There Is No Such Thing As Miscomputation

Abstract

This paper will argue that there is no such thing as miscomputation, contrary to the received view in philosophy of computation. There are just hardware problems on the one hand and design errors on the other, neither of which qualify as a distinct kind of *computational* errors. The main upshot of this argument is that philosophical accounts of physical computation should not be assessed on whether they can accommodate miscomputation, but rather on whether they can make sense of the range of different phenomenona that are commonly (and misleadingly) described as miscomputations.

Introduction

There has been a recent renaissance in philosophical work on physical computation (see e.g. Piccinini 2007, 2015; Milkowski 2013; Fresco 2014), but rather less has been written on the topic of *mis*computation, i.e. the circumstance under which some physical system still counts as performing a computation, but somehow performs it in the wrong way. This is in spite of the fact that one of the leading accounts of physical computation, Gualtiero Piccinini's mechanistic account, lists miscomputation as one of six key desiderata that any account of physical computation must accommodate (2015: 13-14). Aside from Piccinini's own work, there have been articles published on (or relating to) miscomputation by Fresco & Primiero (2013), Dewhurst (2014), Floridi, Fresco, & Primiero (2015), Petricek (2017), Tucker (2018), Primiero, Solheim, & Spring (2019), and Colombo (forthcoming). In general each of these authors assume that there is a distinctive category of computational errors, but here I will argue that this is not the case, and that there is really no such thing as miscomputation *as such.* Section 1 will review existing accounts of physical computation and miscomputation, focusing primarily on mechanistic accounts. Section 2 will present the argument(s) that there is no such thing as miscomputation, but rather just various kinds of hardware malfunctions and design errors. Finally, section 3 will discuss the ramifications of this argument for existing accounts of physical computation, and consider some future work that could be done on the topic formerly known as miscomputation.

1 Physical computation and miscomputation

I am concerned here with *physical computation*, i.e. the question of what it means to say that a physical system performs or implements an abstract computation, and hence what it would mean to say that the same system somehow *fails* to implement the right computation, or miscomputes (note that a failure of this kind is distinct from simply failing to implement any computation at all). Before we can discuss miscomputation specifically I must say a little about physical computation in general, although a full treatment is beyond the scope of this paper. At a first pass we can say that a physical system implements an abstract computation if there is a mapping between the physical structure of the system and the formal structure of the computation (Putnam 1960, cf. Godfrey-Smith 2009). This is a notoriously weak definition, according to which almost any physical system might implement almost any computation (Putnam 1988; see Godfrey-Smith 2009 and Sprevak 2018 for discussions of this issue), and so it is typically strengthened with additional constraints on the kinds of physical system that qualify as computational. Examples include causal constraints (e.g. Chalmers 1994), semantic constraints (e.g. Sprevak 2010), simplicity constraints (e.g. Millhouse 2019), and pragmatic or perspectival constraints (e.g. Schweizer 2019), but I will focus here on a subclass of causal constraints offered by the various mechanistic accounts of physical computation, and specifically Gualtiero Piccinini's version of this account (2007, 2015; see also Milkowski 2013, Fresco 2014). My argument against the very possibility of miscomputation will mostly generalise to other accounts of physical computation, but I will note where it does not, and consider some of these other possibilities in the final section.

According to the mechanistic account, a physical computer is a kind of mechanism whose function is to perform computations, understood as systematic transformations between medium independent digits (2015: chapter 7).[1] Digits are (concrete, physically instantiated) components whose function is to be recognised and systematically transformed (into other digits) by processors, and are medium independent insofar as they are individuated only by those physical properties that are relevant to this function. Processors are (concrete, physically instantiated) components whose function is to identify and systematically transform digits according to a rule specified by the abstract computation that the system is meant to implement (i.e. the program). Further computational component-types include input and output components that transform external stimuli into digits and *vice versa*, memory components that store strings of digits, and so on. A computing mechanism will typically also include *non*-computational components such as a power source, a cooling fan, and so on. Crucially, the core components (digits and processors) must possess a sufficiently stable causal structure to qualify as computational, thus constraining the range of physical systems that will implement a computation (although for further discussion of some concerns about this account, see e.g. Dewhurst

[1]Note that this is specifically an account of *digital* computation. Piccinini does also offer related accounts of analog, generic, and *sui generis* neural computation, but for the sake of simplicity I will focus here just on the digital case.

2018; Coelho Mollo 2018, 2019; Fresco & Milkowski 2019). Examples of possible physical implementations of digits and processors includes voltage levels carried on metallic wires that are systematically transformed by transistors that can amplify or interrupt a voltage level (as in contemporary electronic computers); similar devices but with vacuum tubes instead of transistors (as in earlier twentieth century computers); toy marbles that can be physically directed around a 'circuit'; mechanical gears with different states, connected by rotating rods (as in Babbage's 19th century difference engine); and so on almost indefinitely. What matters in each case is the functional relationship between the digit and processor types, rather than their specific physical composition (at least beyond those aspects of their composition that are necessary for preserving the functional relationship).

