mathematical "trick" to help bring out the similarity. In the second interpretation, we're making the blunt assertion that  $E(p,t)$  will be zero, and consequently have no need for the test-parameter " $\epsilon$ " that appears in the first interpretation. We can re-introduce  $\epsilon$  by observing that *zero* can be defined as "that unique non-negative number which is less than every positive number." Consequently,

$$(\exists T(p))(\forall t)[ (t > T(p)) \rightarrow (E(p,t) = 0) ]$$

is the same statement as

$$(\forall \epsilon)(\exists T(p))(\forall t)[ (t > T(p)) \rightarrow (E(p,t) < \epsilon) ]$$

provided  $T(p)$  and  $E$  are independent of  $\epsilon$  and  $\epsilon$  is independent of  $p$ . Accordingly, we can write the two interpretations in strikingly similar terms, where  $\epsilon$  is again understood to be positive:

$$\alpha(p) \equiv_{\text{def}} (\forall \epsilon)(\exists T(\epsilon))(\forall t)[ (t > T(\epsilon)) \rightarrow (E(p,t) < \epsilon) ]$$

$$\beta(p) \equiv_{\text{def}} (\forall \epsilon)(\exists T(p))(\forall t)[ (t > T(p)) \rightarrow (E(p,t) < \epsilon) ]$$

Seen in this way, the distinction between the two statements lies in the way  $T$  is to be chosen. In  $\alpha$  we choose  $T$  according to the constraint  $\epsilon$ , but in  $\beta$  we assume that  $T$  is a property of the particle itself. To understand the experimental status of these statements, we need to consider  $E$  in more detail. We have assumed that  $E$  is the asymptotically zero exponential decay curve of the Standard Model (so that half-life is meaningful). During any experiment to establish the truth or falsity of  $\alpha$  or  $\beta$  we need to observe a particle  $p$  to see whether or not it has yet decayed; and by doing so we generate new information. In particular, this information forces us to amend our *a priori* assumptions concerning  $E$ . If we find that the particle has indeed decayed, for example, it is no longer meaningful to assign a positive probability to the possibility that it won't have decayed 5 seconds later. Rather, the moment we know that the particle has decayed, all subsequent *a priori* estimates of  $E$  automatically drop to zero.

Suppose then that  $p$  is a particle, and let's consider the relationship of  $p$  to  $\alpha$  and  $\beta$ . There are two cases to consider, depending on whether  $p$  is stable or unstable. If we assume that the particle is in fact stable, this assumption itself provides information that forces  $E$  to be amended. For a stable particle we have, by definition,  $E \equiv 1$ , and now both  $\alpha$  and  $\beta$  evaluate to *false*. On the other hand, if, as physicists increasingly suggest,  $p$  is bound to be unstable, we can consider two subcases. Either there is a time  $t_p$  after which the particle can be seen to have decayed, or there isn't. If there is such a time, then  $E$  has, at some point, to be adjusted so that  $E(p,t) \equiv 0$  for all  $t > t_p$ . Under these circumstances both  $\alpha$  and  $\beta$  automatically evaluate to *true*. If there is no such time, then no adjustment of  $E$  occurs, and we can distinguish  $\alpha$  from  $\beta$ , because  $\alpha$  evaluates to *true* but  $\beta$  evaluates to *false*.

To summarise, the only way we can distinguish  $\alpha$  from  $\beta$  is by finding a particle which is ostensibly unstable, but which never in fact decays. For pedants this ends the argument, because they can argue (with some justification) that an unstable particle that never decays is in fact a *stable* particle. However, even if we allow for the existence of *a priori* unstable particles that never decay, determining that a particle actually possesses this property requires waiting forever, which violates "scientific method" on two distinct grounds. Firstly, an experimental result is only meaningful if it can be replicated, and clearly, an experiment that runs forever can never be repeated thereafter<sup>13</sup>. Secondly, experiments must be conducted using "finite means," and this includes the requirement that they run to completion in finite time. However, if we restrict the experiments to run for no more than  $T$  seconds (say) they would clearly be unable to distinguish between particles that decay after  $T$  seconds from those which never decay.

COROLLARY:  $\alpha$  and  $\beta$  are experimentally indistinguishable.

Let's write  $Unstable(p)$  to mean that  $p$  is an *a priori* unstable particle. We've just seen that  $\alpha$  evaluates to *true* for all ostensibly unstable particles, so the statement

$$\phi \equiv_{\text{def}} Unstable(p) \rightarrow \alpha(p)$$

is a tautology (provided one accepts the asymptotic decay curve assumed by the Standard Model); indeed, it can be seen as the *definition* of instability in the Standard Model. Since  $\alpha$  and  $\beta$  are experimentally indistinguishable, we can regard the statement

$$\psi \equiv_{\text{def}} Unstable(p) \rightarrow \beta(p)$$

as an *experimental tautology*. That is, there may exist *logical* counterexamples to  $\psi$ , but any such counterexample is experimentally meaningless. However, this statement  $\psi$  is precisely the assumption underpinning our design of a true random number generator, so  $\psi$  necessarily entails both [ST] and [HC].

COROLLARY:  $\phi$  is a tautology of the Standard Model,  $\psi$  entails [ST], and any experimental refutation of [HC] is a refutation of the Standard Model itself. ■

## 4 Physical expressions of hypercomputation

The intent of [HC] — that hypercomputation is feasible — usually presupposes a classical Newtonian universe, since this is the type of universe for which Turing computation has traditionally been defined. In recent years, however, there has been much discussion of computation

---

<sup>13</sup> However, see the section on relativistic algorithms, below.

relative to other types of universal model, especially where quantum computation is concerned. The basic idea behind these models is that Turing's conceptualisation of computing machines is sufficiently physical that one can easily envisage how the classical components can be replaced with quantum counterparts (for example, if quantum uncertainty means the squares on a memory tape can't be accurately distinguished, then the tape becomes a fuzzy nondeterministic memory device whose contents can be known statistically but not absolutely). An alternative approach is to amend the type of space and time in which the machine operates, and this is essentially the approach taken when developing relativistic algorithms.

#### RELATIVISTIC STRATEGIES

This is a short section, because the concepts are better expressed elsewhere (Hogarth 1992, 2001, Earman & Norton 1993). Nonetheless, I want to draw attention to the implications of relativistic algorithms.

Relativistic models of physics tend to focus on cosmological questions relative to which the existence of any individual object is essentially insignificant, so the theory itself usually concerns the structure of spacetime rather than the structure of objects. An important issue concerns the topology of spacetime: is it flat (Euclidean) or curved, and does it have point-like holes in it (*e.g.*, black holes) or other singularities? The guiding principle is that any mathematical description of spacetime should satisfy certain fundamental cosmological equations — and one class of these descriptions has important consequences for hypercomputation.

Think for a moment of a black hole. As one approaches it, one can either fall into it or else circumnavigate it and continue on the other side. The hole can be thought of as an 'edge' of spacetime which just happens to lie at a finite distance from us. Now imagine a more general spacetime hole which comes into existence at a location  $\mathbf{x}$  at time  $t$  and then disappears again. If you move to  $\mathbf{x}$  just one second before  $t$  and stay at  $\mathbf{x}$ , the hole will eventually appear one second later and swallow you up; but you can always move sideways, let the hole appear and disappear, and then return to your original position to arrive one second after  $t$ . Doing the latter takes you to a point  $(\mathbf{x}, t+1)$  in the hole's future. These measurements of time are given by a stationary observer not near enough to the hole to be significantly affected. Malament-Hogarth spacetime allows such 'edges,' and attributes to them the following important property. If you allow yourself to fall into the edge, you find that even though outsiders think you disappeared after a finite amount of time, from your own point of view an infinite amount of time passes. If you circumvent the hole, you can reach the point  $(\mathbf{x}, t+1)$  just a finite amount of time after you left  $(\mathbf{x}, t-1)$ .

To see why this is relevant to hypercomputation, imagine that an observer starts a computer program,  $P$ , running on a computer at  $(\mathbf{x}, t-1)$ . The computer is allowed to fall into the edge, but is given the extra instruction that if  $P$  eventually generates an output, it should eject a small signal buoy. This travels *around* the hole and arrives at  $\mathbf{x}$  again at time  $t+1$ . Since the computer experiences an infinite amount of time before the hole appears, it experiences no limitations as to how long it can wait before ejecting the buoy. Meanwhile, having started the program running, the observer travels around the hole and also arrives at  $(\mathbf{x}, t+1)$ . If the observer finds a signal buoy there he knows that the program eventually stopped running, whereas if no buoy is present the program ran forever.

In other words, by using the circumventable edge the observer has managed to solve the Halting Problem. Given *any* program  $P$  the observer can decide whether or not that program eventually terminates. Since the Halting Problem is well known not to be solvable using a standard computer, this implies that Malament-Hogarth spacetimes allow hypercomputational behaviours. It is less clear whether these spacetimes allow *deliberate* construction of such behaviours, because the algorithm we've described relies on the observer knowing beforehand that the hole is going to

---

appear at  $x$ . This is not necessarily impossible. There may exist some predictable procedure in a Malament-Hogarth spacetime that would trigger the creation of just such a hole, just as massive stars can be predicted to trigger black holes when they eventually collapse at some point in their futures. Consequently, while the feasibility of Malament-Hogarth spacetimes entails [HC], it is unclear whether it also entails [ST].

While this is interesting in its own right, it is also interesting for another reason. Most attempts to describe non-computational physical systems concentrate on the manipulation of devices. For example, we might consider the effects of using quantum-bits or analog machinery in place of standard (fully identifiable) bit values and the discrete-time execution steps of standard software. But the relativistic algorithm described above leaves Turing's design unchanged. Hypercomputational behaviour arises not through the introduction of more complex devices, but by allowing a standard computational device to be observed in an unusual way. This undermines those constructivists who consider recursion to give an accurate and essentially complete description of whether a function can or can't be evaluated, because the 'extra' computational functions in Malament-Hogarth spacetime are described using exactly the same recursive definitions as 'standard' computation. The extra power of Malament-Hogarth computation comes not from any enhancement of the underlying mathematical theory, but through a change in the theory's semantic model.

