Nature-Like Computation and a Measure of Programmability

Hector Zenil

Abstract. I will propose an alternative behavioural definition of computation (nature-like) based on whether a system is capable of reacting to the environment—the input—as reflected in a measure of programmability. This will be done by using a phase transition coefficient previously defined in an attempt to characterise the dynamical behaviour of cellular automata and other rule-based systems. The transition coefficient measures the sensitivity of a system to external stimuli and will be used to define the susceptibility of a system to being (efficiently) programmed.

Keywords: Nature-like computation; programmability; cellular automata; compressibility; philosophy of computation; Turing universality.

1 APPROACHES TO THE QUESTION OF COMPUTATION

What is (a) computation? What does it mean to compute something and how much sense does it make to talk about computation outside of mathematics? These fundamental questions have not received a satisfactory answer yet, according to [27], and despite the well-known “Church-Turing thesis” (CT thesis).

The most important notion of computation, however, is the notion of digital computation, and its most important feature is programmability. Turing’s abstract idea of a universal computer has turned out to be technologically feasible, showing that if physics does not compute, it at least supports computation as we can build concrete devices whose behaviour, despite being governed by the laws of physics, effectively implement general-purpose digital computation. More formally, given a fixed description of Turing machines, we say that a Turing machine $U$ is universal if for all input $s$ and a Turing machine $M$, $U$ applied to $(< M >, s)$ halts if $M$ halts on $s$ and provides the same result as $M$ with input $s$, and does not halt if $M$ does not halt for $s$. In other words, $U$ is capable of simulating $M$ with input $s$, with $M$ and $s$ an arbitrary Turing machine and an arbitrary input for $M$.

So far the study of the limits of computation has succeeded in offering us insight into what computation might be. The borderline between the decidable and the undecidable has provided an essential intuition in our search for a better understanding of computation. One can, however, wonder just how much can be expected from such an approach, and whether other, alternative approaches to understanding computation may complement the knowledge and intuition it affords, specially in modern uses of the concept of computation in the context of nature and physics corresponding to situations in which objects or events are seen as computers or computations.

One such approach involves not the study of systems lying “beyond” the uncomputable limit (also called the Turing limit), but rather the study of the minimum requirements for reaching universal computation, through a focus on the ‘smallest’ possible systems capable of universal computation—how easy or complicated it is to build a universal Turing machine, and how efficient such a machine is. This minimalistic bottom-up approach is epitomised by Wolfram’s programme [32] together with its interesting older [25, 28, 22, 50] and more recent developments [33, 4, 29, 15, 20, 10]. The question behind is what enables universality in a computational setup and how pervasive may be? Does it originate from a rich supply of basic operations? As Wolfram [33] has long claimed (and captured in his intuitive Principle of Computational Equivalence), and as more recently Davis [10] has adopted, it takes very little to reach universality.

One can think of formal semantics as another approach to defining computation through programming languages and models of computation, which makes a distinction between syntax and semantics, mapping programs onto mathematical objects describing the relationship between the syntax and the model of computation (hence being model dependent). According to the semantics approach, therefore, a computation is a function that maps input onto output [17]. Syntax defines the correct form for valid programs and semantics determines what (if anything) they compute. In other words:

Computation = PL Syntax + PL Semantics

With PL meaning programming language. There are several widely used techniques (e.g. algebraic, axiomatic, denotational, operational, and translational) for the description of the semantics of programming languages, all of which deal with their behaviour. The distinction is often made between syntax, concerned with what constitutes a program or a computation, and semantics, concerned with the question of what a program computes, or what an expression means and whether two expressions are equivalent. And it is assumed that whatever provides an answer to the syntactical question is a syntactical model.

In most accounts of computational processes as realised by physical mechanisms, it is also often assumed that there is a one-to-one correspondence between the causality of the physical states and the states of a computation, as defined by some abstract model in which these can be represented. The traditional mapping-states definition of physical computation is probably inspired by formal semantics, in that it requires that a mapping be established between a model and a physical system, meaning that states and events in the model are used

---

1 Invited Talk, Symposium on Natural/Unconventional Computing and its Philosophical Significance, AISB/IACAP World Congress 2012 - Alan Turing 2012.
2 Behavioural and Evolutionary Theory Lab, Department of Computer Science, The University of Sheffield, UK, email: h.zenil@sheffield.ac.uk
to label states and events observed in the system treated as mathematical objects. Nature, however, is not like standard computation. One cannot, for example, easily assign a meaning to a natural phenomenon to be mapped to the concept of a halting state, nor is it always known what path nature has taken to produce a given outcome, regardless of whether we see this path as constituting a computation or not.

Usually in computation a system is prepared in an initial state, and is allowed to evolve through a trajectory of events occurring within the space of successive states, until it eventually reaches a state labeled as final. In nature, however, there is no such thing as a final state; everything is part of a causal chain of other events, much more like cellular automata that do not naturally halt (other than in reaching some stable configuration, for example) unless they are arbitrarily stopped.

One can define a computational process \( S \) is universal if it can simulate any other computational process \( \mu \). In other words, \( S \) is universal if for any other computational process \( \mu \), we can find an easy encoding protocol \( C \) and decoding protocol \( C^{-1} \) so that we can encode any input \( x \) for \( \mu \) as an input \( C(x) \) for \( S \) so that after \( S \) has performed its computation we can decode the answer \( S(C(x)) \) to the answer that \( \mu \) would have given us. In symbols: \( C^{-1}(S(C(x))) = \mu(x) \).