Piccinini gives a list of six desiderata that he thinks any account of physical computation ought to be able to fulfil, including that it "should explain how it's possible for a physical system to miscompute [...] because miscomputation, or more informally, making computational mistakes, plays an important role in computer science and its applications" (2015: 14). I will return to the role played by miscomputation in computer science in section 3, focusing for now on Piccinini's own account of miscomputation. He defines miscomputation as cases where a system "fails to follow every step of the procedure it's supposed to follow all the way until producing the correct output" (*ibid.*), or more formally, "system M miscomputes just in case M is computing function f on input i, $f(i) = o_1$, M outputs o_2, and $o_2 \neq o_1$" (2015: 13), i.e. any case where a computing system outputs something other than that which was specified by the function it implements. His account is able to explain miscomputation due to its functional nature: if what it means for a physical system to perform a computation is to be a mechanism whose components have the function of performing that computation, then those components could also *malfunction*, resulting in a *miscomputation* (*ibid*: 122). An example of this kind of miscomputation is a hardware malfunction where e.g. a (physical) logic gate that is supposed to perform AND instead performs NAND, meaning that the digits are processed incorrectly relative to the rule specified by the (abstract) computation that is supposed to be implemented. Piccinini also allows for other kinds of miscomputation, including those resulting from incorrect design, implementation, or usage (*ibid*: 149-50). Here he is presenting a taxonomy that he first introduced in Piccinini (2007), which was then discussed by Fresco & Primiero (2013), and applied again to the mechanistic account by Dewhurst (2014). Floridi, Fresco, & Primiero (2015) further develop the taxonomy of miscomputation presented by Fresco & Primiero (2013) to include software malfunctions, Petricek (2017) specifically discusses programming errors, and Primiero, Solheim, & Spring (2019) provide an additional analysis of malware classification. Tucker (2018) develops Piccinini's functional approach to miscomputation by distinguishing between a system's *proper* function and its *actual* function, and Colombo (forthcoming) applies the notion of miscomputation to computational psychiatry. I will return to each of these issues as they arise naturally in the rest of the paper.

2 There is no such thing as miscomputation

In the previous section I introduced the taxonomy of miscomputations developed by Piccinini (2007, 2015), Fresco & Primiero (2013), and Dewhurst (2014). Let us now examine that taxonomy in more detail. Piccinini (2015: 149-50) lists five notions of miscomputation, each relating to a different perspective from which we might evaluate the performance of a computational system:

1. Miscomputations relative to the designer's intentions, i.e. when a computer is designed badly in the first place, such that it computes something other than what the designer intended.
2. Miscomputations relative to the designer's blueprint, i.e. when a correctly specified design is built incorrectly, such that it computes something other than the original designer intended and the blueprint specified.
3. Miscomputations relative to what was actually built, i.e. when a correctly designed *and* built computer undergoes a physical malfunction that prevents it from computing correctly.
4. Miscomputations due to incorrect programming, i.e. when a computer is designed, built, and (physically) functions correctly, but is programmed badly such that it computes something other than what the *programmer* intended.
5. Miscomputations due to incorrect usage, i.e. when a computer is designed, built, and programmed correctly, and functions adequately, but is used incorrectly by the end user, such that it does not compute what *they* intended it to.

Let us briefly consider an illustrative example of each kind of putative miscomputation. A type-1 miscomputation could occur when the design specification for a new computer indicates that some component ought to perform AND, but the actual wiring diagram specifies a component that would perform NAND. If this computer were built (correctly) according to this specification, it would 'miscompute' relative to the original designers intention. A type-2 miscomputation could occur if the original specification correctly described a component for perfoming AND, but due to errors in the manufacturing process the component that was actually built instead performed NAND. A type-3 miscomputation could occur if the component was both correctly specified and correctly built, but then overheated during operation, causing the wiring to melt in such a way that it now performs NAND instead of AND (as opposed to melting in such a way that it simply no longer computes, which would not constitute a miscomputation). A type-4 miscomputation could occur if the whole system was designed and built correctly, and didn't overheat, but then was programmed with badly written code that does not do what the programmer intended (due to a missing bracket, say). Finally, a type-5 miscomputation could occur if everything functioned correctly right up until the end user who, misunderstanding the interface of the device, entered what they thought was a multiplication and instead got the result for division – from the perspective of this user, the device might appear to be miscomputing. Each of these types qualifies as a performance failure from a certain perspective, but it is not clear that we should count all (or any) of them as genuine cases of miscomputation.

