Available online at www.sciencedirect.com
SCIENCE DIRECT®
Applied Mathematics and Computation 178 (2006) 143–153
APPLIED MATHEMATICS AND COMPUTATION
www.elsevier.com/locate/amc

The many forms of hypercomputation

Toby Ord

Balliol College, Oxford OX1 3BJ, UK

Abstract

This paper surveys a wide range of proposed hypermachines, examining the resources that they require and the capabilities that they possess.

Keywords: Hypercomputation; Oracles; Randomness; Infinite time Turing machine; Accelerating Turing machine

1. Introduction: computation and hypercomputation

The Turing machine was developed as a formalisation of the concept of computation. While algorithms had played key roles in certain areas of mathematics, prior to the 1930s they had not been studied as mathematical entities in their own right. Alan Turing [41] changed this by introducing a type of imaginary machine which could take input representing various mathematical objects and process it with a number of small precise steps until the final output is generated. Turing found a machine of this type corresponding to many of the traditional mathematical algorithms and provided convincing reasons to believe that any mathematical procedure that was precise enough to be recognised as an algorithm or 'effective procedure' would have a corresponding Turing machine. Other contemporaneous attempts at formalising algorithms [4,26] were soon shown to lead to the same class of 'recursive functions'.

The immediate benefit of this analysis of computation was the ability to prove certain negative results. Turing showed that, on pain of contradiction, there could not be a Turing machine which took an arbitrary formula of the predicate calculus and decided whether or not it was a tautology. Since it was agreed that every algorithm had a corresponding Turing machine, Turing's result implied that there could be no algorithm at all for deciding which formulas were tautologies. This style of argument was used to great effect and many problems were shown to be thus lacking an algorithmic solution.

However, there has been considerable recent interest in theoretical machines which have more resources available to them than does the Turing machine.[1] It is well known that certain extensions to the Turing machine such as additional memory tapes or non-determinism do not allow it to compute any new functions.

E-mail address: toby.ord@balliol.oxford.ac.uk
[1] Jack Copeland [10] gives an excellent survey of the field.

0096-3003/$ - see front matter
doi:10.1016/j.amc.2005.09.076

Indeed this is often cited as a reason to accept the recursive functions as being the natural class of 'computable functions'. However, it has been shown that by adding the ability to perform additional primitive functions, to acquire input from the outside world, to perform infinite precision operations on real numbers, or to perform an infinite number of computational steps can all increase the power of the Turing machine and allow it to compute non-recursive functions.

This should not be so surprising. These new resources are qualitatively different to the old ones. The new machines go beyond Turing's attempts to formalise the rote calculations of a human clerk and instead involve operations which may not even be physically possible. This difference in flavour is reflected in the terminology: they are hypermachines and perform hypercomputation. Can they be really said to 'compute' in a way that accords with our pre-theoretic conception? It is not clear, but that is no problem: they hypercompute. Hypercomputation is thus a species of a more general notion of computation which differs from classical Turing computation in a manner that is difficult to specify precisely, yet often easy to see in practice.

To better understand the relationship between classical computation and hypercomputation, it is useful to consider the relationship between Euclidean and non-Euclidean geometry. The original axioms of geometry were chosen so as to reflect our common sense knowledge of space and to formalise an idealization which we could study mathematically. These goals were met and geometry became a very successful field of mathematics. However, early in the 18th century mathematicians began to realize that the axioms of Euclidean geometry were a special case of a more general theory of geometry which also included unintuitive (but logically consistent) non-Euclidean geometries. After considerable reluctance, the mathematical community were persuaded to accept non-Euclidean geometry as a branch of geometry, but it was considered to be merely of academic interest since the geometry of the physical world was Euclidean. Later, in the 20th century, the general theory of relativity completely vindicated non-Euclidean geometry, showing that despite a Euclidean appearance on the small scale, the geometry of the universe is in fact non-Euclidean.

Just as Euclidean geometry was an idealization of space, so classical computation is an idealization of our common sense conception of algorithmic procedures. Like Euclidean geometry, this idealization of algorithms has been a spectacularly successful mathematical theory. Again, however, mathematicians have begun to notice extensions to the idealized theory which make it a special case of a more general theory. As with Euclidean geometry, classical computation is a very important special case, one that was chosen for its connection to actual algorithmic practices, and for all we know hypercomputation might be a physical impossibility. However, it would appear that we have no more reason to rule out physical hypercomputation than the scientists of the 18th and 19th centuries had to rule out physical non-Euclidean space. Indeed, as we shall see, there are several places within current physical theories that hypercomputational processes might be found and there is little argument in the literature to show that of all the myriad ways that physical processes might combine, none of them will be hypercomputational.[2] The question of whether or not we can physically compute more than the Turing machine is of fundamental importance and very much open.