This syntactic approach to defining computation is closely related to the common view that computation is information processing. But from the point of view of our behavioural approach, the syntactic approach falls short of achieving its own objective, viz. distinguishing what computation is from what it is not, because it is not at all clear how one can or cannot find a mapping between the states of any given (even natural or physical) system and a computational one in the broad contexts we are interested in. The semantic approach, however, appears to accommodate some common intuitions about what goes and does not count as a computing system, but it requires that relations be established between computational states. For example, it can be made to work by introducing the notion of an interpreter. If an interpreter exists to transform what is in a system into a set of instructions to be executed by a computer, then the system can be said to compute. This, however, is not an easy task when it comes to making such mappings between natural systems and, say, models of computation (or other natural systems).

David Deutsch [8], for one, has often claimed that the theory of computation has traditionally been studied almost entirely in the abstract, as a topic in pure mathematics. Deutsch argues that computers are physical objects, and computations are physical processes, hence both computers and computations are governed by the laws of physics and not by pure mathematics.

Computation has, however, traditionally been defined in terms of mathematical functions or in terms of how a function is calculated. This has motivated to view computation either as the outcome of a mathematical function or as the study of the time that an algorithm takes to compute a function, which has been evidently extremely successful. Nevertheless, a purely behavioural definition of computation (and of a computer) in terms of whether a system is capable of reacting to the environment—the input—and proceeding to do something with it, may provide a definition that focuses on whether a system is capable of (efficiently) transferring information from its input to its output, which is in a strong sense what it means to program a system. It is this capacity for efficiently transferring information which will serve to indicate the system’s susceptibility to being programmed. Clearly, by this definition one may not call something a computer if it takes in the input but leaves it unchanged, or if for any input one gets always the same output, but my claim is that between these two cases there is room for a behavioural definition. It will be, thus, whether one can program a system what makes it a computer.

## 2 A BEHAVIOURAL APPROACH TO NATURE-LIKE COMPUTATION

Significant effort has been invested in definitions of computation in denotational, operational and axiomatic terms. For example, most approaches prove that a computation and its object denotationally coincide (leading to the CT thesis), some have adopted operational approaches [7] with questions of whether their definitions are just too broad. The axiomatic approach has also been developed with some interesting results [11, 19]. Nevertheless, some authors have extended the definition of computation to physical objects and physical processes at different levels of physical reality [31, 9, 10, 33, 8, 24] ranging from the digital to the quantum. In [33], for instance, Wolfram states that “…all processes, whether they are produced by human effort or occur spontaneously in nature, can be viewed as computations.”

Klaus Sutner [21] has this to say in regards to Wolfram’s conception of computation in nature: “This [Wolfram’s] assertion is not particularly controversial, though it does require a somewhat relaxed view of what exactly constitutes a computation—as opposed to an arbitrary physical process such as, say, a waterfall.” However, the work of several of the aforementioned physicists and computer scientists does indeed permit us to claim that a waterfall is (or can be viewed as) a computational process.

Whether one regards the universe as performing a computation or all natural processes as computations, when something is identified in a particular way because it has a specific property, the aim is to construct a category that includes certain things which share that property and exclude those things that do not, so that one can distinguish one thing from another and claim that one has established a concept with a finite extension which is set apart within the domain of discourse.

But to make sense of the term “computation” in these contexts (modern views of physics), I propose a behavioural notion of nature-like computation (similar in spirit to the way the term physics-like computation has been coined [24, 21]) compatible with digital computation but meaningful in broader contexts independent of representations and possible carriers. This will require a measure of the degree of programmability of a system by means of a compressibility index ultimately rooted in the concept of algorithmic complexity. I ask whether two computations are the same if they look the same and I try to answer with a specific tool possessing the potential to capture a notion of qualitative behaviour.

The fact that we need hardware and software is an indication that we need a programmable substratum that can be made to compute something for us but Turing’s main contribution vis-à-vis the concept of computational universality is that data and programs can be stored together in a single memory without any fundamental distinction. One can always write a specific-purpose machine with no input to perform any computation, and one can always write a program describing that computation as the input for a (universal) Turing machine, so in a strong sense there is a non-essential distinction between program and data. This is crucial, in that the same void distinction holds between hardware and software, as software can be seen as both data and program, and hardware can always be emulated in a program (even if it may appear obvious that hardware is ultimately needed to undertake a computation).

A programmer uses memory space and cpu cycles in a regular
computer to perform a computation, but this is by no means an indication that computation requires a computer (say a PC), only that it needs a substratum. The behaviour of the substratum is the underlying property that makes something a computation, and what carries out the computation a computer.

The behavioural approach takes this abstraction from the substratum to the extreme (keeping it physical as opposed to mathematical), with its central question being whether one can program a system to behave in a desired way. This approach that bases itself on the extent to which a system can be programmed tells us to what degree a given system resembles a computer. It can therefore serve as an epistemological framework for interpreting the computational nature of a system in the broader modern sense of computation, particularly in a physical context.

As suggested by Sutner [21], it is reasonable to require that any definition of computation in the general sense, rather than being a purely logical description (e.g. in terms of recursion theory), should capture some sense of what a physical computation might be. While Sutner’s suggestion [21] has similar motivations to ours, it differs from ours in that his aim is to map the behaviour of a system to the theory of computation, notably computational degrees. Sutner aligns his approach with his reading of the following claim made by Searle: “Computational states are not discovered within the physics, they are assigned to the physics.” Sutner adds “A physical system is not intrinsically a computer, rather it is necessary to interpret certain features of the physical system as representing a computation.”