Fresco & Primiero draw a further useful distinction between "errors of functioning" and "errors of design" (2013: start of section 2), inspired by a similar distinction originally made by Turing (1950: 449). Errors of functioning, or what we might call 'operational malfunctions', occur when the system *as it was actually built* fails to function correctly, i.e. a physical component malfunctions in some way that affects the computational procedure. Piccinini's type-3 miscomputation is a clear case of operational malfunction, but all of the others seem to be of the latter type, which we might call 'design errors'. If a system is designed incorrectly (type-1), such that it cannot perform the function that it was intended for, then this is not a problem with the operation of the system itself, but rather a mistake on behalf of its designer. Similarly, if the initial design is good, but a token system is manufactured incorrectly (type-2), then this is not a case of the system itself malfunctioning, but rather a mistake on behalf of its proximal designer (the agent or system that manufactured it). A programming error (type-4) is also not a case of operational malfunction, as the physical system itself performs perfectly fine, it was just given bad instructions (at least relative to the programmer's intentions – for the system itself there is no sense in which the instructions can be good or bad). Finally, errors on behalf of the user (type-5) are clearly not operational malfunctions, as the system itself (both hardware and software) performs perfectly fine. In fact, it is not even clear that incorrect usage should be considered as a kind of design error either, but for the purpose of this analysis I will stretch the definition of 'designer' to include the end user of a computational system (who indeed might sometimes also be a programmer).

Piccinini suggests that we should consider all five of these notions to be types of miscomputation, even if only one of them (type-3) is the result of a malfunction of the computing mechanism itself (2015: 149-50). I will now argue that in fact *none* of them should be considered to be types of miscomputation, at least in a strict sense of miscomputation. To see a little more clearly what I have in mind by this strict sense of miscomputation, and why it might be beneficial to adopt it, I must first say a little more about malfunctions in general. A (functional) mechanism is usually understood to malfunction only if it fails to perform its function for 'internal' reasons, i.e. due to problems with its own constitutive structure, rather than with external circumstances that are 'outside of its control', so to speak (see e.g. Millikan 2013: 40; cf. Garson 2019: 127). This is meant to rule out cases where a mechanism such as the heart is working perfectly fine in terms of its intrinsic structure, but fails to perform its function of pumping blood around the body due to being in the wrong environment (outside of the body, perhaps), or due to being in an environment that is itself malfunctioning (due to a ruptured or blocked artery). In neither case should we say that the heart itself is malfunctioning, even if we might want to say that the broader mechanism within which it is (meant to be) embedded, i.e. the cardiovascular system, is indeed malfunctioning. Analogously, a computational malfunction, or miscomputation, can only take place due to an error internal to the computing mechanism itself, ruling out external errors due to e.g. its designer, programmer, or end user. But as we shall see in section 2.2, the function of computing is defined in such a way that changing the intrinsic structure of a computing mechanism will only ever *change what it is meant to compute* (provided that it is capable of computing at all), rather than causing it to miscompute. Therefore there is no such thing as miscomputation in this strict sense.

The definition of miscomputation introduced above is (self-admittedly) rather strict indeed, and before moving on I will say a little more to justify why we should prefer it to some looser definition, such as one according to which a system's failure to compute what a designer or programmer intends it to qualifies as a miscomputation. My main concern here has to do with laying the blame in the right place, which can also play a heuristic role in identifying where best to intervene when it comes to fixing a malfunctioning system (cf. Garson 2019: 114). If we simply refer to all of types 1-5 as miscomputations, then, without further disambiguation, it will remain unclear whether any given miscomputation ought to be resolved by intervening mechanically into the system itself, providing the user with better instructions, or rewriting the program that the system is meant to implement. Even restricting the term 'miscomputation' to type-3 errors is somewhat misleading, as in these cases the error is not really with the system *qua* computation, but rather with the physical hardware behaving in an unexpected or unintended way. To talk of miscomputations, without further specifying what is meant, is therefore relatively uninformative, and the requirement to give some further specification in each case removes the value of a general category of 'miscomputation', as these five types of error do not really have much in common. So we would do better to replace the term 'miscomputation' with several more specific categories of computational error – or so I will now argue.

2.1 Design errors are not miscomputations

Fresco & Primiero "argue that a computational system *can only* make an error of *functioning* (i.e. an operational malfunction)" (2013: section 1, emphasis in original), which would presumably mean that design errors are not miscomputations. However, they also have a somewhat broader conception of operational malfunctions than that which I introduced above, including certain kinds of software errors (Piccinini's type-4 miscomputations). So why shouldn't we consider design errors, including programming errors (cf. Petricek 2017) and software malfunctions, to be kinds of miscomputations?