It is worth responding, at this point, to a persistent argument that is made against the coherence of physical hypercomputation. It states that a machine must perform an infinite number of computations for it to count as hypercomputational. After all, for any function f on the integers, there is a Turing machine that computes $f(n)$ for an arbitrarily large finite number of values of n. It is thus claimed either that we could not know that a prospective machine was a hypermachine after witnessing finitely many computations or that it simply *would not be* a hypermachine since its behaviour could be simulated by a Turing machine. Hypercomputation is thus claimed to be on shaky ground. However, it is easy to see that this argument cannot achieve what it hopes to since it does not just collapse hypercomputation to classical computation, but instead it collapses all computation on the integers down to that of finite state machines. This is because for any function f on the integers, there is also a finite state machine that computes $f(n)$ on an arbitrarily large finite domain. Thus, as explained in Copeland [10], whatever force this argument is supposed to have with respect to hypercomputation, it must also have with respect to refuting the existence of classical (general recursive) computation. Since the latter is agreed to be on firm ground, it is difficult to see why this argument should count against the former.

[2] One such attempt is that of Gandy [17].

Let us suppose, however, that hypercomputation does turn out to be physically impossible—what then? Would this make the study of hypercomputation irrelevant? No. Just as non-Euclidean geometry would have mathematical relevance even if physical space was Euclidean, so too for hypercomputation. Perhaps, we will find certain theorems regarding the special case of classical computation easier to prove as corollaries to more general results in hypercomputation. Perhaps our comprehension of the more general computation will show us patterns that will guide us in conjectures about the classical case. Just as the development of the complex numbers helped us prove new theorems about the real numbers, so too we might hope to prove new theorems about classical computation using the tools of hypercomputation.

There are also philosophical issues that arise when one takes the idea of physical hypercomputation seriously. For example, we can take the idea of mind as a computational process and extend it to mind as a hypercomputational process [7,8,3]. What can we say about such minds? Could this be the nature of our own minds? There are also questions concerning epistemology and what counts as mathematical proof. For example, there is an algorithm for an infinite time Turing machine which determines the truth of any formula of first order arithmetic, a feat that no standard Turing machine can perform. Would such a 'hyperproof' be a valid proof despite our inability to follow it through without mechanical aid? What about for hypothetical beings who could follow it through? If physical hypercomputation is possible then these questions would be unavoidable. Even if it is not, the very possibility of hypercomputation makes these questions relevant in the study of epistemology and the philosophy of mathematics.

Hypercomputation has been studied as a topic in its own right for several years now and there are many examples of hypermachines in the literature. These hypermachines are, like the Turing machine, *theoretical machines* using abstract resources to manipulate abstract objects such as symbols or numbers. Thus, when I say that there exists a machine of a specified type which computes the halting function, I am of course making a theoretical claim and not a physical one. Nevertheless the hypercomputational resources are often physically inspired and there is considerable interest as to whether these devices are physically possible—both in theory (is it compatible with certain physical laws?) and in practice (could we actually build one?).

In what follows I shall present a broad collection of hypermachines from the literature and discuss the hypercomputational resources that they use.[3] The focus here shall be on the mathematical and philosophical nature of such resources, touching only briefly on issues of physical realisability—something I consider best left to the specialists. Finally, I shall present an overview of the comparative capabilities of the hypermachines discussed, allowing one to see how much power can be derived from a given resource.

2. Resources

2.1. Oracles

In 1939, Turing [42] considered an interesting modification to his machines. Suppose that a Turing machine was given access to an 'oracle': some unspecified device that can answer questions regarding membership in a specific set of natural numbers. In addition to its usual operations, this oracle-machine, or *o-machine* can specify a certain natural number on its tape and enter a special oracle state, which asks the oracle whether that number is in the oracle set. If the oracle set were recursive then the *o*-machine would gain no new power, but what if the oracle set was not itself computable by Turing machines?

In such a case, the *o*-machine could compute an infinite number of non-recursive functions. Of course it could trivially compute the characteristic function of the oracle set, but it could also incorporate its requests to the oracle into more complex algorithms, allowing non-trivial computations of non-recursive functions. For example, if the oracle set was the *halting* set (containing n iff the nth Turing machine halts) then the *o*-machine would be able to compute any recursively enumerable function.

[3] Due to limitations of space and my own technical competence, I shall pass over Kieu's quantum adiabatic hypercomputation, but the reader can find ample discussion in [25]. I shall also pass over the so-called Putnam–Gold machines [33,19,28] which I consider to be just Turing machines performing limiting computation. The question of how much a given (hyper)machine can compute under the weaker notions of computation (semi-computation, limiting computation, limiting verification, limiting gradual verification, etc. [24]) has considerable interest, but I shall not pursue it here.

A non-recursive oracle is thus a sufficiently powerful resource as to extend the power of the Turing machine. Because of this the *o*-machine is typically considered to be the first hypermachine.[4] It is debatable as to whether Turing would have thought of it in this way, since he used it not as an independent model of computation but rather to provide a definition of being computable relative to another function.[5] However, one *can* interpret it as a hypermachine (regardless of whether Turing did) and it shall be convenient for us to do so here.

In the above formulation of the *o*-machine, the oracle is a black box component. We know that it must give the correct results but how it does so is left deliberately unspecified by Turing. In a way this is no different to the workings of the classical components of the Turing machine (how does it read a square of the tape?) but there is something about the nature of the oracle that begs for further explanation.