This obliges Sutner to take into consideration the act of interpretation of a physical system and the observer. Sutner’s observer’s language maps the physical object to an interpretation of what the object does as a computational process. In Sutner’s view the observer may in the process of interpretation slightly modify the computation without adding to or carrying out the computation attributed to the physical object. One can see Sutner’s model as consisting of a pair of coupled automata, where one is the physical object and the other the observer. The observer is defined as an automaton constrained in computational power, capable of mapping (interpreting)—by way of a transducer—a physical object onto a computational process using electrical signals.

Sutner’s approach [21] is dependent on aspects of the traditional theory of computation in that it requires a mapping, and strong assumptions are made as regards the physical object, the observer and the mapping itself. We don’t focus on these mappings but on the qualitative behaviour of a system, regardless of whether the mapping is known, can be known or even exists, although such a mapping should in principle exist under certain (strong) assumptions, but it only cares about the qualitative character of a computational process and not its inner workings.

We know that systems that nobody ever designed as computers are able to perform universal computation, for example Wolfram’s Rule 110 [33.4], and that this like other remarkably simple systems are capable of universal computation (e.g. Conway’s Game of Life or Langton’s ant). These systems may be said to readily arise physically, or conceived or designed. They are not deliberately designed. There is, however, no universal agreement as regards the definition of what a computer may or may not be, or as to what exactly a computation might be, even though what computation is and what a computer may be are well grasped on an intuitive level.

A program can be defined as that which turns a general-purpose computer into a special-purpose computer. This is not a strange definition, since in the context of computer science a computation can be regarded as the evolution undergone by a system when running a program. However, while interesting in itself, and not without a certain affinity with our approach, this route through the definition of a general-purpose computer is a circuitous one to take to define computation. For it commits one to defining computational universality before one can proceed to define something more basic, something which ideally should not depend on such a powerful (and even more difficult-to-define) concept. Universal computation is without a doubt the most important feature of computation, but every time one attempts to define computation in relation to universal computation, one ends up with a circular statement [computation is (Turing) universal computation], thus merely leading to a version of a CT thesis.

It encompasses minds and computers while excluding almost everything else, investing minds and computers with a special status while viewing most of the rest of reality as computationally vacuous. I think this approach is weak, however. Think of the billiard ball computational model. It is designed to perform as a computer and can therefore be trivially mapped onto the states of a digital computer. Yet it is a counterexample of what the semantic account sets out to do, viz., to cordon off minds and computers (believed capable of computation) from things like billiard balls, tables and rocks (believed to be incapable of computation).

### 3 CASE STUDY: CELLULAR AUTOMATA

A cellular automaton (CA) is a computational model that has been shown to be an interesting object of study both as a computational device per se and for modelling all kinds of phenomena [43][18]. A CA consists of an array of cells where each takes a value from a finite set of states. Every cell updates its value depending on the state of its neighbouring cells. Hence the global behaviour of the automaton depends on the local interaction of its cells.

But what does a CA compute? As shown by Wolfram [33], the evolution of a system like a cellular automaton can be viewed as a computation. As shown in [33] (page 638), ECA Rule 132 (R132) is a simple cellular automaton whose evolution effectively computes the remainder after division of a number by 2. Starting from a row of n black cells, 0 black cells survive if n is even, and 1 black cell survives if n is odd. So in effect this cellular automaton can be viewed as computing whether a given number is even or odd. Wolfram provides other CA examples computing functions in the traditional sense (e.g. R94 as enumerating even numbers; R62 that can be thought of as enumerating numbers that are multiples of 3; the central column of the pattern of R129 that can be thought of as enumerating numbers that are powers of 2; or a CA, with 16 states, as capable of computing prime numbers).

The CA community has developed a strong intuition for determining the ability of a CA to transmit information and eventually be considered a candidate for universal computation. Evident properties of rules like the game of life [5] (a 2-dimensional cellular automaton proven to be computationally universal) and of rules like R110 [22], (a one-dimensional nearest neighbourhood) simple cellular automata, are structures persisting over time but sensitive to perturbations. These structures transmit information through a system, for example, in the form of characteristic gliders and all sorts of other well-known structures. These structures are unpredictable in a fundamental way if the system is capable of universal computation (as we will learn below from the work of Gödel and Turing). Predictable rules, or rules with no persistent structures, are often discarded as incapable of carrying messages and behaving as universal computers. Nevertheless, CAs computing in one-dimensional space, with only 2 states and nearest neighbour have already sufficient internal rich-
ness, in spite of this simplicity, to simulate a cyclic tag system for implementing a universal computing device [13,33]. Wolfman noticed [33] this richness, and by careful visual inspection of the evolution of two-dimensional space-time orbits, he was able to classify all the various behaviours into 4 general classes for systems starting with a random initial condition. Wolfman provided a 4-group classification of behaviour (particularly for cellular automata, specially the so-called elementary i.e. 1-range neighbourhood). His classes can be regarded as reflecting how information from the initial state is retained in the final configuration in a system (e.g. a cellular automaton). Class I, for example, is either unable to transfer any information to future states or simply transfers all or a portion of the information exactly as it came in. For Class II, however, information always remains completely localised into rigid patterns (e.g. fractals). On the other hand, Class III can be seen as scrambling the information from the input, allowing little chance to recover the information from the output because it generates a sort of noise (what Wolfram calls intrinsic randomness) even from the simplest inputs (e.g. a single black cell). Class IV, however, transfers information from the input through the system, interacting with other structures, but neither unfolding into simple structures such as those in Class II nor scrambling the information as happens in Class III.