Dewhurst (2014) previously offered an argument to this effect, focusing on the mechanistic account of computation introduced above. He argued that we should not assess the behaviour of a computing mechanism relative to the designer's *intentions*, but rather relative to their actual design, as implemented in the token system that we are analysing. This is because a computing mechanism is simply a system that transforms strings of digits *according to the rules that it is given* (i.e., that which is specified by its structure), without any understanding of either those rules or the semantic content of the digits. So a (physically) functioning computing mechanism provided with a program to run cannot fail to correctly implement that program, even if the end result is not what the programmer intended. For this reason I do not think that we should consider programming errors of any kind to be miscomputations, at least in the strict sense of computation captured by the mechanistic account.

Other kinds of design error, like user errors or errors in the physical manufacture of the system, are also not miscomputations in the strict sense defined above. Mistakes made by the end user of a computing mechanism, e.g. by issuing meaningless or misunderstood (by the user) commands, are clearly not cases of the system itself miscomputing. This might also include documentation errors, such that the user does not know how to use the system properly (this might not strictly be the users 'fault', but the end result is the same – a bad command is issued to a perfectly well-functioning system). Manufacturing errors might qualify as malfunctions of some kind, but these cannot be *computational* malfunctions (of the end product): either the token system that is the result of the faulty manufacturing process does not function as a computing mechanism at all, or else it does function as one, but not in the way that the designer intended. The latter case is similar to programming errors, insofar as the system will continue to compute *something*, and do so 'correctly' relative to its actual physical structure, even if this is different to how the designer originally envisioned it. Such an outcome might depend on malfunctions *in the manufacturing process*, but once this process is complete, the system thereby created cannot be blamed for these malfunctions. If it functions as a computing mechanism at all it simply computes according to the rules that it was given, i.e. those that are implicit in its actual physical structure.[2]

Software malfunctions, as discussed by Floridi, Fresco, & Primiero (2015), are also typically a kind of design error. Some apparent software malfunctions might be due to physical limitations, such as a logic gate functioning incorrectly or a memory component failing in some way, but it is debatable whether these really qualify as *software* malfunctions, rather than physical malfunctions that affect the software (*ibid*: 18 of preprint). In any case, I will deal with the question of physical ('operational') malfunctions in the next section. Other software malfunctions might be due to badly written code (missing brackets, mistyped variables, infinite loops), but as Floridi, Fresco, & Primiero themselves admit, malfunctions of this kind "are simply design errors for which only the designer can be deemed responsible" (*ibid*). A computing mechanism can only do the best with what it is given; a badly programmed system might do something that we don't want it to, but it would not be at fault for doing so, and hence there would be no miscomputation.

[2]Note that the identity of these rules might itself be ambiguous (cf. Sprevak 2010, Dewhurst 2018), but this is a distinct kind of issue that is not directly related to miscomputation.

Design errors of all kinds, including mistakes in the original specification of the computing mechanism, faulty manufacturing, bad or misguided programming, and erroneous usage, do not qualify as miscomputations, as the mistake in each case is attributable to something other than the computing mechanism itself (i.e. the machine's designer, manufacturer, programmer, or end user).

2.2 Operational malfunctions are not miscomputations

Towards the end of their analysis of miscomputation, Fresco & Primiero suggest, somewhat allusively, that what they call operational malfunctions might not be strictly computational either: "the cause of the miscomputation is often the physical substrate that is *contingent* to the computational process itself" (2013: final paragraph, emphasis added). If the physical substrate is *contingent* to the computations being performed, then it doesn't seem like operational malfunctions are an essentially computational phenomenon. Turing seemed to hold a similar view, writing that (abstract) computational machines (i.e. idealised Turing machines) are "By definition [...] incapable of errors of functioning" (1950: 449). Now, admittedly Turing was discussing abstract computations at this point, and we are concerned here with concrete (physical) computations. So should we say that a physical malfunction that causes a computing mechanism to produce the wrong result is (strictly speaking) a miscomputation?

The answer to this second question should also be "no", but to see why we need to think a little more about what (physical) computations are, and what it might mean for them to fail to operate correctly. According to the mechanistic account (introduced in section 1), a physical computation is just the transformation of some medium-independent vehicles ('digits') according to a rule. The rule itself is specified by the physical structure of the system (in the simplest case, by the structure of a single processing component such as an AND-gate), and, crucially, none of the components of the system themselves 'understand' the rule. It is by this simple conjuring trick that physical computation is able to produce seemingly semantic transformations from merely syntactic (or causal) processes – or as Haugeland memorably put it, "If you take care of the syntax, then the semantics will take care of itself" (1985: 106). An important consequence of this, though, is that a computational component does not "follow a rule" in the sense of an agent choosing to obey it, but rather just responds causally to the physical structure of the system of which it is a part. With this in mind, let us return to the question of miscomputation and operational malfunction.