One, more explicit, specification of an o-machine is via an oracle tape. In this formulation, the *o*-machine has an additional tape which comes initially inscribed with a sequence of ones and zeros. These are arranged in such a way that the *n*th square contains a 1 if *n* is in the oracle set and a 0 otherwise. The two formulations of oracle machines are easily seen to be mathematically equivalent and the second is now more common. However, when considering hypercomputation there is a potentially important difference between the two. While the oracle-tape formulation explicitly requires an infinite amount of memory, Turing's black box formulation does not.

There are several other kinds of hypermachine in the literature that are 'oracular' in nature. For example, Copeland [5] has introduced the *coupled Turing machine* which is a Turing machine that is connected to some form of input channel. As well as its normal operations, it can read the current input from the stream and choose its next action based upon this value. This model has many plausible physical implementations involving such ubiquitous input channels as keyboard and mouse or perhaps some complex apparatus which returns the results of quantum mechanical experiments. If these produced a non-recursive input sequence, then the machine would be able to compute non-recursive functions.

Finally, we can look at what we consider to be flaws in our current physical implementations. Copeland and Sylvan [11] point out that a network of Turing machines operating asynchronously can compute non-recursive functions if the timing functions (specifying when a given machine will perform its next computational step) are non-recursive: a possibility that is difficult to rule out in today's networks. Alternatively, we could consider Turing machines that occasionally make an error when reading from or writing to the tape. If the function describing when such errors occur is non-recursive, such a machine is clearly capable of hypercomputation, albeit of a not very useful kind.

A serious issue with oracular hypercomputation concerns potential sources of non-recursive input streams. It has been suggested [11] that it should be a great scientific astonishment if every single physical process in the universe were recursive. Supposing, therefore, that there are non-recursive processes in the universe, we could couple one to a Turing machine, and thus build a working hypermachine. However, while this would indeed give us hypercomputation, it may not be *harnessable* hypercomputation. We would very much like to be able to perform certain non-recursive tasks such as solving the halting problem, but access to some arbitrary non-recursive stream is not sufficient for this. We would need a stream that has the right information encoded within it and it is difficult to see how such a thing could occur.

Perhaps the most plausible possibility concerns fast growing functions. Consider Tibor Rado's *shift function* [34] which takes on the value of the largest number of steps that a Turing machine with *n* states can make before halting when it is started on the blank tape. It is easy to see that given this function we could compute the halting function. More importantly, however, we could also compute the halting function given any function that grows faster than the shift function. Given the robustness of this type of input stream (any upper bound of the shift function will do) it seems to possess a certain physical plausibility.[6]

[4] See Copeland [10].

[5] See Davis [13].

[6] This argument can be generalised to other limits on growth. Indeed there is an important dichotomy for physics: either we can physically compute the halting function or there must be no physical processes which grow faster than the shift function and none that grow more slowly than the inverse shift function; none that limit to a value more quickly than the reciprocal of the shift function and none that limit to a value more slowly than the reciprocal of the inverse shift function. This would be a physical limit of a kind quite unlike any proposed by our current theories.

2.2. Randomness

An obvious method for producing a non-recursive (albeit unharnessable) input stream is via quantum randomness. However, there is a classical result [18] which appears to bar such an approach, stating roughly that while access to a random bit stream may speed up computations, it does not allow a Turing machine to compute any non-recursive functions. This classical result depends upon a certain definition of what it is for a probabilistic Turing machine to compute a certain function. In particular, it assumes that a probabilistic Turing machine computes the function f if and only if given an input n, the probability of it returning $f(n)$ is greater than $\frac{1}{2}$. This is a good definition for a concept of probabilistic computability at the machine level. That is, it provides a particular function (or partial function) corresponding to each machine.

However, there is another natural concept of probabilistic computability at the level of an individual computation. When a probabilistic Turing machine is run multiple times it may produce different output from the same input. On this second concept of computability, we would say that if there was different output for the same input, then the machine computed a different function on each run. For example, if a probabilistic Turing machine was set up so as to merely take the random bits one by one and print them out on its tape, then it is natural to say that it computes a different sequence each time it is run.[7] While these sequences are not repeatable and it would appear that we cannot harness them to confidently prove any new theorems or solve new problems, we do know that the sequence will be non-recursive with probability one. We could even have a single machine that creates an ever growing table of random bits while other machines use this table to reliably compute certain non-recursive functions. These machines would be using the table as an oracle and operating deterministically with its information, computing the same function every time they are run. Using quantum randomness it appears that we can already build such machines.