#### QUANTUM MACHINES VS QUANTUM COMPUTERS

Quantum computation (Benioff, 1980; Deutsch, 1985; Feynman, 1986; Shor, 1997; Steane, 1998) is seen by many in computer science and beyond as a likely basis for the next major development in computer power, and the device described here is essentially quantum computational. Deutsch and Penrose<sup>14</sup> have argued that quantum computation cannot achieve non-computable results, but only has the potential to produce computable results more quickly than standard computers. When one moves to the physical domain, this claim is no longer valid; I contend that one can implement behaviours in a quantum universe that are not implementable classically. In other words, even though quantum *computers* are equivalent to classical computers, quantum *machines* are not equivalent to classical machines.

This can be seen as evidence against the feasibility of Turing computation itself in a classical universe. The device I describe below is certainly computational — it implements a behaviour that is simulable by a nondeterministic Turing machine, but its behaviour cannot necessarily be considered to exist in a classical universe. In other words, there potentially exists a Turing-computational behaviour that cannot meaningfully be considered to be implementable.

I need to establish two results:

- first, that any classical implementation of a computable system is deterministic;
- second, that a quantum device can be implemented which is non-deterministic.

I contend that the following quantum-device implements a behaviour that isn't implementable classically. The device is essentially just a quantum mechanical coin-tossing machine, and one might ask why tossing a standard coin wouldn't establish the same results. The answer lies in the nature of randomness. When examples are given of randomness based on coin tossing, one can never be certain that they really are random, as opposed to merely under-informed. Following standard Newtonian analysis, if we knew the relevant details of the coin's position and the way in

---

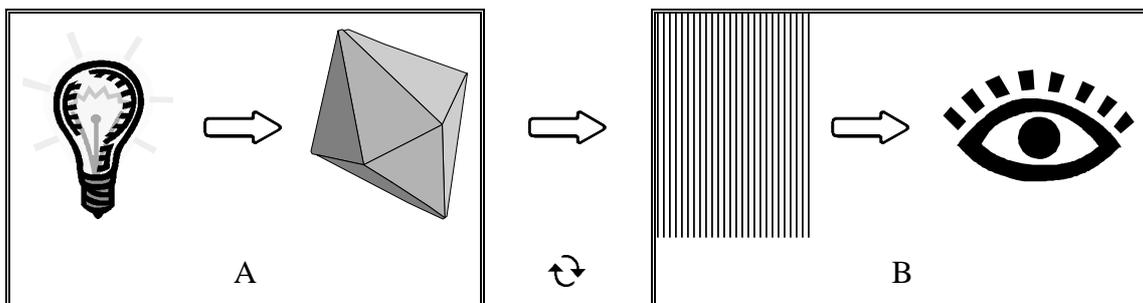
<sup>14</sup> Penrose makes his remarks in his introduction to (Deutsch, 1985).

---

which is launched, we presumably could, and that with certainty, establish ahead of time what the outcome of the toss would be (there are, no doubt, Las Vegas regulars capable of tossing fair coins ‘randomly’ to generate any pre-specified pattern of heads and tails). In a fundamental sense, then, coin tossing is entirely deterministic, and can hardly be described as random at all. For the same reason, it is not meaningful to talk of non-determinism with respect to physical implementations of Turing machines. Theoretically, of course, there is no problem; one can easily consider Turing machines with added non-determinism, added oracles, and so on; but we are interested in *physically relevant* systems – just because a non-deterministic machine can be described, it does not follow that it can be built, and this is what needs to be established.

As in *Section 3*, my proposed device uses quantum behaviours to generate provably random integers, but in this case, the output is always generated after a pre-specified period. Neither the design nor the operating manual for the device makes any reference to quantum theory, and all user-manipulations of the device are purely classical (the user essentially flips a switch to run the device). The role of quantum theory does not lie in the construction of the system, but in ensuring the essential randomness of its output. True randomness cannot safely be asserted of any classical system, because an apparently random system (for example, the tossed coin) is classically regarded as one whose description is simply incomplete. From the classical viewpoint, we are free to refine our description until it contains enough information for *a priori* forecasts of system behaviour to be made. In contrast, randomness is an inevitable consequence of quantum uncertainty and forecasting the behaviour of our device is quantum theoretically impossible, even in principle.

The device uses two standard components arranged in sequence. The first, (A), generates circularly polarised photons (for definiteness, they are always polarised clockwise). The second, (B), determines whether or not photons have a specific linear polarisation (for definiteness we always check for vertical polarisation). If dedicated components are not to hand we can make (A) by passing low intensity light through some appropriately cut calcite crystal. The low intensity ensures that photons can be considered to be generated one at a time, and the calcite imparts circular polarisation. To make (B) we can place a light amplifier behind a vertical grating (or a second piece of calcite, this time cut for vertical polarisation).



Operating the device is also simple. We first calibrate the system by calculating the mean time,  $t$ , between photons being generated at A. If necessary, we adjust the intensity of our light source to ensure that  $t$  is long enough to let us complete the measurement process described next. To generate a random bit we switch on both the source at A and the detector at B for  $t$  seconds. If a photon is detected at B during this period the system has generated a **1**, and otherwise a **0**. Clearly, this system is guaranteed to generate either a **0** or a **1** after  $t$  seconds, and by repeating the process, we can generate bit strings of arbitrary length. To generate a random number between 0 and  $(2^n - 1)$  we would run the device  $n$  times to generate the value as an  $n$ -bit binary value.

The output generated at B is truly random, and **0** and **1** each have approximately a 50% chance of being generated during each run. The reason for this is well known (*see e.g. Feynman et al, 1965*), but we include it for completeness. Consider what happens during a single run of the process. Because the run lasts for  $t$  seconds, we would expect, on average, precisely one photon to be

generated at A. This leaves A in a circular polarisation state  $|\curvearrowright\rangle$  (clockwise). This polarisation state can be re-expressed in normalised terms relative to a basis of vertical ( $\uparrow$ ) and horizontal ( $\rightarrow$ ) linear polarisation states as  $|\curvearrowright\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle + |\rightarrow\rangle)$ . So the probability of acceptance at D is  $(\frac{1}{\sqrt{2}})^2 = \frac{1}{2}$ .

In practice our calibration value  $t$  will be inaccurate, say  $t = (1 + \varepsilon)t_m$ , where  $t_m$  is the true mean. This would cause, on average,  $\varepsilon$  extra photons to pass through the device on each run. Every  $1/\varepsilon$  runs, then, we expect two photons to pass through the device during the same run, and this increases the likelihood of a **1** being recorded on that run from  $\frac{1}{2}$  to  $\frac{3}{4}$ , because we only need one of the two photons to be transmitted by the grating at B. Provided  $|\varepsilon|$  is small we can ignore the possibility of 3 or more photons passing through the device during the same run. To first order, then, the actual probability of the system generating a **1** during any run is  $(0.5 + \varepsilon/4)$ . By tuning our calibration techniques this probability can easily be bounded away from both 0 and 1 (so the behaviour remains random, albeit with slightly offset mean) and can be made to approach a true 50/50 split by driving down the systematic error  $|\varepsilon|$  (or by improving our control over the intensity of the generating light source).

In principle, there is no reason why devices of this, or some essentially equivalent, construction could not be included in the physical computing systems in use today. Calcite is essentially stable at standard temperatures, low intensity light sources would cause little drain of system resources, and such detectors typically generate their output electrically as a matter of course. Provided low-power detectors can be installed, the system would not compromise the power provisions of a standard PC.

## 5 The indistinguishability of hypercomputation

Over the last decade I have heard it asserted on many occasions that non-computable values cannot have any physical meaning, for no matter how carefully you measure a length (say), the result will always be indistinguishable from a sufficiently close computable value, and values which cannot be distinguished experimentally cannot be assigned any physical interpretation. This argument seems at first sight to be irrefutable, but it can be reframed in a way that shows it to be rather more questionable. If we replace the words ‘computable’ with ‘rational’ and ‘non-computable’ with ‘irrational,’ we obtain the following assertion:

~~non-computable~~ *irrational* values have no physical relevance, for no matter how carefully you measure a length (say), the result will always be indistinguishable from a sufficiently close ~~computable~~ *rational* value.

This statement has precisely the same logical framework as the original, and so ought to be true if and only if the original is true. However, can we really argue that irrational numbers are physically meaningless, when the irrational value  $\sqrt{2}$  can be constructed physically as the diagonal of a unit square? Surprisingly, the status of irrationals was indeed the subject of debate in the years before computability. Writing in 1923, Hobson comments:

It has been charged against Mathematicians that, in setting up such a scheme as the arithmetic continuum, they have introduced an unnecessary complication, in view of the fact that rational numbers suffice for the representation, to any required degree of approximation, of all sensibly continuous actual magnitudes; that in fact an instrument has been created of an unnecessary degree of fineness for the purposes to which it is to be applied. The answer to the charge is that Mathematical Analysis, which is based upon the arithmetic continuum, and essentially consists of operations involving numbers, would become unworkable as a conceptual scheme, or at least much more cumbrous, if the conception of irrational numbers were excluded from it. The results of operations involving rational numbers constantly lead to irrational numbers, without which the operations would be impossible if their effects are to be regarded as definite. But in order to

appreciate the full weight of this answer it is necessary to consider the great generalization of Arithmetic which is made when variables are introduced which denote unspecified numbers. The passage is then made from the primitive form of Arithmetic to Algebra, in which the formal operations of Arithmetic are represented as relations between sets of unspecified numbers represented by non-numerical symbols. The result of an algebraic operation, expressed by general formulae, such for instance as the simple case of the solution of a quadratic equation, would not always be interpretable when special numerical values are assigned to the symbols, if the only admissible numbers were rational ones.

Without the employment of the conception of irrational numbers the functions of Mathematical Analysis would be degraded to that of determining only approximate results of operations it employs, and in consequence its technique would have indefinitely greater complication, of such a character that, at least in its more abstruse operations, it would break down, or lead to results which contained a margin of error difficult to estimate.