There is one class of operational malfunction that we can dismiss immediately, those that render the system incapable of performing any computations at all. If the system overheats and catches fire, or if the power source stops working, then the system has not miscomputed, it is simply no longer computing. This much should hopefully be uncontroversial. The more difficult cases are those where the system still computes, i.e. we provide it with an input and it provides us with an output, but, due to a physical (operational) malfunction, it does not provide us with the 'correct' output (that which we would expect to receive, were we able to verify the abstract computation by some other means). A simple case like this might be one where the physical sensitivity of a voltage gate has changed, such that where it once computed AND it now computes OR (and even a simple case like this could of course have serious consequences for the more complex operations within which it is embedded).

This might seem *prima facie* like a clear case of miscomputation due to operational malfunction: the component was meant to compute AND, but (due to a change in its physical structure) it instead computed OR. One might say that its *function* was to compute AND, and its failure to do so qualifies as a malfunction. For the sake of argument I will accept this, but I do not think that the fact a computational component is malfunctioning necessitates that it is also *miscomputing*. Instead, the malfunction has transformed the computational identity of the component, by adjusting its physical structure such that it now performs a distinct computation. Recall that the function of a computing mechanism is to transform digits according to a rule, and that all it means for a computational component to follow a rule is for it to respond (according to its own physical structure) to the physical structure of the system of which it is a part. Our malfunctioning AND-gate has done precisely this – it receives a pair of voltage levels as inputs and, depending on whether the sum of those voltages is above a certain threshold, it produces another voltage level as output. So it is still following the rule embodied in its physical structure, and thus computing correctly, it's just that this structure has changed. That change itself might very well qualify as a malfunction, such that we can say that the AND-gate is a malfunctioning component, but I don't think that we should say that it is *miscomputing*. This is because the component, *qua* computation, has done nothing wrong; it is just doing what its (new) physical structure instructs it to (cf. Rapaport 2019: sec. 2).

Here one might be tempted to say that we should give priority to the rule that the component was originally designed to follow, such that it miscomputes if it does not follow *that* rule, rather than the new rule embodied by its (post-malfunction) physical structure. This would be to adopt something like a type-token distinction for computational components, and say that a token component miscomputes if it fails to follow the rule dictated by its type (which is fixed when it is first designed or built, rather than by its current physical state). This is a common move in the literature on malfunctions, but it will have some strange consequences if extended to miscomputation. Most notably, the computational identity of a component will come to depend on (spatially and temporally) distal facts about its manufacture and design, rather than proximal facts about its physical structure, i.e. what it can actually compute. While (relative to its design) the AND-gate discussed above might be malfunctioning, it also seems to be computing OR just fine, and could be used for this new purpose by someone entirely ignorant about its origins or (original) purpose. To say that it is miscomputing, then, would be to obfuscate its current capacity to compute OR, and to confuse the intended function of a component with the computational function that it actually now performs.

There are some obvious objections and replies to this argument that I will turn to in the next section, but for now I just want to summarise where we have got to. We saw that there are two main classes of (putative) miscomputations, those due to design errors of some kind, and those due to operational malfunctions. The former do not qualify as miscomputations because the fault lies outside of the system, whether that be with the system's designer, its manufacturer, its programmer, or its end user. The latter do not qualify as miscomputations because they either prevent the system from performing any computations at all (in the case of a total breakdown), or else they change the rule that the system is meant to follow (embodied in its physical structure) such that while it computes something different from what it was originally designed to, it does not *miscompute*. Therefore there is no such thing as miscomputation.

3 The topic formerly known as miscomputation

In this final section I will attempt to address a number of outstanding issues, possible objections, and further applications of the arguments developed in the previous two sections. Even if there is no such thing as miscomputation, we must still say something about cases where people (philosophers, cognitive scientists, and computer scientists) talk as though there was, in order to make sense of both our existing concept(s) of computation and actual scientific practice. It is worth emphasising, though, that the account presented here is not intended to be revisionary: rather than attempting to stipulate how researchers *should* talk about computation, I think this account can actually help us to explicate how researchers currently *do* talk about computation. So I am not advocating a general ban on the term 'miscomputation', but instead suggesting that we should exercise some caution in how it is understood and what it implies. (Or to put it another way, while there is strictly no such a thing as miscomputation, it might still make sense to use the term in an informal or colloquial sense, provided that it does not lead to any misunderstanding.)

3.1 Two objections

In the previous section I argued that an operational malfunction that changes the physical structure of a processing component, such that it now carries out a different computation, should not be classified as a miscomputation. Given that I deny that *anything* could count as a miscomputation, it seems like an obvious response would be to push back here, and argue that we should count at least *this* kind of operational malfunction as a miscomputation. After all, these are malfunctions that straightforwardly cause a component to compute something other than what it was designed for, so why not just call them miscomputations? I agree that this is an intuitive thought, and that it might be a relatively harmless way of using the term miscomputation, but I still think that it implies a misunderstanding about the nature of physical computation: to compute is just to follow a rule, but the rule that a computing mechanism follows is *not* that which was intended by its designer (to which it has no access), but rather that which is embodied in its physical structure. Ideally that structure will conform to its designer's intentions, but when it does not (whether due to poor design or later malfunction) the system itself cannot be blamed for following the 'wrong' rule. What has gone wrong here is not anything computational, but rather the initial specification of the rule itself (as embodied in the system's structure). So to call an operational malfunction a miscomputation is misleading, because the system is computing (according to the rule that it was given) just fine.