Alternatively, Ord and Kieu [32] consider a Turing machine with the ability to flip a biased coin[8] that has probability p of landing heads. If p is a non-recursive real, then the probabilistic Turing machine can compute non-recursive functions. In effect, it can use the probability as an oracle, where n is in the oracle set if and only if the nth digit of the binary expansion of p is 1. The oracle set can be extracted from the biased coin quite simply: it is just a matter of approximating the value of the coin's bias by flipping it a very large number of times and this can be made to work. In any given run, only finitely many bits of p would be required and thus the coin only needs to be flipped finitely many times. Furthermore, we do not actually require a coin with an (infinitely precise) non-recursive mean. Instead, we could use a sequence of coins with increasingly accurate rational biases or even coins whose biases are chosen from any probability distribution with a given non-recursive mean.[9] This approach to oracular hypercomputation via randomness has a benefit over most other approaches that involve continuous quantities in that it does not require the measurement process to get more precise when more bits from the oracle are required.

2.3. Fair non-determinism

It is well known that Turing machines can be generalised to behave non-deterministically [23]. These non-deterministic Turing machines can have states with several applicable transitions, causing the computation to 'branch'. The machine is defined as outputting 1 if and only if there is a branch which leads to it halting in an accepting state. It returns 0 if and only if all branches lead to rejecting states and it diverges otherwise. In this manner, a non-deterministic Turing machine can be considered as using parallel processes (or lucky guesses) to quickly compute its solution. While this may well lead to significant speedups in computation time, it does not allow the machine to compute any non-recursive functions since a non-deterministic machine is readily simulated by a deterministic one. However, by restricting this non-determinism to *fair* computations (as done by Edith Spaan, Leen Torenvliet and Peter van Emde Boas [38]), the situation is quite different.

[7] It may be objected that this is so simple a process that it does not count as computation at all. In this case replace the example (which was chosen simply for clarity) with a machine that prints out the digits of π in binary, but inverts the nth digit if the nth random bit is a 1.

[8] Of course when we consider physical implementations the randomising device need not have any similarity to an actual coin.

[9] Indeed, we could also combine these two options and use a sequence of distributions around finitely accurate, recursive means. Providing that these means converge to a non-recursive value, a Turing machine could use the results to perform hypercomputation.

A computation branch of a non-deterministic Turing machine is said to be *unfair* iff it holds for an infinite suffix of this branch that the machine is in a state s infinitely often but one of the transitions from s is never chosen. A non-deterministic Turing machine is called *fair* if it produces no unfair computations. Now consider a fair non-deterministic Turing machine F which is designed to take a machine/input pair and determine whether or not it halts. F first goes to the end of its input and non-deterministically writes an arbitrary natural number n in unary. This can be done by beginning with 0 and non-deterministically choosing between accepting this number and moving on to the next stage or incrementing the number and repeating the non-deterministic choice. Once F has generated its value for n it can then check to see whether the machine/input combination halts in n steps. If it does, F accepts. If not, it rejects. It is clear that if there is a time at which the machine/input combination halts, F will have a finite computation branch that accepts and F will thus halt.

What if the given combination does not halt? In this case, the only way F will fail to halt is if it has an infinite computation branch. The only place one could occur is when the arbitrary number is being generated, but the only way this could happen is if the number was incremented an infinite number of times. Since it would be choosing the 'increment' branch over the 'continue the computation' branch infinitely often, this would be an unfair computational branch and would thus be impossible. Since all the other branches halt in the rejecting state, the machine will return 0 which is the correct answer. In this strange way, fair non-deterministic Turing machines can compute the halting function.

An interesting question about fair non-deterministic Turing machines concerns the amount of time it takes for them to halt. For standard non-determinism, this is the length of the shortest accepting branch (if there is one) or the length of the longest rejecting branch (if all reject). For fair non-deterministic Turing machines, the length of the computation would be defined similarly in the case of acceptance, but what about rejection? For example, if the machine F is given as input the code of a Turing machine which does not halt on its input, then there are infinitely many computation branches, all of which halt in a rejecting state and there is no upper bound on the length of such branches. While we could define the run time in such cases to be whatever we wish, the natural answer appears to be that it is infinite, conflicting with the idea that it does actually halt. Considerations such as this show fair non-determinism to be a rather strange hypercomputational resource and one that has relatively little physical plausibility.

2.4. Infinite time

In the early 20th Century, Bertrand Russell [35], Ralph Blake [1] and Hermann Weyl [45] independently proposed the idea of a process that performs its first step in one unit of time and each subsequent step in half the time of the step before. Since $1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots<2$, such a process could complete an infinity of steps in two time units. The application of this temporal patterning to Turing machines has been discussed briefly by Ian Stewart [39] and in much more depth by Copeland [6,9] under the name of *accelerating Turing machines*. Since Turing's account of his machines has no mention of how long it takes them to perform an individual step, this acceleration not in conflict with his mathematical conception of a Turing machine.

Consider an accelerating Turing machine A that was programmed to simulate an arbitrary Turing machine on arbitrary input. If the Turing machine halts on its input, A then changes the value of a specified square on its tape (say the first square) from a_0 to a_1. If the Turing machine does not halt, then A leaves the special square as 0. Either way, after 2 time units, the first square on A's tape holds the value of the halting function for this Turing machine and its input.