I find Hobson's defence of irrationals unconvincing, since it is unclear why a desire on the part of mathematicians to make their lives both easier and more interesting should be construed as a reason why Nature should have any use for irrational numbers. It might be argued, for example, that such a thing as a unit square is itself unconstructible, since it is impossible in practice to construct exact squares, so that no 'true' diagonal of length  $\sqrt{2}$  can in fact exist. Taking objections of this kind into account, might it perhaps be possible that all 'true' distances are actually rational multiples of some fixed unit, and that the irrational values we posit have no 'real' existence? If so, this would require an overhaul of standard ways of thinking, because it would no longer be meaningful to consider 'continuous' motion in the usual sense – if irrational distances have no physical meaning, then at no point during a journey from 'here' to 'there' would we ever have covered an irrational distance, and likewise when a balloon rises from sea-level to 1000 metres, it would not occupy all heights in-between, but only the rational heights. This view of the world is at odds with modern academic preferences, but such a model is not 'obviously' impossible<sup>15</sup>, and its elucidation might well prove sufficiently mathematically challenging even for algebraists.

I contend that the argument "non-computable numbers cannot be observed and so need not be considered" is not an argument against hypercomputation at all, but about the role of approximation in the measurement process. This becomes clear if we consider that the 'unobservables are irrelevant' argument actually applies to rational numbers as well. Just as we can approximate  $\sqrt{2}$  with ever closer rationals, so we can approximate the value  $\frac{1}{2}$  (say) with rationals from the sequence  $\frac{1}{2} - (\frac{1}{2})^n$ , and any argument to the effect that " $\sqrt{2}$  cannot be distinguished from sufficiently close approximations" applies with equal force to the rational value  $\frac{1}{2}$ . If indistinguishability renders  $\sqrt{2}$  meaningless, it also renders all rational numbers meaningless as well.<sup>16</sup>

Furthermore, the measurement of distinguishable values has very little to do with physical processing. What matters in physical computation is not always whether an output can be measured, but whether it can be used as the input to another process. One can, for example, generate an analog model of blood flow by connecting together various fluid-filled pipes, valves and pumps, and one can also build a machine that responds to the pressure of fluid in a pipe by adjusting the height of mercury in a vertical tube, thereby causing a continuously varying electrical resistance along the length of the tube. If I connect this pressure-device to one of the pipes in my blood-flow model, I

---

<sup>15</sup> The mathematical discipline 'general topology' often concerns itself with describing and analysing the nature of continuity in spaces that aren't "continuous" in any everyday sense of the word; there is nothing intrinsically nonsensical about the idea.

<sup>16</sup> The only way to obviate this argument seems to be to assume that all numbers are 'isolated,' *i.e.* cannot be approximated by sequences of other numbers. This effectively means restricting attention to integers, and banning all other numbers from consideration.

---

have effectively used blood pressure as an output from the flow-model, and supplied it as an input to the pressure device, but at no point have I attempted to generate an accurate measurement of the pressure itself. I could, no doubt, have measured the pressure in the tube, and then supplied details of my measurement to an input driver on the pressure device, but this would have entailed expressing the values as recursive decimal fractions, with the introduction of wholly avoidable measurement inaccuracies. By connecting the two analog systems directly, I allow accurate transmission of the required value from one system to the other, in a way that is impossible if I enforce a policy that all inputs must be measurable. Measurement, I suggest, has little to do with computation, and everything to do with our desire to control things, for this is surely our main reason for interfering in processes that can operate more accurately if they are left to themselves. This has been made especially clear by recent successes in practical quantum teleportation, where an entity at  $A$  comes to reside at  $B$  without moving bodily between the two locations (Bennett *et al.*, 1993; Bennett, 1995; Steane, 1998). This manipulation, essentially a physical representation of the basic assignment  $B = A$ , specifically requires that the data supplied for processing (the inputs) are *not* observable, since observation would scramble the required quantum information and prevent reconstruction at  $B$ . It may seem strange to declare a value computable only if it is not observable, but this merely confirms the shortcomings of standard terminology when dealing with non-classical modes of computation. From a physical point of view, it is (literally) no different to declaring a superposition of quantum states to be a valid description of a system only so long as the system remains unobserved.

Nonetheless, it is a fact that many computational systems today are essentially Newtonian and involve measurement, so it is instructive to consider why the value  $\sqrt{2}$  is meaningful even in Newtonian measurement-based systems. Analysis of this question throws light on the question why hypercomputational values can also be meaningful under these circumstances. There are two questions to consider, corresponding to the two different ways in which  $\sqrt{2}$  can be assigned physical and computational meaning.

The value  $\sqrt{2}$  is normally said to be computable because one can calculate any digit in its decimal expansion in a finite amount of time, but this explanation is unsatisfactory from a physical point of view because the decimal expansion *as a whole* obviously cannot be completed in finite time. A more finitistic rationale behind  $\sqrt{2}$ 's computability would be the observation that we can regard  $\sqrt{2}$  as a function  $root2: \mathbb{N} \rightarrow \{0,1\}$  mapping each  $n$  to the  $n$ 'th digit of its binary expansion. Given any choice of  $n$ , we can use standard numerical approximation methods to evaluate the associated digit,  $root2(n)$ , in finite time, so we can say that  $\sqrt{2}$  is computable *as a function*. This is accurate, but not particularly satisfying, since one does not normally think of numbers as functions, and it is not easy to see how we can associate this view of numbers with the measurement process.

What we really need is an explanation why  $\sqrt{2}$  is meaningful *as a measurement*. As the 'indistinguishability' argument suggests, physical measurements are rarely exact, but this is not so much a problem as a feature. Exact measurement is not normally achievable, but this is irrelevant because exactness isn't the goal of measurement in the first place. As with all physical processes increased information entails increased cost, and the goal of measurement is only ever to establish the magnitude of some quantity without entailing excessive charges. It is undoubtedly possible to generate finer tools than are currently available, but only if we're willing to pay for their development. By common agreement, we avoid these charges by accepting national and international standards of weights and measures, and by agreeing to accept that all measurements can only be as accurate as is permitted by the standard measuring rods at our disposal. Thus, for example, it is meaningless to give the length of an object in metres to one hundred decimal places,

because the metre itself cannot be physically replicated to this accuracy.<sup>17</sup> This, however, may only be the *current* state of affairs, and we cannot say with certainty that more accurate measurement procedures will not be developed in the future; all we can safely say is that measurements will probably never be infinitely accurate. Relative to this background of ever increasing, but always finite, accuracy, we can say that a process for computing magnitude is meaningful if it is able to provide the magnitude in question to whatever accuracy may be required at the time. In other words, as far as decimal measurement is concerned, a *value* is really a function  $v(n)$  that generates, for each  $n$ , a number that is within  $10^{-n}$  of *value*. There may be many such approximating functions, and the *value* is computable provided at least one of these functions is computable. In this respect, the magnitude  $\sqrt{2}$  is certainly computable, because a standard numerical approximation algorithm can be used to evaluate  $\sqrt{2}$  to any number of decimal places.

It is also useful to ask what we mean by decimal expansions in the first place, because these seem to lie halfway between the discrete world of programs and the continuous world of real numbers. Such an expansion is simply a stream,  $\mathbf{x}$ , of digits, which acquires meaning when we supply those digits to an algorithm. For example, decimal expansions of the form  $\mathbf{x} \equiv (x_1, x_2, x_3, \dots)$  acquire meaning as values in the range  $0 \leq x \leq 1$  by being regarded as input streams to the program

```

DECIMAL( $\mathbf{x}$ ) = {
   $x = 0$ ;  $n = 1$ ;
  while (true) {
     $x = x + x_n/10^n$ ;
     $n = n+1$ ;
  }
}

```

This program will terminate if and only if the stream  $\mathbf{x}$  is finite, or can be shown to comprise only 0's from some point onwards. For a number like  $\sqrt{2}$ , these conditions cannot be met, so the ongoing value of  $x$  is only ever an estimate of the true value.

However, there are many ways to represent numbers, and no obvious reason why the standard *DECIMAL* algorithm should be considered more appropriate than any other. It is well known that a real number is rational if and only if it has a repeating decimal expansion. In general, if  $A$  is some algorithm for calculating real numbers and  $\mathbf{x}$  a stream of symbols to be supplied as inputs for  $A$ , we can argue that the number  $A(\mathbf{x})$  is 'rational' with respect to  $A$  if and only if the representation  $\mathbf{x}$  eventually starts repeating. For example, we might choose to represent values as continued fractions, relative to which the notation  $x = [abc\dots]$  means that  $x$  can be represented as the output of the algorithm

$$x = a + \frac{1}{b + \frac{1}{c + \dots}}$$

Relative to this representation  $\sqrt{2}$  can no longer be considered 'irrational,' as it has the simple repeating expansion  $\sqrt{2} = [1,2,2,2,2,2,\dots]$ . Recurring expressions of this kind can be written in the finite form  $\sqrt{2} = [1;2]$ , where the digits following the semicolon represent the recurring sequence. This form of finite numerical representation is actually used for physical computational purposes<sup>18</sup>, so has clear physical meaning. Even transcendental numbers can have finitely expressed expansions

---

<sup>17</sup> Even fifteen decimal digits assumes that lengths can be known with sub-atomic accuracy (atomic nuclei have diameters in the region of  $10^{-15} m$ ).

<sup>18</sup> Gowland and Lester give a useful summary of the use of continued fractions, and other representational schemes, in their "Survey of Exact Computer Arithmetic," in (Blanck *et al*, 2000).

---

as continued fractions, for example  $e = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, \dots] = [2; 1, 2n+2, 1]$  (Vuillemin, 1987).