I have focused here on *mechanistic* accounts of computation, but there is another (fairly popular) class of *semantic* accounts, according to which the analysis of miscomputation might look quite different. While there has not actually been much work done on miscomputation by defenders of the semantic account, we can try and reconstruct what they might say. According to this account, physical computation is necessarily semantic; in addition to possessing the right kind of causal structure (as per the mechanistic account), a computational system must also *represent* something, and to compute is to manipulate *representational* vehicles according to a rule (see Sprevak 2010 for a basic overview). While there are many other points on which mechanistic and semantic accounts disagree (see e.g. Piccinini 2008, Dewhurst 2018, Shagrir 2018), this does not actually change things too much when it comes to our analysis of miscomputation. I think it is clear that semantic accounts should still deny that design errors are a kind of miscomputation, for all the reasons that I discussed in section 1. The same goes for operational malfunctions, except that there is a certain kind of malfunction that might qualify as miscomputation according to the semantic account, namely *misrepresentations* (cf. Neander 1995). If computation is representational, then (at least some) misrepresentations might plausibly be understood as miscomputations, especially those caused by operational malfunctions. For example, if an operational malfunction causes a computational state to misrepresent some feature of its environment, and as a result of this it produces an erroneous output, then according to the semantic account this might qualify as a miscomputation. However, there are also downsides to this approach, as Piccinini points out: a system might compute perfectly fine over states that we interpret as misrepresentations, or it might 'miscompute' over perfectly good representational states, so there does seem to be some benefit to keeping these two concepts logically independent (2015: 48). I will leave a full analysis of representational miscomputations for a proponent of the semantic account, but I mention this possibility here for the sake of fairness. If, like Piccinini, one thinks that allowing for the possibility of miscomputation ought to be included in the list of desiderata for our theory of physical computation, and furthermore one thinks that the notion of misrepresention provides some insight into the notion of miscomputation, then this might even be a reason to favour the semantic account over the mechanistic account.

3.2 Applications and further issues

I have focused so far (at least implicitly) on *artificial* computers, i.e. like the one that I am typing on now, but it would also be interesting to consider the question of miscomputation in *natural* computational systems, such as (potentially) the human brain. A full analysis of this question will have to wait for future work, but I would like to briefly comment on one particularly interesting aspect of it, which is the putative role played by miscomputations in computational psychiatry. Colombo (forthcoming) has recently argued that there are several different notions of miscomputation at play in computational psychiatry, and that a semantic account of computation is required to account for all of them. I will not respond directly to that argument here, but I instead want to suggest a different way to think about 'miscomputation' in computational psychiatry, building on Garson's (2019) work on mental disorders and malfunctions.

Garson argues that rather than being malfunctions as such, many cases of mental disorder might instead be better understood as "developmental mismatches", i.e. cases where a perfectly well-functioning system has just been placed into the wrong environment (2019: 176-8). He gives the example of how a child who has developed aggressive behavioural tendencies in order to cope with an abusive environment might later be removed from that environment and diagnosed with a conduct disorder (*ibid*: 179). In their initial environment this behaviour was adaptive, but once it has become ingrained and they are removed from that environment it becomes maladaptive. Design errors, in the context of computational psychiatry, look very much like developmental mismatches: the computational system (in this case, the brain) has been designed or programmed for one purpose (whether intentionally or by accident), but is then placed in an environment where this behaviour is maladaptive. These are neither cases of malfunction nor miscomputation: the system performs perfectly adequately, *qua* computation, but this performance does not match its novel context (some of the cases that Colombo considers, such as differences in the magnitude of prediction error signalling in healthy controls and schizophrenic patients, might also be explained in this way). Operational malfunctions, on the other hand, look more like traditional 'physical' diseases, which might cause the symptoms associated with mental disorders, but should not be understood as miscomputations (for the reasons I discussed in section 2.2). So the neurodegenerative process that is thought to cause Alzheimer's disease might be an operational malfunction that changes how the brain performs computations, but it is not itself a miscomputation, nor does it produce any. The distinction between operational malfunctions and design errors maps neatly on to Garson's loose distinction between physical diseases (often, but not always, understood as malfunctions) and mental disorders (often, but not always, understood in terms of developmental mismatches), and can help us to make sense of the role played by what might previously have been called 'miscomputations' in computational psychiatry.