So far, there has been no difference between an accelerating Turing machine and a standard Turing machine other than the speed at which it operates. In particular, A has not solved the halting problem because Turing machines are defined to output the value on their tape after they halt. In this case, A does not halt if its simulated machine does not halt. However, the situation described above suggests a simple change which will allow A to solve the halting problem—we consider the machine's output to be whatever is on the first square after two time units.

This process of using an infinite computation length can be extended. Joel Hamkins and Andy Lewis [21,20] have presented a model of a Turing machine that operates for a transfinite number of steps: an *infinite time Turing machine*. We could imagine, for instance, a machine that included an accelerating Turing machine (M) as a part. It could initiate M's computation, then after two time units, stop M's movements and reset M to its

initial state, leaving the tape as it was at the end of the computation. It could then restart M with its tape head on the first tape square, running it for another two time units. In such a manner, this machine would perform two infinite sequences of steps in succession. One could even imagine a succession of infinitely many restarts, with M performing the whole sequence twice as fast each time, leading to an infinite sequence of infinite sequences of steps.

The infinite time Turing machine is a natural extension of the Turing machine to transfinite ordinal times. To determine the configuration of the machine at any successor ordinal time, the new configuration is defined from the old one according to the standard Turing machine rules. At a limit ordinal time, however, the machine's configuration is defined based on all the preceding configurations. The machine goes into a special *limit-state* and each tape square takes a value as follows:

$$\text{square } n \text{ at time } \lambda = \begin{cases} 0, & \text{if the square has settled down to 0,} \\ 1, & \text{if the square has settled down to 1,} \\ 1, & \text{if it alternates between 0 and 1 unboundedly often.} \end{cases}$$

The tape head is placed back on the first square and the machine then continues its computation from this limit-state as it would from any other. As usual, if there is no appropriate step to execute at some point, the machine halts. It can thus perform a finite number of steps and halt, or an infinite number of steps and halt, or keep operating through all of the ordinal times and never halt.

There are several ways in which we may hope to physically perform an infinite number of steps in a finite time. One of these is suggested by the accelerating Turing machine and its increasing speed. However, if one imagines an ordinary computer which is accelerated in some manner, signals will need to travel arbitrarily fast and it will run into conflict with the special theory of relativity.[10]

Another possibility, considered by Brian Davies [12] is to have a series of machines, which together compute the function. Each machine would perform one computational step and then build the next machine in the sequence. If the machines diminish in size at an appropriately fast rate, then the distances that need to be traversed will diminish in time with the increased computational speed required and the physical speeds required can be bounded. In addition, the amount of matter needed to build the infinite sequence of machines will also be bounded. However, while this approach does seem to be consistent with Newtonian mechanics, it requires infinite precision[11] and this would appear to conflict with quantum mechanics.

There has also been some research on using general relativity to accelerate the computation with respect to a certain observer. Mark Hogarth [22] demonstrates that within a particular type of space-time (which our universe may or may not possess), there can be situations in which a form of observer-relative acceleration occurs and an observer can witness the evolution of an entire infinite world-line by some finite time t. One could then propose to solve the halting problem by having a classical computer travelling along the specified world-line, simulating a Turing machine on its input and sending a signal if it finds that the machine halts. If the observer witnesses the signal, then she knows that the Turing machine halts on that input and if there is no signal by t, then she knows that it does not. Relativistic hypercomputation of this sort has been followed up by several other authors who have discovered new concerns regarding the physical realisation of such processes as well as new techniques which may overcome these [14–16,30].

Infinite computations are also subject to the famous paradox of *Thomson's lamp* [40]. If we ran an accelerating Turing machine with a program which simply wrote a 1 on the first square of the tape then replaced it with a 0 and repeated this procedure infinitely many times, what would be on this square after two time units? There does not seem to be a natural answer. The specification of an accelerating Turing machine made sure that one could only change the designated square once to avoid exactly this type of problem. However, this situation still arises on the standard parts of the tape. While we do not need to look at these squares for the computation to work, it seems quite problematic if they may have no defined value after two time units. In the

[10] There is no *direct* conflict with special relativity since it allows objects to move faster than light, so long as they never move slower than the speed of light. However, this is not a promising beginning for such an approach.

[11] Strictly speaking, each machine is only required to be finitely precise, but for every finite level of precision, a machine will be required which exceeds it.

case of accelerating Turing machines, one can simply restrict the model so that no square can be changed more than once. This completely eliminates Thomson's lamp style problems and (by a result due to Minsky [29]) it can be seen that this does not reduce the machine's power. Infinite time Turing machines suffer from a similar problem: they *do* have a defined answer for such cases, but it is difficult to see how it could be enforced physically. A similar approach can be applied [44], restricting the number of times a square can be changed between limit times, although in this case the approach does somewhat reduce the power of the model.

2.5. Infinite input

While the standard definition of the Turing machine involves only finite input, there is a natural generalisation to infinite input.[12] By allowing the input to be written across infinitely many squares of the machine's tape, it is no longer restricted to countable input sets such as the natural numbers, but can compute functions on sets such as the reals. Consider, for example, a particular machine that takes a real number x (encoded as a sequence of rapidly converging rational intervals) and squares it. This machine could operate by squaring the rational endpoints of each successive interval and outputting these as it progresses. While the machine has only scanned a finite amount of the input at any time, this does not prevent it gradually outputting a representation of x^2. All such computable functions on the reals are continuous.