Before, when I objected to the argument that  $\sqrt{2}$  could be considered computable because we can calculate its decimal expansion, I did so on the grounds that the expansion  $\mathbf{x}$  could not be expressed as a finite string of digits, whence the algorithm *DECIMAL*( $\mathbf{x}$ ) could not terminate in finite time with the correct value of  $x$ . It's clear, however, that there may be other algorithms,  $A$ , for which the same number *can* be expressed as a finite stream of symbols. For example, the values  $\sqrt{2}$  and  $e$ , both of which involve non-terminating executions of *DECIMAL*, can be expressed as finite strings relative to the continued-fraction algorithm. Likewise, one can argue that " $\sqrt{2}$ " itself, or the logical description " $(x^2 - 2) = 0 \ \& \ x \geq 0$ ," are equally valid descriptions of  $\sqrt{2}$  which again use only finitely many symbols. Taking this into account, we can argue that a value is computable *as a number* (rather than as a function) if there is at least one algorithm  $A$  relative to which the value's representation is a *finite* character string. Perhaps surprisingly, it's easy to show there exists a general-purpose representational algorithm *ITER* relative to which *every* real number that is computable-as-a-function can be expressed as an *integer* with respect to *ITER*.<sup>19</sup> So any value that is computable-as-a-function is also computable-as-a-number.

Is the converse also true? If  $A$  is an algorithm and  $\mathbf{x}$  a finite string of symbols that represents a real number  $x = A(\mathbf{x})$  relative to  $A$ , does it follow that, given any  $n$ , we can compute the  $n$ 'th digit in the decimal expansion of  $x$  in finite time? The first thing to notice is that we can ignore  $\mathbf{x}$  altogether, because we can easily write a program that generates the symbols in  $\mathbf{x}$  from scratch. So we may as well assume that  $A$  is a program that takes no inputs, and that  $A \downarrow x$ . In other words, our definition of *computable-as-a-number* is the most obvious one possible — a number is computable provided there is a program that computes it. Surprisingly, this is a strictly more general concept than the computable-as-a-function definition normally used.

**THEOREM.** There exists a value  $\sigma$  which is computable-as-a-number but not computable-as-a-function.

**SUMMARY OF PROOF.** I'll generate  $\sigma$  in such a way that *individual digits* take infinitely long to compute (in other words, although  $\sigma$  can be computed as a real number, it cannot be represented relative to any integer base). For convenience I'll assume that the digits required are those of the binary, rather than decimal, expansion of  $\sigma$ . First, arrange all possible programs in a list  $P_1, P_2, \dots$  (this is known to be possible), and then declare  $\sigma$  to be the number with  $\sigma_n = 1$  if  $P_n$  halts, and  $\sigma_n = 0$  if it doesn't. Because the Halting Problem is well known to be undecidable, there is no computable function  $\xi: \mathbb{N} \rightarrow \{0,1\}$  which can generate the binary expansion of this particular  $\sigma$ . Nonetheless, there is a program that computes  $\sigma$  — I have included an example of such a program in the *Appendix*. ■

I contend that values like  $\sigma$  are as meaningful in their own terms as recurring decimal expansions are in theirs. When we see the recurring decimal representation  $\frac{1}{3} = 0.\dot{3}$ , we recognise this finite string of digits as the input to an algorithm that takes infinitely long to generate its result. But intuitively speaking, we don't classify it as uncomputable, because we *understand* the nature of its

---

<sup>19</sup> For suppose  $x$  is any value that is computable-as-a-function. That is, the function  $\xi: \mathbb{N} \rightarrow \{0,1\}$ , which gives the  $n$ 'th digit in the decimal expansion of  $x$ , is a computable function. Let  $[\xi]$  be the index of  $\xi$  in some Gödel numbering of programs (that is,  $\xi$  is the  $[\xi]$ 'th program in some list of all programs), and write  $P_m$  for the  $m$ 'th program in this numbering. Let *ITER* be the program  $ITER(m) \equiv \{ \text{result}=0; n=1; \text{while}(\text{true}) \{ \text{result} = \text{result} + P_m(n)/10^n; n = n+1; \} \}$ . Then  $x = ITER([\xi])$ , so  $x$  can actually be represented as the integer  $[\xi]$  with respect to the representational algorithm *ITER*.

---

representation. The algorithm may technically require infinitely long to run, but the fact that the representation is finite means we can interpret the value, and establish its relationships to other values, without actually performing the infinitely long computation. Likewise, the recurring repeated fraction  $\sqrt{2} = [1;2]$  is a finite string of digits for use with an algorithm that takes infinitely long to generate its true output, but the nature of the representation is well enough understood that comparisons can be made with other values, even without evaluating the entire continued fraction. In other words, provided we can represent a value as a finite string of symbols, we can regard it as meaningful, because we can perform an ‘abstraction’ over the amount of time required for the answer to be generated (*i.e.*, we choose to regard the time required as insignificant to our discussion). Suppose that  $U$  is a universal program, so that  $U(m)$  simulates  $P_m()$  for each program  $P_m$ . Since  $A$  is a program there is some  $a$  for which  $A = P_a$ , and now  $U(a) = \sigma$ . So  $\sigma$  can be regarded as the output obtained when the finite string of digits  $a$  is supplied as input to the program  $U$ . Because  $a$  (which is an integer) is a *finite* representation of  $\sigma$ , it can meaningfully be considered a *bona fide* value, relative to the basic abstraction described above, so it is appropriate to use  $\sigma$  as an input to calculations and comparisons. There is literally no difference between the way in which  $\frac{1}{3}$  and  $\sqrt{2}$  are generated with respect to their algorithms, and the way in which  $\sigma$  is generated with respect to  $U(a)$ .

To summarise,  $\sigma$  is a well-defined value that has the same status as a potential input to calculations and comparisons as values like  $\frac{1}{3}$  and  $\sqrt{2}$ . However,  $\sigma$  is a *hypercomputational value*, because the standard definition of what it means for a value to be computable is not satisfied. There is *no* computable function  $\xi$  for which  $\xi(n)$  evaluates to the  $n$ 'th digit of  $\sigma$  in finite time.

Finally, what about our original observation that  $\sqrt{2}$  must be implementable, because we can construct it as the diagonal of a unit square? At this point implementation and computability seem to part company, because there seems to be no better proof that a value is implementable than a demonstration of its construction. In contrast, recursive number theory (see the Appendix) only accepts those functions for which a calculation-based definition can be given. Given the logico-mathematical origins of recursion theory this focus on calculation is entirely understandable, but it is nonetheless inappropriate when studying implementations. We have already seen that some algorithms generate values of interest only if they are allowed to run forever, and this immediately throws their physical relevance into doubt. Why should it be considered preferable to perform an infinitely long calculation in order to generate a physical representation of  $\sqrt{2}$ , when an alternative, and *finite*, procedure is available which simply involves the construction of a square and the drawing of its diagonal? There may be sound mathematical reasons for preferring calculation to demonstration, but the feasibility of hypercomputation is never going to be resolved on purely mathematics grounds; it is also a physical question, and physical considerations should be involved. The whole point of the question ‘is hypercomputation feasible’ is that this feasibility must be *demonstrated*, and this cannot happen if we are required to use only those techniques that violate the very basis of physical demonstration itself. You could easily argue that “ $\sqrt{2}$  can't be constructed in the real world because I refuse to allow you to use anything but an infinitely long construction process” but this says nothing about  $\sqrt{2}$  or the physical world; it says only that you are imposing unrealistic and unwarranted constraints that have nothing to do with physics and everything to do with arbitrary mathematical assumptions. That is not to say that mathematics cannot inform the debate, and cannot be used to characterise many of the processes inherent in implementation. There may well be values for which no direct construction is forthcoming, and in that case, we may need to fall back on mathematical constructions to supplement the techniques at our disposal. Moreover, it is worth knowing that certain numbers are computable, because this gives us good grounds for at

least considering that they might also be implementable. Without mathematical guidance of this kind, there may be little to assist us in the search for implementable values.

What of our objection that unit squares are not in fact constructible? To answer this, we need to consider what we mean when we associate values with physical constructions. I claim that the statement “this line has length  $\sqrt{2}$ ” has exactly the same degree of meaning as a statement of the form “this pile contains six objects.” Mathematical definitions of counting give pride of place to the notion that we can tell if two collections have the same number of members by setting up a one-to-one correspondence between those members; the argument then has it that counting is nothing more than placing the members of an observed collection into one-to-one correspondence with a set of (recursively constructed) standard sets<sup>20</sup>. According to this argument, I ‘know’ that there are precisely four books in front of me because I have intuitively constructed in my mind a one-to-one correspondence between the collection “books in front of me” and the members of the set  $\{\{\},\{\{\}\},\{\{\},\{\{\}\}\},\{\{\},\{\{\}\},\{\{\},\{\{\}\}\}\}$ . If this is really the case, then all I can say is that my intuition is a far finer mathematician than I am!

A more reasonable explanation is suggested by Goodstein (1964, pp. 8-9), who argues that

the process of counting consists in a transformation from one number notation to another by means of the rules “one and one is two,” “two and one is three,” “three and one is four” and so on. It is the recitation of these rules (in an abbreviated form in which each “and one” is omitted, or replaced by pointing to, or touching, the object counted) which gives rise to the illusion that in counting we are associating a number with each of the elements counted, whereas we are in fact making a translation from the notion in which the number signs are “one,” “one and one,” “one and one and one,” and so on, to the notation in which the signs are “one,” “two,” “three,” and so on ... [C]ounting does not discover the number of a collection but *transforms the numeral which the collection itself instances* from one notation to another. To say that any collection has a number sign is just to say that any collection may be used as a number sign.

By extension, when we say that a given construction yields a line two metres long (say), we do not mean that a measurement of the line would reveal it to be exactly twice the length of a standard metre – we have already seen that such exactness is not in the purview of measurement – but that the line can be used as a *number sign* representing the value 2, as part of the same numerical system in which the standard metre is taken to be a number sign for the value 1. Likewise, when I construct the diagonal of a unit square, and say that it has length  $\sqrt{2}$ , I am really saying that, as part of the number system in which the sides of the figure I have drawn may be taken to be number signs for the value 1, the diagonal may itself be taken to be number sign, one for which the value represented is  $\sqrt{2}$ . Likewise, hypercomputational values can be used as meaningful descriptions of physical extents. because, when we say that a given construction generates a hypercomputational length, we are not suggesting that exact measurement of the length either can or should take place, but that, from the point of view of that system in which our unit length itself acquires meaning, the constructed length is a number sign for a non-computable value.