A measure based on the change of the asymptotic direction of the size of the compressed evolutions of a system for different initial configurations (following a proposed Gray-code enumeration of initial configurations) was presented in [34]. It gauges the resiliency or sensitivity of a system vis-à-vis its initial conditions. This phase transition coefficient led to an interesting characterisation and classification of systems, which when applied to elementary CA, yielded exactly Wolfram’s four classes of systems behaviour, with no human intervention. The coefficient works by compressing the changes of the different evolutions through time, normalised by evolution space, and it is rooted in the concept of algorithmic complexity.

3.1 A measure of programmability

Based on the principles of algorithmic complexity, one can use the result of the compression algorithms applied to the evolution of a system to characterise the behaviour of a system [34] by comparing it to its uncompressed evolution as it is captured in eq. [1]. If the evolution is too random, the compressed version won’t be much shorter than the length of the original evolution itself. It is clear that one can characterise systems by their behaviour [34]: if they are compressible they are simple, otherwise they are complex (random-looking). The approach can be taken further and used to detect phase transitions, as shown in [34], given that one can detect differences between the compressed versions of the behaviour of a system for different initial configurations. This second measure allows us to characterise systems by their sensitivity to the environment: the more sensitive the greater the variation in length of the compressed evolutions. A classification places at the top systems that can be considered to be both efficient information carriers and highly programmable, given that they react succinctly to input perturbations. Systems that are too perturbable, however, do not show phase transitions and are grouped as inefficient information carriers. The efficiency requirement is to avoid what is known as Turing tarps [14], that is, systems that are capable of universal computation but are actually very hard to program. This means that there is a difference between what can be achieved in principle and the practical ability of a system to perform a task. This approach is therefore sensitive to the practicalities of programming a system rather than to its potential theoretical capability of being programmed.

The transition coefficient is derived from a characteristic exponent and is defined as follows: Let the characteristic exponent \( c_t \) be defined as the mean of the absolute values of the differences between the compressed lengths of the outputs of the system \( M \) running over the initial segment of initial conditions \( i_j \) with \( j = \{1, \ldots, n\} \) following the numbering scheme devised in [34] based on the Gray-code, and running for \( t \) steps in intervals of \( n \). Formally,

\[
    c_t = \frac{|C(M(i_1)) - C(M(i_2))| + \ldots + |C(M(i_{n-1})) - C(M(i_n))|}{n(t - 1)}
\]

Let \( C \) denote the transition coefficient defined as \( C(U) = f(S_r) \), the derivative of the line that fits the sequence \( S_r \) by finding the least-squares as described in [34] with \( S_r = S(c_t^r) \) for a chosen sample frequency \( n \) and running time \( t \). \( S(c_t^r) \) is simply a sequence of \( c_t^r \) for increasing \( t \) and fixed \( n \). That is, to capture the asymptotic behaviour of \( M_t \). The value \( C(U) \) (simply \( C \) until the discussion of definitions in the next section) will be therefore an indicator of the degree of programmability of a system \( U \) relative to its external stimuli (input). The larger the derivative, the greater the change of \( U \) and therefore the possibility of program \( U \) to perform a task encoded in some form.

For example, according to this coefficient (or index) \( C \), cellular automata (CA) with rule numbers 0 and 30 are close to each other because they remain the same despite the change of initial conditions (despite the choice of \( t \) and \( n \)), and they are hardly perturbable. The measure indicates that rules like rule 0 or rule 30 (denoted from now on as R0, R30, etc.) are incapable of transmitting information, given that they do not react to changes in the input. In this sense they are alike because there is no change in the qualitative behaviour of these CA when fed with different inputs, regardless of how different the inputs may be—and this is what \( C \) measures. Rule 0, for example, remains entirely blank, while R30 remains mostly random-looking, with no apparent emergent coherent propagating structures (other than the regular and linear pattern on one of the sides).

On the other hand, rules such as 122 and 89 have \( C \) close to each other because they are sensitive to initial conditions. As is shown in [34], they are both highly sensitive to initial conditions and present phase transitions which dramatically change their qualitative behaviour when starting from different initial configurations. This means that rules like 122 and 89 can be used to transmit information through the system, from the input to the output.

Values of \( C \) for the subclass of CA referred to as elementary (the simplest one-dimensional closest neighbourhood, also known as ECA [33]) have been calculated and published in [34], and a further investigation of the relation between this transition coefficient and the computational capabilities of certain known (Turing) universal machines has been undertaken in [35]. We will refrain from exact evaluations of \( C \) to avoid distracting the reader with numerical approximations that may detract from our particular goal in this paper. The aim here is to propose a behavioural definition of computation based on this measure rather than to evaluate specific values that have already been calculated in [36].

This transition coefficient will be used to dynamically define computation based on the degree of programmability of a system. The advantage of using the transition coefficient \( C \) is that it is indifferent to the internal states, formalism or architecture of a computer or computing model; it doesn’t even specify whether a machine has to be digital or analog, or what its maximal computational power is. It is only based on the behaviour of the system in question. It allows us to minimally characterise the concept of computation on the basis of
behaviour alone.

Figure 1. ECA R255 (equivalent by colour inversion to R0, R255 is used here for visual convenience) is stuck, unable to perform any computation— it does not react to any external stimulus. This is an illustration of a C-computer for C close (or equal) to zero \[34\]. The picture shows a series of evolutions for 12 random inputs (3 per row) next to the cellular automaton rule (top).