Primiero, Solheim, & Spring (2019) describe malware (i.e., malicious software such as computer viruses) as a kind of miscomputation induced by a targeted attack. The taxonomy and analysis they provide is helpful, and provides some practical recommendations, but following my arguments in the previous section I think it is clear that malware does not cause miscomputations as such, but rather modifies the behaviour of a computational system in some way that is not desirable to its designer or user. The system itself either continues to compute perfectly fine (in the case of malware that hijacks or otherwise infects a system) or simply ceases to compute (in the case of malware that shuts it down entirely), but it does not miscompute. There is a further question of whether we should even think of malware as inducing (non-computational) *malfunctions* in the target computer, i.e. by rewriting the instructions (rules) embodied by its physical structure (see section 2.2). Here an analogy with biological viruses might be helpful: relative to the host system, a (biological or computational) virus certainly induces a malfunction, but relative to the virus itself (or its designer), the system functions perfectly well, by serving to reproduce the virus. So in many cases there may not be a clear answer to the question of whether malware induces a malfunction, let alone a miscomputation.

Finally, I want to briefly discuss a useful distinction introduced by Tucker's (2019) assessment of miscomputation, between a mechanism's *proper* and *actual* functions. He is a realist about miscomputation, but he defines it in terms of *norms* that are individuated widely (i.e. with reference to selection history or designer intentions), whereas he defines computational *behaviour* in narrow terms (i.e., just in terms of the physical structure of the system). Widely individuated norms fix the proper function of a computing mechanism (what it's *supposed* to do) while narrowly individuated behaviour fixes its actual function (what it *actually* does), and, for Tucker, a miscomputation occurs when these two functions are misaligned. This could be because of either design error (the system was programmed badly) or operational malfunction (it doesn't work properly). I am willing to allow that mismatches of this kind explain what we usually call miscomputations, but I would argue that we should restrict strict talk of computation to what Tucker calls 'actual' functions, with the consequence that there are also, strictly speaking, no miscomputations.

If computational identity is determined by proper functions in Tucker's wide sense, then some kinds of design error might also end up counting as miscomputations, because the designer's original intentions could be taken to determine what the system 'ought' to compute. For example, type-2 (manufacturing) errors could count as miscomputations if we fix computational identity relative to the designer's original blueprint. The problem with this approach, however, will be deciding *which* and *whose* intentions ought to count for fixing computational identity, especially in cases where these intentions come apart (with downstream consequences for what will count as a *mis*computation). If the designer in the previous example had intended for the system to be built one way, but provided a bad blueprint such that the actually constructed does not function 'correctly', is this a type-2 miscomputation due to a bad manufacturing process relative to their original intentions, or a type-1 miscomputation due to their poor specification of the blueprint? Or consider another example, focusing on the other end of the taxonomy: if an end-user struggles to use a complicated program and receives error messages, is this a type-5 miscomputation due to their failure to use the system properly, or a type-4 miscomputation to the program being written in a way that makes it hard to use properly? If this were just a one-off case it might seem obvious that we should go with the first option, but if enough users experienced similar problems then we might be more tempted to lay the blame on the programmers (and indeed, such complaints might trigger a product recall or update to fix the 'error').It would be better, I suggest, to restrict computational identity to the actual functions specified by the physical structure of the system, and treat each of these putative cases of miscomputation as distinct kinds of problems with distinct kinds of solutions, the variety of which continuing to refer to all of them as 'miscomputations' would obfuscate.

If there is no such thing as miscomputation, then what ought we to say about the inclusion of miscomputation in Piccinini's list of desiderata for an account of physical computation? I think he is correct to say that we need to make sense of the kinds of thing people (including computer scientists, computational neuroscientists, etc.) say when they believe that a computational process has 'gone wrong' in some sense, but I don't think that we need any specific notion of miscomputation in order to do this – and in fact, relying on such a notion might hold us back in some cases. The taxonomy provided by Fresco & Primiero (2013) makes it clear that there many different ways in which a computational process can 'go wrong' that don't really have much in common with one another at all. Our diagnosis of, and solution to, a programming error is going to be very different to that which is appropriate for a manufacturing defect or operational malfunction. Each case calls for a different kind of analysis and response, and so referring to them all under the broad category 'miscomputation' is misleading at best. Of course, one could choose to use the term to refer to just one of these cases (probably operational malfunctions), but for the reasons I have argued for here, I think even this limited usage could have confusing implications, and so it is better that we avoid it in favour of more precise language. Where miscomputation talk is used in a formal setting (e.g. computer science or computational psychiatry) it ought to be replaced with more precise terminology relating to each putative type of miscomputation, or at least it should be made very clear what specifically is meant by 'miscomputation' in each case. In fact, I think this is a less controversial point in these disciplines than the likes of Piccinini or Colombo assume it to be, and so I take my conclusion here to be less in contradiction with current scientific practice, and more a gentle encouragement towards conceptual hygiene. On the philosophical side, this would mean that we could drop the requirement to accommodate miscomputation from the list of desiderata for an account of physical computation, potentially making it somewhat easier to come up with such an account.