## 6 Summary

We have touched on many issues in this paper, but our focus has been the relationship between physical and conceptual computability. Hypercomputation concerns the existence of physical systems whose behaviours don’t possess certain mathematical properties; as such, the status of hypercomputation depends on the mathematical model of physics we hold to be valid.

---

<sup>20</sup> The standard collection with no elements is the empty set  $\{\}$ . The standard collection with  $n+1$  elements is the set  $\{0, \dots, n\}$ . Thus, the standard set with one element is  $\{\{\}\}$ , the standard set with two elements is  $\{\{\}, \{\{\}\}\}$ , and so on.

---

The standard definitions of what it means for a function to be computable<sup>21</sup> implicitly assume a special type of universe in which unlimited portions of space and time can be accessed in a controlled way; loosely speaking they are Newtonian universes. Moreover, the question whether a value is computable is, arguably, largely syntactic, for as the worked example in the *Appendix* shows, a proof whether a given function is primitive recursive can be generated by pattern matching. In other words, there seem to be good grounds for considering recursion (and definite grounds for considering primitive recursion) to be a purely syntactic property of sentences. The fact that standard systems can generate non-standard results when placed in a Malament-Hogarth spacetime suggests that this need not always be the case. The fact that adding a simple notification system (the signal buoy) to a universal program effectively enables Turing machines to solve their own Halting Problem indicates that some aspects of computability need to be defined not syntactically but semantically.<sup>22</sup> Is it possible, given any model of physics, to define general notions of syntactic and semantic computability? If so, does the relationship between the two types of computability mirror other mathematical features of physical models (for example, curvature<sup>23</sup>)?

Just as relativistic systems allow Turing machines to solve their Halting Problem, we've seen that quantum theory allows the construction of a number generator whose behaviour cannot be distinguished experimentally from that of a true random number generator. We've also seen that the relationship between implementation and computation is different in different physical models: Deutsch's Universal Quantum Computer (Deutsch 1985) is formally equivalent in power to a Universal Turing Machine, but it is possible to design a quantum theoretical *machine* whose behaviour cannot meaningfully be implemented under Newtonian constraints.

We've also considered what it means to say that a physical property, a length say, has hypercomputational extent. I've argued that the status of such claims is the same, at a reasonably deep level, as that of the claim that rational numbers like  $\frac{1}{2}$  and  $\frac{1}{4}$  are meaningful. I've also argued that computation doesn't require measurement to take place. In the 19<sup>th</sup> century, a set was a collection of objects; the axiomatization problems of the early 20<sup>th</sup> century changed this concept fundamentally — we now recognise that a key feature of sets is that they *belong* to other sets. The nature of computation may also need to be reinterpreted: what matters about computations is not that they generate distinguishable (that is, observable or measurable) outputs, but that these outputs can be used as inputs to subsequent operations. I've argued that a value  $\sigma$  can be defined which is as meaningful as an input as values like  $\frac{1}{3}$  and  $\sqrt{2}$ , but which is technically hypercomputational.

There are also many questions I have not discussed here. In particular, I have not addressed Siegelmann's extensive corpus of work on the hypercomputational properties of analog recursive neural networks (see *e.g.* Siegelmann 1999). I have also avoided resurrecting issues discussed in my own earlier work, describing purely mathematical functions with hypercomputational flavours (Stannett 1991). An important area about which very little seems to be known is the relationship between hypercomputation and undecidability. We know from Gödel's work on undecidability that any theory powerful enough to describe recursive arithmetic must be incomplete. Since physical models of arithmetic can be constructed — it can be argued that arithmetic arose by abstraction of physical behaviours in the first place — this suggests that every model of physics is likely to be incomplete, and that every version of physics includes at least one system whose behaviour cannot

---

<sup>21</sup> For our purposes the terms *partial recursive* and *computable* are interchangeable; see the *Appendix*.

<sup>22</sup> It is reasonable to assert that semantic computation is precisely what we have been referring to as implementation.

<sup>23</sup> Does curvature have any relationship to computability? It seems unlikely, but the hypercomputation described above for Malament-Hogarth spacetimes clearly depends on their having non-Euclidean topology. Were they to be completely flat, the required behaviour at circumventable edges would presumably be unavailable.

---

be deduced or explained from physical laws. Such behaviours seem to be inherently hypercomputational, so we seem to be saying that hypercomputation is not merely feasible, but *necessary*. It is also interesting that systems with undecidable properties can be defined relatively easily. A particularly good example of this is Dornheim's work on plane polygonal mereotopology (Dornheim 1998); rather than focus on the ambiguous notion of general physical systems, Dornheim considers a system in which the objects of interest are groups of finitely many polygons lying in a single plane. Mereotopology is a branch of topology that studies the concept of inclusion; in this case the relevant notion is that some polygons can be considered to be 'part of' other polygons. Using just these basic building blocks, Dornheim shows that statements can be made about polygons that are undecidable. Given the simplicity of this system, and the likelihood that 'polygons' can be defined meaningfully in most models of physics, this seems to confirm the contention that hypercomputation is not merely feasible, but a necessary fact of life.

## Appendix: Basic computability theory

### BASICS

*Computability* refers to the idea that some functions can be calculated by programs running on formal computers. For the purposes of this paper, the terms "computable" and "partial recursive" are synonymous. It is important to understand that *computability* (or *recursion theory*) presents a very different theory of functions to that assumed in standard mathematics. Mathematically, a function is defined to be a set of pairs, so that, for example, the function  $square(n) = n^2$  is *defined* to be the infinite set  $square = \{ (n, n^2) \mid n \in \mathbb{N} \} = \{ (0,0), (1,1), (2,4), (3,9), \dots \}$ . Theoretical computer science rejects this definition as unworkable in practice, because it requires the construction and manipulation of infinite objects. Instead, a function is a *rule* or *process* by which a collection of arguments is manipulated in such a way as to *evaluate* a result. Sometimes the evaluation process fails to generate an answer, and in that case, we say that the function is *undefined* (for that particular collection of arguments). The processes by which functions are evaluated are called computer programs. Those (mathematical) functions that can be evaluated in the computer-science sense are called *computable* functions; those that can't are *uncomputable*.

In traditional recursion theory the computers in question are what I call *unbounded-discrete-space*, *unbounded-discrete-time*, *classical computers*. This means:

- The space used for storing values (*i.e.* their *memory*) is divided into discrete chunks whose dimensions in each direction are always greater than some positive (*i.e.*, necessarily non-zero) minimum length. They store one bit of information in each chunk. This means that computers can't store an infinite amount of information in a finite volume of space. The number of chunks available to the computer is unbounded.
- The time used for executing instructions is divided into discrete chunks whose length is always greater than some positive minimum length. Processing proceeds in discrete steps, one per chunk of time. This simply means that computers can't perform infinitely many instructions in finite time. The amount of time available for processing is unbounded.
- The Universe in which they operate is *classical* (this is a technical term meaning that the physics in question, and in particular the nature of causality, is of the pre-quantum physics kind). This means that the outcome of each definite operation is itself definite – we don't have to worry about quantum information effects.

These computers are *idealised* in the sense that they have access to as much memory as they need, and if an answer exists, they can take as long as needed to generate that answer. However, even though the time and memory resources available to the computer are essentially unlimited, the total

---

amount of memory used at any given time must always be finite. Whether such computers can ever exist ‘for real’ is a moot point. In principle, a standard PC satisfies the requirements if it can be supplied with as much extra memory as it needs (*e.g.* by providing an ever-increasing supply of re-writable CD-ROMs or floppy disks) – the unbounded-space assumption ensures an indefinite supply of usable memory locations. If the Universe is destined to end in a ‘Big Crunch,’ then the amount of space available is limited, and programs presumably won’t be capable of running for arbitrarily long periods of time – the unbounded-space and -time assumptions avoids this complication. We simply assume that sufficient memory can be provided and that a program can be left running for as long as needed, and ask what functions this allows us to compute.

Given these assumptions there are many obviously computable functions. For example, *addition* can easily be handled by a spreadsheet program, so addition is computable. This definition is easy to understand, but has the disadvantage that it’s not at all obvious how to identify or provide a formal definition of those functions which are, and those which are not, computable.

We can describe the set of computable functions by considering how a function,  $f$ , is evaluated by a computer. We obviously can’t provide the computer with a complete list of all the values taken by  $f(n)$  as  $n$  takes every possible value, because that would require using an infinite amount of memory. Consequently, rather than telling the computer what the answers *are*, we have to tell it how to *calculate* those answers.

Before we can do this, we have to decide which functions are ‘built-into’ the computer. A program is a scheme for translating complicated processes (like “calculate January’s profits”) into patterns of simpler processes (like “start by setting (the value stored in the spatial chunk called) *total* to 0; for each  $n = 1 .. 31$ , calculate *profitForDay(n)* and add that result to *total*”). This downward-translation process has to be grounded in a collection of basic operations that are deemed available to the computer by *fiat*. Technically, therefore, there is no absolute definition of computability. All we can say is that ‘given this particular set of basic functions, the computable functions are such and such.’ By common consensus, and this is supported by physical evidence that these functions can be implemented, a certain set of basic functions (and ways of combining functions) is nonetheless taken as standard.