Let’s denote as a C-computer (see Fig. 3.1) a system with programmability coefficient C capturing the capability of the system to transfer information from its input towards its output. Under this notation, R255 in Wolfram’s one-dimensional elementary cellular automata (ECA) enumeration (Fig. 1), for example, is a 0-computer, that is a computer unable to carry out any operation because it cannot transfer any information from the input to the output (another way to say this is that R255 does not compute). ECA R255 cannot by any means be programmed to perform any task, despite the input. We have then captured the sense of what it means not to be a computer with the following definition:

**Definition 1.** A 0-computer is not a computer in any intuitive sense because it is not capable of carrying out any calculation.

Figure 2. ECA R110 is efficient at carrying information through persistent local structures through the output reacting to external stimuli. Its \(C^*_n\) value for sensible choices of \(t\) and \(n\) \[34\] is compatible with the fact that it has been proven that R110 is capable of universal computation (it has been proven \[33, 4\] for a particular semi-periodic initial configuration).

A system capable of (Turing) universal computation (see Fig. 3.1) would therefore have a non-zero \(C\) limit value. \(C\) also captures some of the universal computational efficiency of the computer in that it has the advantage of capturing not only whether it is capable of reacting to the input and transferring information through its evolution, but also the rate at which it does so. So \(C\) is an index of both capability in principle and ability in practice. A non-zero \(C\) means that there is a way to codify a program to make the system behave (efficiently) in one fashion or another, i.e. to be programmable. Something that is not programmable cannot therefore be taken to be a computer.

One can also see that things that seemed to behave like computers but were not called computers can indeed be considered computers under this approach. Mathematical functions, for example, can be considered C-computers for some C determined by the domain of the function. That a function can be considered a computer does not controvert the theory wherein a computer is defined in terms of a function and a domain, and a function in terms of an algorithm having the input as its arguments and the output as its function evaluation. A function, however, seems to require a carrier. Usually that carrier is a piece of paper and a pencil being wielded by a person, but it can also be a physical computer. Can the simple description of the function be considered a computer or a C-computer? I think it should not be. Something static shouldn’t be considered to have a behaviour, and I think it can be captured by C. To evaluate C one needs to actually run a program, otherwise it remains unevaluated.

This makes for a clear distinction between, for example, a vision of the universe as a mathematical structure and a vision of the universe as a computer. While the latter may account for the physical carrier, implying that the computation is being carried out by the universe itself, it does not seem clear how a mathematical structure can come equipped with the carrier on which it should be executed, unless it becomes a computer program and therefore a computer.

Figure 3. ECA R4 is a kind of program filter that only transfers bits in isolation (i.e. when its neighbours are both white). It is clear that one can perform some very simple computations with this automaton. One could not, for example, implement a typical logic gate based on its particular behaviour. It cannot clearly carry (Turing) universal computation. It has a low \(C\) for random chosen \(n\) and \(t\) \[34\].

Toy computers (e.g. Fig. 3) can also be considered C-computers, as indeed can everyday things like fridges or lamps. When one turns on a lamp the lamp is programmed to do something, in this case to turn on. Likewise when it is turned off. Even if trivial, it reacts to the input by producing light as the outcome. A fridge can be seen as cooling objects that are introduced into it, the output being the cooling—after an interval—of the objects in question. That both a lamp and a fridge can be viewed as C-computers for a very limited C, given that they have limited programmability (to perform a single, specific
task), should not be surprising, at least not in light of the definition of a $C$-computer. With the advantage that one can now ask whether a lamp or a fridge is or isn’t a computer without trivialising either the question or the answer. Under our formal and precise definition they are, as long as it is stated that they are limited in scope, as indicated by their behaviour as captured by the coefficient $C$, while an ordinary static table may be some kind of $C$-computer, certainly for $C$ very close to 0, if it is thought to be computing anything at all. On the other hand, the universe as a whole can now legitimately be seen and treated in this context as a computer, as it is a $C$-computer for maximal $C$ given that it contains all possible $C$-computers.

3.2 Reversibility, 0-computers and conservation laws

In [23], Margolus asserts that reversible cellular automata can actually be used as computer models embodying discrete analogues of classical notions in physics such as space, time, locality and microscopic reversibility. He suggests that one way to show that a given rule can exhibit complicated behaviour (and eventually universality) is to show (as has been done with the game of Life [5] and R110 [33]) that “in the corresponding ‘world’ it is possible to have computers” starting these automata with the appropriate initial states, with digits acting as signals moving about and interacting with each other to, for example, implement a logical gate for digital computation. Wolfram reinforces this vision by suggesting, through his Principle of Computational Equivalence, that it is indeed the case that non-trivial behaviours inevitably lead to universal computation.

This does not mean that a system must necessarily be bijective (hence reversible) in its input/output mapping in order to be universal. But it is actually reversible CA with high entropy (number of possible states) which will tend to show the greatest behavioural richness and therefore be considered the best candidates for being classified as computers. In other words, the greater the richness a system is capable of, the greater $C$ coefficient it will have. A reversible CA (RCA) has the property that starting it from a random state is like starting from a maximum entropy state in a thermodynamical system, because the RCA is not allowed to get simpler in its evolution, the only way to get simpler being to collapse the number of states, making it irreversible. Entropy in a randomly initiated RCA can only increase, but if it reaches maximum entropy it can’t get any more complicated, and so nothing much happens. This is also captured by $C$, in that the RCA always look the same and are immune to evolutionary changes, presenting homogeneous local entropy everywhere.