It is possible for the Turing machine with infinite input to produce a non-recursive real as output, but it can only do this if the input is also non-recursive. This is importantly different to the coupled Turing machine (above) which can produce non-recursive output only if its coupled stream is non-recursive. In that case, the coupled stream is not considered part of the input and is thus the same across multiple computations. For example, if it is computing the halting function, then it takes different natural numbers and returns zero or one as appropriate, but the coupled stream which gives it this power is the same regardless of the input number. For infinite input Turing machines, however, all potential to create non-recursive output depends upon it having non-recursive input and thus no non-recursive functions on the natural numbers can be computed.

While this is technically a hypermachine (it computes functions that the standard Turing machine cannot), it is a very modest extension of computability to infinite or continuous inputs. It is used extensively in the field of *recursive analysis* and given its rather uncontroversial nature, as well as the fact that we can imagine its input as being finite but growing, the functions on the real numbers that it computes do seem to deserve to be called 'recursive'.

2.6. Infinitely many states

An *infinite state Turing machine* [31] is a Turing machine in which the set of states is allowed to be infinite. This also implies that there are infinitely many transitions, although only finitely many leading from any given state. This gives the Turing machine an infinite program, although in terminating computations only a finite amount of it is used. With the freedom of infinitely many states, it is easy to show that there is an infinite state Turing machine that computes any function from $\mathbb{N}$ to $\mathbb{N}$. It can do so simply by a kind of infinite look-up table. This is reminiscent of the relationship between finite state machines and finite functions: for any function with a finite domain, there is a finite state machine which computes it via a finite look-up table. In both cases the real work seems to be done in creating the look-up table and it is debatable as to what extent the machine is computing at all. This problem can be seen to arise for all those hypermachines which can compute every function from $\mathbb{N}$ to $\mathbb{N}$.

In the case of finite state machines, attention turned to their ability to compute functions on infinite domains and since lookup tables were no longer possible, an interesting theory emerged. Similarly, we could turn our attention to computing functions on an infinite domain such as $\mathbb{R}$, in which case infinite state Turing machines will only be able to compute a certain well defined subset of these functions and a non-trivial analysis of computation can arise.

[12] The approach discussed here is known as the Type-2 Theory of Effectivity (with Type-1 being the standard Turing machine model). See Weihrauch [43] for a comprehensive survey of such generalisations.

2.7. Neural networks

Recurrent neural networks are a well known biologically inspired model of computation. Each network is specified by a set of nodes (including input, output and internal nodes) with weighted links connecting them. If the weights are taken from the rational numbers, then recurrent neural networks are equal in power to Turing machines, computing the recursive functions. Alternatively, if arbitrary real numbers are allowed as weights, then the set of computable functions increases dramatically.

Siegelmann and Sontag [37,36] consider the power of such networks when applied to discrete input and binary output. It is found that within exponential time, they can compute any function from $\mathbb{N}$ to $\{0,1\}$. This can be done by encoding the function into one of the real valued weights and then reading off the appropriate digit. In polynomial time, however, the class of computable functions drops to what is known as P/poly: the class of Turing machines with a polynomial amount of oracular advice, operating in polynomial time. This is a much smaller class, but one that still includes non-recursive functions.

A common complaint about any model that uses continuous quantities, such as recurrent neural networks with real valued weights or certain implementations of o-machines, is that they require infinite precision measurement. This is half true. When the weight is used in the computation it only needs to be of finite accuracy, but the accuracy is dependent on the input, increasing without bound as the size of the input does. Just as we say that the Turing machine requires unlimited memory, not infinite memory, so too we should say that when accessing these real valued quantities, these models require unlimited precision, but not infinite precision. However, in order to avoid adding more information to the *specification* of the machine when it is run on larger and larger inputs, the values of its weights do need to be infinitely precise. Thus, such models require an infinite precision real numbered quantity with unlimited precision measurement of this quantity. Such a possibility might be consistent with our best physical theories,[13] but even so, it does rely heavily upon the idealisation of continuous quantities as real valued. As we shall see, the next model's resources are even more demanding for it requires infinite precision measurement as well, giving a different, non-continuous, flavour to its hypercomputation.

2.8. Infinite precision branching

Blum et al. [2] have introduced a model of computation over the real numbers (or any other ring). Their machines are presented as a generalisation of flowchart computation, consisting of the following parts:

(i) *Input node*: a linear map from the input space to the internal space.
(ii) *Output nodes*: linear maps from the internal space to the output space.
(iii) *Computation nodes*: polynomial maps from the internal space to itself.
(iv) *Branch nodes*: take the computation down one of two paths depending on whether a given polynomial over the internal space is less than zero or not.