#### PRIMITIVE RECURSION

In describing these functions, we will use the following standard notation (all of the functions relevant to recursion theory are of the same form). There is some positive integer  $n$  for which each argument of the function (let’s call it  $f$ ) is a finite list  $\mathbf{x} = (x_1, \dots, x_n)$  of precisely  $n$  non-negative integers; the result of applying the function is either undefined, or else it is itself a non-negative integer. The process in which  $f$  is applied to  $\mathbf{x}$  is written  $f(\mathbf{x})$ . The set of non-negative integers is denoted  $\mathbb{N}$ , and we say that  $f$  is a function from  $\mathbb{N}^n$  to  $\mathbb{N}$ , written  $f: \mathbb{N}^n \rightarrow \mathbb{N}$ . If the list only contains one element, say  $\mathbf{x} = (x)$  we normally write  $f(x)$  rather than  $f((x))$ , and  $f: \mathbb{N} \rightarrow \mathbb{N}$  rather than  $f: \mathbb{N}^1 \rightarrow \mathbb{N}$ . If the outcome of the process  $f(\mathbf{x})$  is undefined we say that  $f$  *diverges* at  $\mathbf{x}$  and write  $f(\mathbf{x})\uparrow$ , and if it is defined we say it *converges* at  $\mathbf{x}$  and write  $f(\mathbf{x})\downarrow$  or  $f(\mathbf{x})\downarrow a$  (or simply  $f(\mathbf{x}) = a$ ) if it is provable that the answer generated by  $f(\mathbf{x})$  is  $a$ . If  $f$  converges for every choice of  $\mathbf{x}$ , we say it is a *total* function, and otherwise that it is a *partial* function.

We always regard the constant 0 as computable. The following functions are always considered to be computable. We call these the *basic* functions:

- $0: \mathbb{N} \rightarrow \mathbb{N}$

This function, called *zero*, always returns the constant value zero:  $0(n) = 0$ .

---

- $S: \mathbb{N} \rightarrow \mathbb{N}$   
This function, called *successor*, adds one to its argument:  $S(n) = n+1$ . This allows us to represent all members of  $\mathbb{N}$ , e.g.  $1 = S(0)$ ,  $2 = S(S(0))$ , etc.
- $U_i^n: \mathbb{N}^n \rightarrow \mathbb{N}$   
This function, called a *projection*, is used to select numbers from a list:  $U_i^n(x_1, \dots, x_n) = x_i$ .

There are three basic ways in which computable functions can be combined to generate more complicated computable functions. Two of these correspond to obvious computational procedures.

- *composition*  
If  $f(x_1, \dots, x_n)$  and  $g_1(\mathbf{y}), \dots, g_n(\mathbf{y})$  are all computable functions, then so is the function  $h$  defined by  $h(\mathbf{y}) = f(g_1(\mathbf{y}), \dots, g_n(\mathbf{y}))$ .
- *primitive recursion*  
If  $f(\mathbf{x})$  and  $g(\mathbf{x}, y, n)$  are both computable, so is the function  $h$  defined by

$$h(\mathbf{x}, 0) = f(\mathbf{x})$$

$$h(\mathbf{x}, n+1) = \begin{cases} g(\mathbf{x}, n, h(\mathbf{x}, n)) & \text{if } h(\mathbf{x}, n) \text{ is defined} \\ \text{undefined} & \text{if } \mathbf{x} \text{ or } h(\mathbf{x}, n) \text{ is undefined} \end{cases}$$

Notice that the length of the parameter list  $\mathbf{x}$  depends on the functions being considered. It may even be empty, in which case we suppress  $\mathbf{x}$  altogether. The examples below show this in action. We normally don't bother including the clause saying " $h(\mathbf{x}, n+1)$  is undefined if either  $\mathbf{x}$  or  $h(\mathbf{x}, n)$  is undefined," but it is always implicit.

We say that a function is *primitive recursive* if it can be generated from the basic functions by successive applications of composition and primitive recursion. All primitive recursive functions are computable – this is because the basic functions are computable by assumption, and the processes of composition and primitive recursion represent standard computer behaviours, as I'll now explain.

*Composition.* Given a complex calculation we can always hand sub-tasks over to other routines under the control of the same computer. The program receives the various sub-answers and combines them to generate the overall result. This is all we mean by composition: the answer from one task can be used as an argument to the next.

*Primitive recursion.* The rather frightening definition given above actually encapsulates quite a straightforward idea, which is most easily illustrated by giving some basic examples. We'll start with something familiar, and show how to break it down into progressively easier cases.

One way to program the factorial function is to define  $fact(n) = n \times fact(n-1)$ . This allows us to define the values of  $fact(n)$  one at a time, provided the process is grounded by a base case. In this case we need to know that  $fact(0) = 1$ . Given this information we can evaluate  $fact(2)$ , say, by recursively unfolding the definition until the base case is reached,  $fact(2) = 2 \times fact(1) = 2 \times (1 \times fact(0)) = 2 \times (1 \times 1) = 2 \times 1 = 2$ . That's all we mean by primitive recursion. If you want to calculate the  $n$ 'th value in a sequence, you can do it by calculating its successive predecessors and manipulating the results in some carefully determined way, *provided* the process is eventually grounded in a base case. In this case, we have

$$fact(0) = 1$$

$$fact(n+1) = times(n, fact(n))$$

There's no need in this example to involve any extra parameters, so we've taken  $\mathbf{x}$  to be the *empty list*,  $()$ , and suppressed it in the equations. The function  $times(a, b) = a \times b$  is just standard multiplication. We'll see in the following example that  $times$  is primitive recursive, whence so is  $fact$ .

#### DETAILED EXAMPLE

Now we know the basic idea behind primitive recursion, let's see how we use the definition by considering the complicated-looking function  $h(m, n) = m^{(n+1)} \times fact(n)$ . This satisfies

$$h(m, n) = m \times n \times h(m, n-1)$$

and has the base case  $h(m, 0) = m^1 \times fact(0) = m$ . Because  $m$  is a variable, the base case this time is itself a *function*, in this case the *identity function*  $I(m) = m$ . Looking at the original definition this means we're taking  $f$  to be  $I$  and  $\mathbf{x}$  to be the single-element list,  $\mathbf{x} = (m)$ . The rule for evaluating  $h(m, n)$  tells us that we have to combine the three values  $m$ ,  $n$  and  $h(m, n-1)$  in a certain way; this combination process is "multiply the three values together," which is the function  $times3(a, b, c) = a \times b \times c$ . Collecting these two parts of the definition together we have

$$h(m, 0) = I(m)$$

$$h(m, n+1) = times3(m, n, h(m, n))$$

Consequently, we say that the function  $h$  – i.e. the function  $m^{(n+1)} n!$  – is *primitive recursive* in the functions  $I$  and  $times3$ . So even though  $h$  looks like a very complicated function, it is recursively-speaking quite simple, nothing more than a combination of the identity function and three-parameter multiplication.

We haven't yet shown that  $h$  is primitive recursive, because the functions we've used to construct it ( $I$  and  $times3$ ) aren't yet known to be primitive recursive themselves. We now have to show that *they* are primitive recursive.

**identity:** The identity function can be defined recursively by taking  $I(0) = 0$  and  $I(n+1) = S(I(n))$ . The problem here is that the 'recursive step' is defined using  $S$ , which is a function of only one parameter, whereas we need a function  $g(a, b)$  of two parameters. Since we want  $g(n, I(n)) = S(I(n))$  this means taking (say)  $g(a, b) = S(U_2^2(a, b))$ . This function is just the composition of  $S$  and  $U_2^2$ , both of which are basic functions, so  $g$  is primitive recursive.

$$I(0) = 0$$

$$I(n+1) = S(U_2^2(n, I(n)))$$

Consequently  $I$  is defined by applying primitive recursion to the primitive recursive functions  $0$  and  $S(U_2^2(a, b))$ , whence it is itself primitive recursive.

**times3:** Three-parameter multiplication can be defined by composition of projections and standard two-parameter multiplication:

$$times3(a, b, c) = times(a, times(b, c)) = times(U_1^3(a, b, c), times(U_2^3(a, b, c), U_3^3(a, b, c)))$$

so we need to show that  $times$  is primitive recursive.

**times:** Two-parameter multiplication can be defined recursively as repeated addition. To this end we need to choose  $g$  so that

$$\text{times}(m, n+1) = m + \text{times}(m, n) = g(m, n, \text{times}(m, n))$$

so we take  $g(a, b, c) = \text{plus}(U_1^3(a, b, c), U_3^3(a, b, c))$  which is primitive recursive provided  $\text{plus}$  is.

$$\text{times}(m, 0) = 0(m)$$

$$\text{times}(m, n+1) = \text{plus}(U_1^3(m, n, \text{times}(m, n)), U_3^3(m, n, \text{times}(m, n)))$$

Since  $\text{times}$  is defined by primitive recursion in  $\text{plus}$  all that now remains is to show that  $\text{plus}$  is primitive recursive.

**plus:** Adding  $n$  is the same as repeatedly adding 1, *i.e.* applying  $S$ . We need to choose  $g$  so that

$$\text{plus}(m, n+1) = 1 + \text{plus}(m, n) = g(m, n, \text{plus}(m, n))$$

so we take  $g(a, b, c) = S(U_3^3(a, b, c))$ . This is primitive recursive because it's defined by composition of the basic functions  $S$  and  $U_3^3$ .

$$\text{plus}(m, 0) = I(m)$$

$$\text{plus}(m, n+1) = S(U_3^3(m, n, \text{plus}(m, n)))$$

This shows that  $\text{plus}$  can be defined by applying primitive recursion to  $I$  and  $S(U_3^3(a, b, c))$ , both of which we already know to be primitive recursive.

To summarise:

	<b>This is primitive recursive</b>	<b>because</b>
1	0	defined as a computable constant
2	$U_2^2$	basic function
3	$S$	basic function
4	$S(U_2^2)$	composition of 3, 2
5	$I$	primitive recursion in 1 and 4
6	$U_3^3$	basic function
7	$S(U_3^3)$	composition of 3, 6
8	$\text{plus}$	primitive recursion in 5 and 7
9	$U_1^3$	basic function
10	$\text{plus}(U_1^3, U_3^3)$	composition of 8, 9, 6

11	0	basic function
12	$times$	primitive recursion in 11, 10
13	$U_2^3$	basic function
14	$times3 \equiv times(U_1^3, times(U_2^3, U_3^3))$	composition of 12, 9, 12, 13, 6
15	$m^{(n+1)} \times fact(n)$	primitive recursion in 5, 14

#### PROPERTIES OF PRIMITIVE RECURSIVE FUNCTIONS

There are two important properties of primitive recursive functions that are useful for our purposes. The first is that all primitive recursive functions are *total* – no matter what values you supply (as long as they're defined), the functions always generate an answer. This is a general feature of primitive recursive functions, as is easily determined by looking again at the definitions. The basic functions are obviously total, and composition of total functions yields total functions. Finally, the whole point of primitive recursion is that we can always calculate the  $(n+1)^{th}$  value of the function provided the  $n^{th}$  value is defined. Since the base case is given in terms of a function whose primitive recursive credentials are established independently, it follows by structural induction that primitive recursion over total functions also yield total functions. Therefore, all primitive recursive functions are total.