RCA are interesting because they allow information to propagate, and in some sense they can be thought of as perfect computers—indeed in the sense that matters to us. If one starts an RCA from a non-uniformly random initial state, the RCA evolves, but because it cannot get simpler than its initial condition (for the same reason given for the random state) it can only get more complicated, producing a computational history that is reversible and can only lead to an increase in entropy. The RCA, however, is only reshaping the message that it got at the beginning in the form of an initial configuration, and so the amount of information in the RCA evolution remains the same. Which makes it a perfect example of a system with increasing entropy but consistent complexity over time. The algorithmic complexity of the RCA is the same because one can track the RCA back to the original information content represented by its initial configuration. So the state of the CA at any time always carries the same information content. In non-reversible CA, however, information can be lost, and even though the algorithmic complexity of the evolution of a CA is always the same, one cannot recover it after it has decayed from any later state. In reversible CA, entropy, like information content, may increase or decrease over time. As Margolus himself states, it is one thing to know that a gas was in one corner at a given state, and another to return the gas from its expanded condition to its original position. It may thus seem that RCA in Wolfram’s class III may all be chaotic, but Wolfram [32] offers examples of one-dimensional reversible cellular automata exhibiting three types of behaviour of local structures as they propagate in space.

In nature-like computation, conservation laws are important because the physical carrier on which a computation will be performed is governed by physical conservation laws (laws that conserve physical invariants such as mass, energy, momentum, etc.). In RCA, there are cases in which the simplest locally-computable invariants are cells whose values never change, and which are analogous to nature-like conservation laws. That is, laws such that for any given property, the physical state of the system does not change as the system evolves. The simplest RCA capable of doing this are those that ignore their neighbouring cells and only look at the central one, reproducing it identically. One may have doubts about calling these computers because there is no transformation of information whatsoever, with the system just letting pass through it anything that it is fed. Even worse, there are systems that may look as if they are computing the identity function while in fact performing a series of intermediate transformations which lead to the same output a few steps later. From the behavioural perspective based on the transition coefficient, under the qualitative definition the two would be behaving differently if they deliver their richness at different rates even if they produce the same output. This discussion helps us to see how close these computational systems are to physical phenomena and to purely behavioural descriptions, but also to address some potential concerns raised by the qualitative approach proposed herein.

4 PROGRAMMABILITY AND BEHAVIOURAL EQUIVALENCE

We can then define a system performing computation based on its behaviour simply as follows:

Definition 2. A system $U$ computes if $C_n(U) > 0$ for some $t, n > 0$.
Meaning that $U$ can be programmed. Whether $U$ can perform certain computations or all computations will not depend only on $C$ but on the details of $U$ that escape the behavioural definition. Yet this definition suits a much broader sense of nature or physics-like computation as used in, for example, modern models of physics (to mention but a few examples [18, 14, 16, 25]). One can see that there are systems that are not computers under this definition, simple ones such as R0 and R255 Elementary Cellular Automata (see Fig. 23). Notice that $C$ depends on two parameters, $t$ and $n$, from the original coefficient definition [1] indicating the number of steps that a system has run ($t$), and the sampling frequency ($n$). This is of course a downside of any a posteriori behavioural approach, and it is precisely what makes this empirical approach a difficult one. Nevertheless, one can do better and ideally define:

**Definition 3.** A system $U$ has programmability if
\[ \lim_{t \to \infty} C_{n=1}^t(U) = n \]

Meaning that the sampling frequency is $n = 1$ (i.e. the compression comparison is applied at every step for every initial condition at a time) and for all steps. Evidently this limit cannot be calculated in finite time ($t \to \infty$) by, say, a Turing machine. This is ultimately related to a problem of induction, that is, how to characterise the behaviour of a system that can start from a countable infinite number of possible states by looking only at a finite sample of them. If $O'$ is an oracle Turing machine, however, then $n=0$ can be computed, and fully describes the qualitative behaviour of $U$.

This means that equivalence in the theoretical sense is ultimately undecidable. In the empirical sense it can only be approached, given that the transition coefficient on which the qualitative definition of computation is based is limited by finite resources (reflected in the parameters $t$ and $n$), providing only an approximate indication of the behavioural programmability of a system.

Notice that this is consistent with the behavioural approach, because if two systems have about the same $C_{n=1}^t$, for $n$ and $t$ fixed it means that it does react to changes at about the same rate, so it may not only transfer or not information but if it does so or not it does so at the same rate if they both have the same $C_{n=1}^t$ for that $n$ and that $t$. By varying $n$ and $t$ one can also define rates of convergence to $C$ making a refinement to the original definition (perhaps a subject for a future continuation of this approach).

Clearly, under this definition, behaviour space is less dense than algorithm and program space because there may be different programs implementing different algorithms but leading to the same behaviour. So one can only define two behaviourally equivalent systems as follows:

**Definition 4.** A system $U$ and $U'$ are computationally equivalent in behavioural terms if $C(U) = C(U')$.

Simple examples of a behavioural computational class are $C$-computers for $C = 0$, i.e. they cannot be programmed, and are behaviourally equivalent. Under Def. 1 and 2, systems that are identified as $0$-computers do not compute, as they are not capable of being programmed.

Experience tells us that something that behaves in a certain way will continue doing so, as we have empirically established in [55]. This can be justified by algorithmic probability, because the longer the observation time of a computing system the smaller the chance that the behaviour in question will radically change. So even though one cannot guarantee a behaviour ad infinitum, algorithmic probability may provide the stability required to make reliable generalisations. So one can weak Def. 4 by allowing $C(U)$ to be close enough to $C(U')$ as follows:

**Definition 5.** A system $U$ and $U'$ are $c$ computationally equivalent if $|C(U) - C(U')| < c$.

It is worth stressing that two systems (or computers) are not the same in any other sense if they have the same coefficient $C$. $C$ is a measure of sensitivity (what I take as how programmable the system is); it cannot on its own indicate whether two computers compute the same function, and is therefore a different measure than that provided by traditional computability and formal semantics. It can tell when two computers diverge in their behaviour, because for two computers to be the same, a necessary but not sufficient condition is that they must both have the same transition coefficient (or to differ by a desired $c$), which would mean that they have the same capability of reacting to external stimuli, and transmit information at about the same rate. Because $C$ itself depends on two parameters ($n$ and $t$), this also means that $C$ can only make comparisons for fixed $t$ and $n$ (the same runtime and the same sampling frequency) between two systems. So two $C$-computers are behaviourally equivalent if they have the same $C$.