The power of the model depends on the nature of the space over which it computes. A machine is said to compute over a given ring R and the input/output/internal spaces for the machine are of the form R^n or R^∞. When the model is used for computation over the integers, it is equal in power to the Turing machine. However, when computing over the real numbers, its power is radically enhanced. For one thing, the infinite precision branching allows it to compute discontinuous functions such as step functions. It can also solve the halting problem by having a real valued constant in its program which encodes a halting problem look-up table and then some computational apparatus to read off the appropriate value. Similar techniques allow such a machine computing over $\mathbb{R}$ to compute any function from $\mathbb{N}$ to $\mathbb{N}$.

We could also, however, consider a modification to the original type of machine in which only recursive polynomial maps were allowed. This would prevent non-recursive lookup tables being coded into the

[13] The quantum mechanical Heisenberg uncertainty principle does not explicitly disallow arbitrary precision measurements, it only insists that those measurements must involve a sacrifice in the precision of the conjugate quantities.

machine's constants and the machines would thus be weaker than those considered above. This change would also make the number of machines countable regardless of the ring that they are computing over. Thus the machines would be finitely specifiable and yet could still compute all the interesting non-recursive functions over the reals that the authors consider in [2].

3. Comparative powers

While most of the discussion of hypercomputational power so far has concerned the solving of the halting problem, the power of many hypermachines goes far beyond this. To compare the computational power of such hypermachines, we can look at which predicates on the natural numbers they can decide, or in other terms, which functions from $\mathbb{N}$ to $\{0,1\}$ they can compute. To do so, it is convenient to make use of the *arithmetical hierarchy* [27,23]. This hierarchy has its roots in the study of formulae of first order arithmetic. When these formulae have free variables, they can be considered as predicates over the natural numbers. For example $\exists x(x \cdot x = y)$ has y as a free variable and is true if and only if y is a perfect square. It can thus be thought of as the predicate *is-square*.

Many such arithmetical predicates are recursive, having Turing machines which can decide them. All other predicates expressible in first order arithmetic can be written as a recursive predicate preceded by a chain of alternating quantifiers. The number of alternations can then be used as a measure of the complexity of the predicate. If a predicate is expressible with n alternating quantifiers beginning with $\exists$, then the predicate is in the class Σ_n, while if the quantifier chain begins with $\forall$, the predicate is in the class Π_n. The intersection of Σ_n and Π_n is then denoted Δ_n while the entire set of arithmetical predicates is denoted Δ_ω. There is a close connection between computational power and the arithmetical hierarchy. The class of recursive predicates is Δ_1, while the class of recursively enumerable predicates (including the halting predicate) is Σ_1. Furthermore, the class of all functions computable with an oracle for the halting function is Δ_2.

Only a few classes of hypermachine have capabilities that are not expressible in terms of the hierarchy. The infinite time Turing machines can compute all predicates in the arithmetical hierarchy as well as those in its transfinite extension. They can even compute some functions within the *analytic hierarchy*, which resembles the arithmetical hierarchy but is applied to second order arithmetic. Also, when restricted to polynomial time, recurrent neural networks compute those functions belonging to the class P/poly. This class includes non-recursive functions, but also lacks many recursive functions. It is thus not easily comparable with the other classes here.

Model	Power
Turing machine	Δ_1
Accelerating Turing machine	Σ_1
Infinite time Turing machine (in $<\omega \cdot n$ timesteps)	Δ_n
Infinite time Turing machine (in $<\omega \cdot \omega$ timesteps)	Δ_ω
Infinite time Turing machine (in arbitrary time)	$\Sigma_1^1 \cup \Pi_1^1 <$ power $< \Delta_2^1$
Malament–Hogarth machine	up to Δ_ω
Fair non-deterministic Turing machine	Σ_1
Infinite state Turing machine	All
Recurrent neural network (in exponential time)	All
Recurrent neural network (in polynomial time)	P/poly
BSS Machine (over $\mathbb{R}$)	All
Oracle based machine (o-machine, coupled TM...)	Varies with set/stream
Turing machine with random coupled stream	Varies with set/stream

References

[1] Ralph M. Blake, The paradox of temporal process, Journal of Philosophy 23 (1926) 645–654.
[2] Lenore Blum, Mike Shub, Steve Smale, On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines, Bulletin of the American Mathematical Society 21 (1989) 1–46.