The second property is more surprising, and its proof requires more detail than can be supplied here. It is the statement that an algorithm exists for listing all possible primitive recursive functions of one argument. That is, we can enumerate these functions in an infinite list  $\{\phi_0, \phi_1, \phi_2, \dots\}$  and there is a computable function  $\Phi(m,n)$  such that  $\Phi(m,n) = \phi_m(n)$ .

#### COMPUTABLE AND RECURSIVE FUNCTIONS

All of the functions we've considered so far are *total*. In particular, then, even though all primitive recursive functions are computable, the converse can't possibly be true, because we can easily program computers to give answers for some input values, and to run forever when presented with others. If the computer runs forever while evaluating  $f(\mathbf{x})$  it will clearly fail to generate any result, so we have  $f(\mathbf{x}) \uparrow$ . Since the basic functions are all total, and both composition and primitive recursion preserve totality, we need some other way of constructing partial functions from total ones.

The solution is a process called *minimalisation*. Suppose  $g(\mathbf{x}, n)$  is a computable function and consider what happens when we vary  $n$ . Each of the values  $g(\mathbf{x},0), g(\mathbf{x},1), \dots$  is a non-negative integer, and we can try to find out which value of  $n$  is the first value for which  $g(\mathbf{x}, n)$  yields the answer 0. One way to do this is via the following program, so minimalisation is definitely a computable procedure.

- 10 Let  $n = 0$
- 20 Evaluate  $g(\mathbf{x},n)$
- 30 If  $g(\mathbf{x},n)$  yields the value 0, return the answer  $n$  and quit this program
- 40 Increase  $n$  by 1
- 50 Go back to instruction 20

This program can fail to terminate in two different ways. There may not be any value of  $n$  for which  $g$  gives the value 0 – this is possible even if  $g$  is a total function, so minimalisation can yield partial functions when applied to total ones. Or else  $g$  may itself be partial. If we find ourselves evaluating  $g(\mathbf{x}, n)$  for some value of  $n$  for which  $g$  is actually undefined, instruction 20 will run forever and no answer will be returned.

- *minimalisation*

Given a function  $g(\mathbf{x}, n)$  define

$$h(\mathbf{x}) = \begin{cases} \text{the least } n \text{ for which } g(\mathbf{x}, 0), \dots, g(\mathbf{x}, n) \text{ are all defined, and } g(\mathbf{x}, n) = 0 \\ \text{undefined if there is no such } n \end{cases}$$

We say that  $h$  is constructed by *minimalising*  $g$  with respect to  $n$ , and denote  $h$  by the expression  $h(\mathbf{x}) = (\mu n)(g(\mathbf{x}, n) = 0)$ , or just  $(\mu n)(g(\mathbf{x}, n))$ .

Formally, we define the *computable* (or *partial recursive*) functions to be those functions that can be generated from the basic functions by repeated application of composition, primitive recursion and minimalisation. If a function is both computable and total, it is said to be *recursive*. Since we can always choose to refrain from applying minimalisation if we want to, it follows that all primitive recursive functions are computable by definition, and since they're all total functions, they are also recursive.

#### RECURSIVE AND RECURSIVELY ENUMERABLE

We also extend the idea of computability to *sets*. Given any set  $A \subseteq \mathbb{N}^k$  its *characteristic function* is the (mathematical) function  $\chi_A: \mathbb{N}^k \rightarrow \mathbb{N}$  that takes the values

$$\chi_A(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is in } A \\ 0 & \text{if } \mathbf{x} \text{ is not in } A \end{cases}$$

We say that  $A$  is *recursive* if and only if  $\chi_A$  is a recursive function. A set  $A \subseteq \mathbb{N}^k$  is said to be *recursively enumerable* (*RE*) if it is the domain of a computable function. That is, if there is some computable  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  and, for each  $\mathbf{x}$  in  $\mathbb{N}^k$  we have  $\mathbf{x} \in A \leftrightarrow f(\mathbf{x}) \downarrow$ . The two concepts are closely related.

Theorem:<sup>24</sup>  $A \subseteq \mathbb{N}^k$  is recursive  $\leftrightarrow$  both  $A$  and its complement  $\mathbb{N}^k \setminus A$  are *RE*.

Theorem:<sup>25</sup>  $A \subseteq \mathbb{N}^k$  is *RE*  $\leftrightarrow$  there is a recursive function  $f: \mathbb{N} \rightarrow \mathbb{N}^k$  for which  $A = \{f(0), f(1), \dots\}$

#### UNIVERSAL FUNCTIONS

The usefulness of modern computers comes not merely from the fact that they can follow instructions, but from their ability to be *programmed*. This is a process by which the computer can be made to follow *any* set of instructions. At its core, however, the computer is just like a clerk, performing tasks without intuition of internal reflection. The reason it 'knows' how to perform the various programs it receives is that it is designed around a single hardwired program capable of simulating all other programs. From the computer's point of view, the programs we supply it are

---

<sup>24</sup> Cutland (1980) p. 124

<sup>25</sup> Cutland (1980) p. 125

---

just so much data, but the computer responds to this data in such a way that its behaviour is indistinguishable from that of a machine that ‘understands’ their intentional role as instructions. This idea, that a single program can be used to simulate all other programs, arises from Turing’s conceptualisation of ‘universal machines.’ Indeed, when a human being reads a program and pretends to be the computer executing it, the human is behaving precisely like a universal machine. The human is taking in inputs (the words on the page are just visual stimuli) and yet seems to be treating them not as data but as instructions. If the instruction says “*let price = 2 + 5*” the human being might write ‘7’ in the box on the blackboard labelled ‘price,’ and it seems that they are obeying the instruction. They are not! They are obeying quite a different set of instructions, which go something like this:

- I can see the word ‘*let*’ so I’ll try to parse the subsequent text to find an expression of the form ‘text-without-spaces =’. The text ‘price =’ fits the bill, so this ‘text-without-spaces’ is ‘price’.
- Interpret this text-without-spaces as the name of a box. OK. I’m looking for a box labelled ‘price’.
- There isn’t one. Never mind, I’ll create one now.
- Now I’ll try to find the longest bit of text that can meaningfully be regarded as a value. The text ‘2’ can be regarded as a value. Hmmm. The next non-space character is ‘+’. According to my manual, I can regard an expression of the form ‘2 + text-without-spaces’ as a value provided this new ‘text-without-spaces’ can itself be regarded as a value. Let’s see, I can make this work provided I take the new ‘text-without-spaces’ to be ‘5.’
- ‘2 + 5’ can be thought of as a value, but I need to evaluate it first. How do I do this? Oh yes, I memorised that particular sum at school, and here comes the answer now from my long-term memory. It’s 7.
- The value in question is 7 and the box is called *price*. My manual says to interpret any text of the form ‘*let boxname = value*’ as an instruction to write the value in the box with that name, so I better write ‘7’ in the box labelled ‘price.’

Looking at this sequence of events it’s clear that the human being is certainly following a set of instructions, but they are not the instructions of the original program. For one thing, there’s the use of an auxiliary variable called ‘text-without-spaces,’ and for another, there’s the clear need to perform relatively complex parsing on the input text. The same, of course, is true of electronic computers. Humans who pretend to be computers are just like electronic computers – they go to a great deal of trouble doing things in order to make it look like they’re doing something else. The thing they’re *actually* doing is still a program; it’s just not the program you think it is! The *actual* program is one that reads in things like text and parses it in such a way that it can interpret the text as a set of instructions. It then simulates the effect of obeying those instructions. The reason this is possible is very simple and very obvious: there is absolutely no physical difference between data and programs (they are both made up of arbitrary symbols) so the computer is free to treat a string of digits (say ‘011010110’) as either an instruction or as data, or as both, depending on what its program tells it to do.

## ORACLES

We saw above that computation is never defined absolutely, but only with respect to the basic functions that are assumed to be computable by *fiat*. If a function  $\phi$  is not currently computable, we can consider adding  $\phi$  as a new basic function, and asking how this enlarges the set of computable functions. If  $\psi$  is part of this new set of computable functions, we say that  $\phi$  is acting as an *oracle*

---

for  $\psi$ , and say that a  $\psi$  is computable *relative* to  $\phi$ . Oracles play an important part in linking Turing's research to hypercomputation, because he is known to have considered models in which an oracle  $\psi$  is supplied as a basic functions by the simple expedient of attaching an auxiliary machine capable of generating the consecutive values  $\psi(0)$ ,  $\psi(1)$ , and so forth.

#### COMPUTATION OF $\sigma$

Given a program  $P$  one can ask whether or not  $P$  eventually terminates, or else runs forever. It is well known that no program can solve this *Halting Problem*. On the other hand, it is possible to find a universal program  $U$  with the property that  $U(n)$  simulates every possible program as  $n$  ranges through the values 1, 2, ... In other words, the set of all programs is *RE*. If  $\Gamma$  is a recursive function for which  $\Gamma(n)$  lists every program, we can adjust  $\Gamma$ , if necessary, so that the list contains no duplicates, and in that case we call  $\Gamma$  a Gödel numbering of the computable functions.

Choose some Gödel numbering  $\Gamma$  of the programs, so that we can arrange all programs (without duplicates) in a list  $P_1, P_2, \dots$ ; this is just the list  $\Gamma(1), \Gamma(2), \dots$ . In the main text of this paper I have defined a value  $\sigma$  whose binary expansion has the form

$$\sigma_n = \begin{cases} 1 & \text{if } P_n \downarrow \\ 0 & \text{if } P_n \uparrow \end{cases}$$

Clearly, if  $\xi$  were a computable function for which  $\xi(n) = \sigma_n$ , then  $\xi$  would solve the Halting problem. Consequently no program can compute the digits of  $\sigma$ . Nonetheless I claim that an algorithm can be written which computes  $\sigma$ .