For the same reason that one cannot tell whether a machine will halt for a given input, one cannot decide whether two computers compute the same function, but one can relate nature-like computation and abstract computation by means of Turing machines as follows: for every $C$-computer $U$, there exists a program $P$ behaviourally equivalent to $U$, that is, with transition coefficient $C(U) = C(M)$ independent of $n$ and $t$, because there exists a universal Turing machine $U'$ capable of reproducing the exact behaviour of $U$.

It is worth also noting that this behavioural definition is cumulative (but not additive), in the sense that a $C$-computer can be embedded in the workings of another $C'$-computer for $C 
eq C'$. If the $C'$-computer does not impose any behavioural restriction on the $C$-computer, then clearly $C' \geq C$, given that the new computer will be capable of at least $C$-computation. This is the sense in which one may see R255 as a program in the context of a $C$-computer with $C 
eq 0$ capable of running R255. If the $C$-computer is, for example, a universal computer, R255 would be a program but cannot by itself be a computer.

### 5 DISCUSSION

The topic and content of Nature-like computation is, on purpose, related to the question of whether the universe can be said to compute. It does, for we know there are $C$-computers in it capable of universal computation, but we don’t really know whether the universe (e.g. as represented by its physical laws) constrains $C$, a limit broad enough to encompass every possible $C$-computer for a maximal $C$ contained in the physical universe. One can think of the law of gravitation as carrying out some kind of computation, with the degree of programmability of such a system limited to performing a particular task (in this case pulling objects toward each other and keeping them in their gravitational trajectory). Classical mechanics guarantees that the system is deterministic, even if that doesn’t mean one can predict the system for any specific parameters (e.g. 3 bodies). There is no fundamental reason, however, for following the approach described herein when assessing whether a system can compute based on its
degree of programmability. Still, the fact that one can coarse grain what computation may mean by way of the parameter \( C \), and guarantee that there are both systems with maximal \( C \) and \( C = 0 \) for systems that can be programmed to do something, and others that cannot be programmed at all and show no reaction to any external stimulus (e.g. see Fig. [5.7]), imbues this approach and its definition of computers and computation with sense, particularly in the context of nature-like computation as proposed by some of the aforementioned authors. There are also \( C \)-computers for small values of \( C \), meaning that the system can hardly be programmed because it does not transfer information efficiently enough (this may be the case, for example, with R30, see Fig. [5.7]).

5.1 The question of scale

So far, the object of this behavioural approach to computation has been to provide a reasonable framework for assertions connecting the notion of computation to nature, and how nature may or may not compute, in light of current uses of the term ‘compute’. Lloyd [23], for example, claims that since the universe is computing itself, things in the universe would therefore also be computing themselves. Think of the example of a still physical object (e.g. a desk or a sheet of paper). These objects would hardly compute anything at their macroscopic level, say an addition between any 2 numbers, yet they may be constituted at a molecular or atomic scale of particles capable of carrying out all sorts of computations, which unlike the objects, may be programmed, either as part of another system in or themselves. It is clear then that the span of behaviour at that scale is greater than at the scale of the object itself. But does it make sense to say that something computes itself? [23]. It may or it may not.

In the real world, things are constituted by smaller elements unless they are elementary particles. One therefore has to study the behaviour of a system at a given scale and not at all possible scales, otherwise the question becomes meaningless, as elements of a physical object are molecules, and ultimately atoms and particles that have their own behaviour, about which too the question about computation can be asked. This means that a \( C \)-computer may have a low or null \( C \) at some scale but contain \( C' \)-computers with \( C' > C \) at another scale (for which the original object is no longer the same as a whole). A setup in which \( C' \subseteq C \) is actually often common at some scale for any computational device. For example, a digital computer is made of simpler components, each of which at some macroscopic level but independently of the interconnected computer is of lower behavioural richness and may qualify for a \( C \) of lower value. In other words, the behavioural definition is not additive in the sense that a \( C \)-computer can contain or be contained in another \( C' \)-computer such that \( C \neq C' \).

Can R255, for example, be thought of as computing itself as it evolves? Under the qualitative definition, even if R255 is computing itself it cannot be programmed, and so is a 0-computer under our approach, a computer not capable of computation and therefore hardly a computer at all. On the other hand, R255 does not present any problem of scale as it represents itself at all scales. A table, however, is made of smaller components to which may be assigned some specific task, and one may even consider reprogramming the matter of which it is made, in the manner epitomised in the subfield of programmable matter. In which case one may say that the table is computing itself, since it could be computing something else out of its atoms. So the definition of a \( C \)-computer is scale dependent and its implementation in the real world is subtle, yet at the abstract level it seems to correspond to an interesting and well-delineated definition of computation based on its behavioural capabilities.

In the physical world, under this qualitative approach, things may compute or not depending on the scale at which they are studied. To say that a table computes only makes sense at the scale of the table, and as a \( C \)-computer it should have a very limited \( C \), that is a very limited behaviour given that it can hardly be programmed to do something else.