[3] Selmer Bringsjord, A new Gödelian argument for hypercomputing minds based on the busy beaver problem, Applied Mathematics and Computation, in press, doi:10.1016/j.amc.2005.09.071.
[4] Alonzo Church, An unsolvable problem of elementary number theory, American Journal of Mathematics 58 (1936) 345–363.
[5] B. Jack Copeland, The broad conception of computation, American Behavioral Scientist 40 (1997) 690–716.
[6] B. Jack Copeland, Even Turing machines can compute uncomputable functions, in: Cristian Calude, J. Casti, M.J. Dineen (Eds.), Unconventional Models of Computation, Springer-Verlag, Singapore, 1998, pp. 150–164.
[7] B. Jack Copeland, Turing's o-machines, Penrose, Searle and the brain, Analysis 58 (1998) 128–138.
[8] B. Jack Copeland, Narrow versus wide mechanism, Journal of Philosophy 96 (2000) 5–32.
[9] B. Jack Copeland, Accelerating turing machines, Minds and Machines 12 (2002) 281–301.
[10] B. Jack Copeland, Hypercomputation, Minds and Machines 12 (2002) 461–502.
[11] B. Jack Copeland, Richard Sylvan, Beyond the universal turing machine, Australasian Journal of Philosophy 77 (1999) 46–66.
[12] E. Brian Davies, Building infinite machines, British Journal for Philosophy of Science 52 (2001) 671–682.
[13] Martin Davis, The myth of hypercomputation, in: C. Teuscher (Ed.), Alan Turing: Life and Legacy of a Great Thinker, Springer, Heidelberg, 2004, pp. 195–212.
[14] John Earman, Bangs, Crunches, Whimpers and Shrieks—Singularities and Acausalities in Relativistic Spacetimes, Oxford University Press, Oxford, 1995.
[15] John Earman, John D. Norton, Infinite pains: The trouble with supertasks, in: A. Morton, S.P. Stich (Eds.), Benacerraf and his Critics, Blackwell, Cambridge, MA, 1996, pp. 231–261.
[16] Gábor Etesi, István Németi, Non-Turing computations via Malament–Hogarth space-times, International Journal of Theoretical Physics 41 (2002) 341–370.
[17] Robin O. Gandy, On the impossibility of using analogue machines to calculate non-computable functions, 1993. Unpublished.
[18] John Gill, Computational complexity of probabilistic Turing machines, SIAM Journal of Computing 6 (1977) 675–695.
[19] E. Mark Gold, Limiting recursion, Journal of Symbolic Logic 30 (1965) 28–48.
[20] Joel D. Hamkins, Infinite time Turing machines, Minds and Machines 12 (2002) 521–539.
[21] Joel D. Hamkins, Andy Lewis, Infinite time Turing machines, Journal of Symbolic Logic 65 (1998) 567–604.
[22] Mark L. Hogarth, Non-Turing computers and non-Turing computability, PSA 1 (1994) 126–138.
[23] Hartley Rogers Jr., Theory of Recursive Functions and Effective Computability, McGraw-Hill, New York, 1967.
[24] Kevin T. Kelly, The Logic of Reliable Inquiry, Oxford University Press, New York, 1996.
[25] Tien D. Kieu, Quantum hypercomputation, Minds and Machines 12 (2002) 541–561.
[26] Stephen C. Kleene, General recursive functions of natural numbers, Mathematishe Annalen 112 (1936) 727–742.
[27] Stephen C. Kleene, Recursive predicates and quantifiers, Transactions of the American Mathematical Society 53 (1943) 41–73.
[28] Peter Kugel, Computing machines can't be intelligent (... and Turing said so), Minds and Machines 12 (2002) 563–579.
[29] Marvin Minsky, Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, NJ, 1967.
[30] István Németi, Gyula David, Relativistic beyond Turing computers, Applied Mathematics and Computation, in press.
[31] Toby Ord, Hypercomputation: Computing more than the Turing machine. Technical Report arXiv:math.LO/0209332, University of Melbourne, Melbourne, Australia, September 2002. Available at <http://www.arxiv.org/abs/math.LO/0209332>.
[32] Toby Ord, Tien D. Kieu, Using biased coins as oracles, 2005. Available at <http://arxiv.org/abs/cs.OH/0401019>.
[33] Hilary Putnam, Trial and error predicates and the solution of a problem of Mostowski, Journal of Symbolic Logic 30 (1965) 49–57.
[34] Tibor Rado, On non-computable functions, The Bell System Technical Journal 41 (1962) 877–884.
[35] Bertrand A.W. Russell, The limits of empiricism, Proceedings of the Aristotelian Society 36 (1936) 131–150.
[36] Hava T. Siegelmann, Neural and super-Turing computing, Minds and Machines 13 (1) (2003) 103–114.
[37] Hava T. Siegelmann, Eduardo D. Sontag, Analog computation via neural networks, Theoretical Computer Science 131 (1994) 331–360.
[38] Edith Spaan, Leen Torenvliet, Peter van Emde Boas, Nondeterminism, fairness and a fundamental analogy, EATCS Bulletin 37 (1989) 186–193.
[39] Ian Stewart, Deciding the undecidable, Nature 352 (1991) 664–665.
[40] James F. Thomson, Tasks and super-tasks, Analysis 15 (1954) 1–13.
[41] Alan M. Turing, On computable numbers, with an application to the entscheidungsproblem, Proceedings of the London Mathematical Society 42 (1936) 230–265.
[42] Alan M. Turing, Systems of logic based on the ordinals, Proceedings of the London Mathematical Society 45 (1939) 161–228.
[43] Klaus Weihrauch, Computable Analysis, Springer, Berlin, 2000.
[44] Philip D. Welch, On the possibility, or otherwise, of hypercomputation, British Journal for the Philosophy of Science 55 (2004) 739–746.
[45] Hermann Weyl, Philosophie der Mathematik and Naturwissenschaft, R. Oldenburg, Munich, 1927.