Conclusion

I first reviewed some existing accounts of physical computation and miscomputation, focusing on the version of the mechanistic account developed by Gualtiero Piccinini. I then argued that, according to this account, there can be no such thing as miscomputation, as all putative cases of miscomputation are either the result of design errors (for which the computing mechanism itself cannot be blamed) or operational malfunctions (which are not strictly computational). Finally, I considered some possible objections and further implications, including the idea that operational malfunctions should in fact qualify as miscomputations, an alternative semantic account of miscomputation, the role of miscomputation in computational psychiatry, the case of malware, and a recently proposed distinction between proper and actual functions. In future work I would like to extend these considerations to include some more general applications of the argument that there is no such thing as miscomputation, such as to debates about the neuroscience of free will, agency in artificial (and natural) systems, and what it means for a computational system to follow a rule.

References

Chalmers, D. 1994. "On implementing a computation." *Minds and Machines*, 4(4): 391-402.

Coelho Mollo, D. 2018. "Functional individuation, mechanistic implementation." *Synthese*, 195(8): 3477-3497.

Coelho Mollo, D. 2019. "Are There Teleological Functions to Compute?" *Philosophy of Science*, 86(3): 431-452.

Colombo, M. Forthcoming. "(Mis)computation in Computational Psychiatry." In Calzavarini Viola (eds), *Neural Mechanisms.* Springer.

Dewhurst, J. 2014. "Mechanistic Miscomputation." *Philosophy & Technology*, 27(3): 495-498.

Dewhurst, J. 2018. "Individuation Without Representation." *The British Journal for the Philosophy of Science*, 69(1):103–116.

Floridi, L., Fresco, N., & Primiero, G. 2015. "On Malfunctioning Software." *Synthese*, 192:1199–1220.

Fresco, N. 2014. *Physical Computation and Cognitive Science.* Springer Netherlands.

Fresco, N. & Miłkowski, M. 2019. "Mechanistic Computational Individuation without Biting the Bullet." *The British Journal for the Philosophy of Science*, online first.

Fresco, N. & Primiero, G. 2013. "Miscomputation." *Philosophy & Technology*, 26(3):253-272.

Garson, J. 2019. *What Biological Functions Are And Why They Matter.* Cambridge, CUP.

Godfrey Smith, P. 2009. "Triviality arguments against functionalism." *Philosophical Studies*, 145:273–295.

Haugeland, J. 1985. *Artificial Intelligence: The Very Idea.* Cambridge, MA: MIT Press.

Milkowski, M. 2013. *Explaining the Computational Mind.* Cambridge, MA: MIT Press.

Millhouse, T. 2019. "A Simplicity Criterion for Physical Computation." *The British Journal for the Philosophy of Science*, 70/1: 153-78.

Millikan, R.G. 2013. "Reply to Neander." In Ryder, Kingsbury, Williford (eds.), *Millikan and Her Critics.* Malden, MA: Wiley-Blackwell.

Neander, K. 1995. "Misrepresenting & malfunctioning." *Philosophical Studies*, 79:109–141.

Petricek, T. 2017. "Miscomputation in software: Learning to live with errors." *The Art, Science, and Engineering of Programming*, 1.2: 14.

Piccinini, G. 2007. "Computing Mechanisms." *Philosophy of Science*, 74(4): 501-526.

Piccinini, G. 2008. "Computation without representation." *Philosophical Studies*, 137(2):205-241.

Piccinini, G. 2015. *Physical Computation.* Oxford: OUP.

Primiero, G., Solheim, F.J., & Spring, J.M. 2019. "On malfunction, mechanisms and malware classification." *Philosophy & Technology*, 32(2), 339-362.

Putnam, H. 1960. "Minds and Machines." In Hood (ed.), *Dimensions of Mind: A Symposium*, New York: Collier.

Putnam, H. 1988. *Representation and Reality.* Cambridge, MA: MIT Press.

Rapaport, W.J. 2019. "Syntax, Semantics, and Computer Programs." *Philosophy & Technology*, online first.

Schweizer, P. 2019. "Triviality Arguments Reconsidered." *Minds and Machines*, 29(2):287-308.

Shagrir, O. 2018. "In defense of the semantic view of computation." *Synthese*, online first.

Sprevak, M. 2010. "Computation, individuation, and the received view on representation." *Studies in History and Philosophy of Science Part A*, 41(3):260-270.

Sprevak, M. 2018. "Triviality arguments about computational implementation." In Sprevak & Colombo (eds.), *The Routledge Handbook of Philosophy of Computation.* Routledge.

Tucker, C. 2018. "How to Explain Miscomputation." *Philosophers' Imprint*, 18(24).

Turing, A. 1950. "Computing Machinery and Intelligence." *Mind*, 49:433-460.