5.2 Program versus physical realisation

The behavioural approach may need a major shake-up if used in a quantum context, given that our understanding of the mechanisms at the quantum scale are subject to various interpretations. For example, the standard interpretation considers quantum mechanics to be fundamentally non-deterministic, and so our definition of a deterministic computer (necessary to evaluate \( C \)) becomes inapplicable. If quantum particles are capable of, for example, being in all possible states at the same time when entangled, that means that they can perform every possible computation at the same time (which is at the core of the quantum computational paradigm as based on the concept of the quantum bit or qubit and taking advantage of quantum properties). Hence it would obviously have a transition coefficient \( C' \) beyond any attained by a digital system, given that it would represent all possible behaviours at the same time—which in the macroscopic world would not be possible. If one takes atoms to be computers, or quantum computers, one can therefore trivially claim, as has been done by Lloyd [23], that the world is a quantum computer. In this case, the only content of such a claim, as opposed to its contrary (that the world is not a computer) concerns whether or not the world is a classical computer. Lloyd claims, as do Deutsch [8] and others, that it is not a classical computer, if it is a computer at all, but rather a quantum one, simply because computers, like everything else, rely on the very basic physical properties of our world.

When Gödel provided the proof of his incompleteness theorem what he did was to unify symbols and operators, just as Turing did for data and programs. Because Gödel’s and Turing’s approaches are extensionally equivalent, as long as one can find a Gödel numbering encoding a system, one can conclude that such a system can be interpreted as a program. For example, based on Davis’ work encoding Diophantine equations, it would seem that the extension of what a program can be is formally quite large.

The consequence of universal computation is that hardware and software are not essentially different, for one can be encoded in the other. But in the real world why are hardware implementations of software faster? (e.g. the case of Intel’s Pentium coprocessor: they could have certainly solved it with a patch, i.e. software, but that would perhaps have jeopardised the promise of a faster cpu). So are there real world differences between software and hardware, e.g. in execution time? It seems software always requires a transformational process— from a description to a physical embedding—in order to be executed. What makes a program, as a sequence of text, become a set of instructions that are executed? This is sometimes called the problem of computational implementation. The usual way to get round this problem is to separate programs from their physical implementations, on the grounds that the former are abstract while the latter are concrete, thus in some sense reinstating the difference between software and hardware when it comes to the physical world. At the fundamental level, however, and given that one can always (under Church-Turing’s thesis) implement a program, the difference is not essential in nature.

Physically, computer programs may be a collection of punched
cards or configurations in a magnetic tape. Is software part of a computer? If data and program can be exchanged one for the other, can software or hardware by themselves constitute a computer? Hardware alone may, if the computer is designed to serve a specific purpose, though it thereby loses its potential to reach computational universality. But what a purely software computer means may be unclear, and as suggested by Deutsch and Lloyd, the notion may make no sense. It may seem, for example, that the description of a Turing machine is pure software if no distinction is made between the input of the machine and its transition table (whether capable of universal computation or not). Is the difference only practical? Software seems not to have a physical execution carrier, and software before implementation may only be a description of a computation and not a computation per se, which means it cannot be executed in the real world until it finds a physical carrier. Is the description still a computer? I don’t think so, but I don’t aim to answer all these questions, and other authors have attempted to shed some light on such matters. The answer also seems related to a relativisation of software. Software written in a higher language like C or Mathematica is different from software written in machine language, and much closer to hardware.

The fact that we do need a computer for running, say, cellular automata rules, may be misleading, since it may suggest that there is always a need of hardware (and software) in the way we know and use it. This leads to the question of computation in nature, and ultimately to the question of the computer running the universe itself, if one embraces such an idea, and the associated question of whether such a computer, if it exists, is part of the universe or runs in some higher world.

The first thing to note is that the costs related to such a computer would be huge, on the order of the computational power of the universe itself, and likely to require even more than the energy in the universe itself, due to thermodynamical laws, if they apply at such a scale. This suggests that one need not look for a computer if one thinks that the computation comes equipped with its physical carrier or one would fall into an infinite hierarchy of worlds running each a program of a lower level universe. When the concept is at the same time matched and disentangled from its carrier in the behavioural approach, one can see that it is a particle at a time that creates the universe both carrying software and hardware together. As has been suggested, the universe would then be running itself and would be doing so rather efficiently and may be quite simple, yet there is no need to deny the physical carrier, which is simply the performer.

6 CONCLUDING REMARKS

I have proposed a novel qualitative notion of computation based in the sensitivity of a system to external stimuli connected to a concept of programmability. A notion I have called nature-like computation that provides a behavioural interpretation of computation (and of computers). This goes along current lines of technology for programming molecules and cells to compute, see for example Ref. [1]. This in some way can be seen as reprogramming a cell to do certain tasks that wasn’t supposed to do from their natural course. This is what in some way we have done with digital computers too, building machines out of natural matter to make them do calculations for us. All around a single concept, that of programmability, that I have suggested can be captured by a measure of behaviour rather than syntactic or even semantic approaches, given that the former requires descriptions of inner workings, even though we may not even fully understand the machinery of a cell, and the latter requires an interpretation of computation. The behavioural approach, however, is agnostic in most of these accounts, and it only cares about the qualitative behaviour of a system to transfer information by being stimulated. The concept also helps to give sense to current uses of computation in the context of natural phenomena, including the universe itself.

ACKNOWLEDGEMENTS

I would like to thank the organisers of the symposium Natural/Unconventional Computing and its Philosophical Significance for their kind invitation to talk at the AISB/IACAP World Congress 2012 – Alan Turing 2012.

REFERENCES