The structure of my algorithm is very simple. We will initialise the value of  $\sigma$  to be 0, and then start running simulations of all the  $P_n$ s. If the simulation of some  $P_n$  eventually halts, we'll effectively stop that particular simulation and add the value  $(1/2)^n$  to  $\sigma$ . Consequently, whenever  $n$  is one of those values for which the question "does  $P_n$  eventually halt?" is undecidable, the ultimate value of  $\sigma_n$  will only be known once the program runs to completion. This is *why* no computable function  $\xi$  can be found which gives  $\sigma_n$  in finite time. We cannot simply calculate  $\sigma_1$  first, then  $\sigma_2$ , and so on, because we will eventually hit a digit whose value can't be evaluated in finite time, and this will limit how well we can approximate  $\sigma$ . Instead, we run a large looping computation. The first time round we simulate the first instruction of  $P_1$ . The second time round we simulate the second instruction of  $P_1$  and the first instruction of  $P_2$ . Then the third instruction of  $P_1$ , the second of  $P_2$  and the first of  $P_3$ . Each time round the loop we progress the execution of those programs we've already initiated, and then initiate one more program, so by the time we complete the  $n$ 'th loop we are simulating  $n$  different programs. As the program continues, so every program is eventually brought into the simulation process, and the value of  $\sigma$  is approximated ever more closely.

Here's my program for computing  $\sigma$ , expressed in pseudo-code to keep it intelligible to non-programmers. The arrays *instruction[]* and *state[]* are used to hold information about the state of each program's simulation and the instruction it considers to be next in line (this can change unexpectedly in response to GOTO instructions, but otherwise the instruction counter simply increases by one each time round). Consequently, *instruction[n]* holds the line number of the next instruction to be executed in  $P_n$ , and *state[n]* holds  $P_n$ 's internal state information. Given this information we can simulate  $P_n$  by repeatedly performing its *instruction[n]*'th instruction and storing the changing state information in *state[n]*. Remember, we can generate the entire listing of  $P_n$  at any time by evaluating  $\Gamma(n)$ , so identifying particular instructions by number is no problem. The array *halted[]* keeps track of which simulations have halted so far. At any time during this

computation only finitely many of the various array values are in use, so we only ever use finite memory resources.

```

Compute_Sigma() ≡ {

    // Start with  $\sigma$  set to zero
    let  $\sigma$  = 0;

    // Start the main looping process
    let loop = 0;
    repeat (forever) {

        // Increment the loop counter
        let loop = loop + 1;

        // Initiate the loop'th program
        let halted[loop] = false;
        let instruction[loop] = 1;
        let state[loop] = standard_start_state;

        // Progress any of the first loop programs that haven't halted yet
        for n = 1 to loop do {
            if (not halted[n]) {
                simulate instruction[n]'th line of  $P_n$ ;
                update instruction[n] and state[n];
                if ( $P_n$  has reached a halt state) {
                    let halted[n] = true;
                    let  $\sigma$  =  $\sigma + (\frac{1}{2})^n$ ;
                }
            }
        }
    } // END of progressing non-halted programs
} // END of repeat(forever)
}

```

## 7 References

- Bains, S. and Johnson, J. 2000 'Noise, physics and non-Turing computation.' *Joint Conference on Information Systems*, Atlantic City, 28 February – March 3, 2000.
- Benioff, P. 1980. 'The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines.' *J. Statist. Phys.*, 22, 563-591.
- Bennett, C.H. 1993. 'Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels.' *Phys. Rev. Lett.*, 70, p. 1895-1898.
- Bennett, C.H. 1995. 'Quantum information and computation.' *Phys. Today* 48(10), 24-30.
- Blanck, J. *et al* 2000. *Computability and Complexity in Analysis*. Informatik Berichte 272-9/2000, FernUniversität, Fachbereich Informatik, Postfach 940, D-58084, Hagen, Germany.
- Boden, M.A. 1988. *Computer Models of Mind*. Cambridge: Cambridge University Press.
- Church, A. 1936a. 'An unsolvable problem of elementary number theory'. *American Journal of Mathematics*, 58, 345-363.
- Church, A. 1936b. 'A note on the Entscheidungsproblem'. *Journal of Symbolic Logic*, 1, 40-41.
- Churchland, P.M. 1988. *Matter and Consciousness*. Cambridge, Mass.: MIT Press.
- Copeland, B.J. 1997. 'The Church-Turing Thesis.' In E. Zalta (ed.) *The Stanford Encyclopaedia of Philosophy*. Available at <http://plato.stanford.edu/entries/church-turing/>.

- Copeland, B.J. 2000. 'Narrow versus Wide Mechanism: Including a Re-examination of Turing's Views on the Mind-Machine Issue.' *J. Philosophy*, 97(1).
- Copeland, B.J. and Sylvan, R. 1999 'Beyond the Universal Turing Machine.' *Australasian Journal of Philosophy* 77, pp. 46-66.
- Crisp, R., 1995. 'Scientific Determinism'. In *The Oxford Companion to Philosophy*, p. 196, Oxford: Oxford University Press.
- Cutland, N.J. 1980. *Computability: An Introduction to Recursive Function Theory*. Cambridge: Cambridge University Press.
- Danielsson, U. 2001 'Introduction to string theory.' *Rep. Prog. Phys.* 64, 51-96.
- Davies, E.B. 2001 'Building Infinite Machines.' *British Journal for Philosophy of Science*.
- Deutsch, D. 1985. 'Quantum theory, the Church-Turing principle and the universal quantum computer.' *Proc. Roy. Soc. Lond.*, A 400, 97-117.
- Dornheim, C. 1998. 'Undecidability of Plane Polygonal Mereotopology.' In A.G. Cohn, L. Schubert, S.C. Shapiro (eds), *Principles of Knowledge Representation and Reasoning, Proceedings of the 6<sup>th</sup> International Conference (KR'98)*, pp. 342-353, Trento, Italy.
- Earman J. and Norton J. 1993 'Forever is a day: supertasks in Pitowsky and Malament-Hogarth Spacetimes.' *Philosophy of Science* 5:22-42.
- Feynman, R. 1986. 'Quantum mechanical computers.' *Found. Phys.*, 16, 507-531, 1986; originally in *Optics News* (February 1985), 11-20.
- Feynman, R.P., Leighton R.B. and M. Sands, 1965. *The Feynman Lectures on Physics* (3 volumes). Addison-Wesley, 1965.
- Gandy, R. 1980. 'Church's Thesis and Principles for Mechanisms'. In Barwise, J., Keisler, H.J., Kunen, K. (eds) 1980. *The Kleene Symposium*. Amsterdam: North-Holland.
- Gödel, K 1931 'Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I.' *Monatshefte für Mathematik und Physik*, 38, 173-198.
- Goodstein, R.L. 1964. *Recursive Number Theory*. (Brouwer, Beth and Heyting (eds), *Studies in Logic and the Foundations of Mathematics*). Amsterdam: North-Holland.
- Gowland and Lester, 2000. 'Survey of Exact Computer Arithmetic.' In (Blanck 2000).
- Hobson, E.W. 1923. *The Domain of Natural Science*. Cambridge University Press. Reissued by Dover Publications, 1968.
- Hogarth, M. 1992 'Does General Relativity allow an observer to view an eternity in a finite time?' *Foundations of Physics Letters* 5:173-181.
- Hogarth, M. 2001 'Deciding Arithmetic in Malament-Hogarth Spacetimes.' (preprint, submitted to BJPS, October 2001)
- Loez, J.L. 1996 'Supersymmetry: from the Fermi scale to the Planck scale.' *Rep. Prog. Phys.* 59, 819-65.
- Meltzer, B. 1962 *Kurt Gödel – On formally undecidable propositions of Principia Mathematica and related systems*. English translation of (Gödel 1931) with an introduction by R.B. Braithwaite. Edinburgh and London: Oliver & Boyd.
- Milner, R. 1989 *Communication and Concurrency*. Prentice-Hall International.
- Siegelmann, H.T. 1999 'Stochastic Analog Networks and Computational Complexity.' *J. Complexity* 15, 451-75.
- Stannett, M. 1990. 'X-machines and the Halting Problem: Building a super-Turing machine.' *Formal Aspects of Computing*, 2, 331-341.
-

- Stannett, M. 1991 *An Introduction to post-Newtonian and non-Turing Computation*. Technical Report CS-91-02, Dept of Computer Science, Sheffield University, UK.
- Stannett, M. 2001 'Hypercomputation is experimentally irrefutable.' Technical Report CS-01-04, Dept of Computer Science, Sheffield University, UK.
- Steane, A. 1998. 'Quantum computing.' *Rep. Prog. Phys.*, 61, 117-173.
- Turing, A.M. 1936. 'On Computable Numbers, with an Application to the Entscheidungsproblem'. *Proc. LMS*, Series 2, 42 (1936-37), pp.230-265.
- Turing, A.M. 1946. 'Proposal for Development in the Mathematics Division of an Automatic Computing Engine (ACE)'. In Carpenter, B.E., Doran, R.W. (eds) 1986. *A.M. Turing's ACE Report of 1946 and Other Papers*. Cambridge, Mass.: MIT Press.
- Turing, A.M. 1948. 'Intelligent Machinery'. National Physical Laboratory Report. In Meltzer, B., Michie, D. (eds) 1969. *Machine Intelligence 5*. Edinburgh: Edinburgh University Press.
- Vuillemin, J. 1987. 'Exact real arithmetic with continued fractions.' Technical report, Institut de Recherche en Informatique et Automatique, 78150, Rocquencourt, France, 1987.
- Weatherford, R.C. 1995. 'Determinism'. *The Oxford Companion to Philosophy*, p. 195, Oxford: OUP, 1995.
- Zhong, N. and Weihrauch, K. 2000 (preprint) 'Computability Theory of Generalised Functions.'

Sheffield, March 2002