# Software Intensive Science

Jack Horner and John Symons

## Introduction

The increasingly central role of computing technologies has changed science in significant ways. While the practical import of this transformation is undeniable, its implications for the philosophical understanding of scientific inquiry are less clear. Some philosophers have argued (and we agree) that there is more to recent changes than simply the appearance of a more powerful set of tools for scientists. In our view, there are important new philosophical questions that arise with the use of computing technology in science. There are at least two aspects of recent changes in science that merit philosophical reflection. From our perspective, the clearest and most fundamental of these is the role of software in science. Strikingly, the impact of software on scientific practice has attracted almost no attention from philosophers of science. By contrast, many philosophers have addressed the second prominent aspect of this change; the appearance of powerful and relatively inexpensive computing technology for scientific modelers. Philosophers have noted that greater processing power has led to more sophisticated scientific models and that new possibilities for modeling have opened new domains for scientific inquiry. Since the mid-1990s there has been extensive philosophical research into the role of computational models in scientific explanation.[1] While we agree that the quantitative increase in power and the accompanying widening of the scope of computational models and simulations is exciting and important, in this paper we will focus on the less widely discussed role of software in modern science.

The purpose of this paper is to explain the difference between pre- and post- software intensive science in precise terms and to explain why this difference should matter to philosophers of science. We will argue that the clearest difference between contemporary software intensive scientific practice and more traditional non-software intensive varieties derives from the characteristically *high conditionality* of most software used in modern science. Very roughly, by *conditionality* we mean that the course of a computation (i.e., the order in which software instructions are executed) is determined by conditional schemata, e.g. "If *x* obtains then do *y*". The *if-then* (or *if-then-else*) schema is at the heart of most modern programming languages and as we will show it is this very basic characteristic that marks the most

[1] See Symons 2008 for a discussion of how computational models have figured in discussions of the metaphysics and epistemology of science.

important difference between modern software intensive science and its predecessors.[2] Our purpose in this paper is to help to clarify the significance and implications of high conditionality for software intensive science in formal terms. Appreciating the difference between software intensive and non-software intensive science will facilitate a properly informed discussion of the challenges that accompany the ongoing transformation of scientific inquiry in our time.

## Recent work in the philosophy of software intensive science

Even a superficial survey of the changing landscape of scientific inquiry reveals potential areas of philosophical interest. Consider, for example two prominent papers from *Science* and *Nature* respectively: In 2009, Michael Schmidt and Hod Lipson described how their program *Eureqa* inferred Newton's second law and the law of conservation of momentum from descriptions of the behaviour of a double-pendulum system. (Schmidt and Lipson 2009) More recently, Eugene Loukine and colleagues demonstrated a model that was able to predict unforeseen side-effects for pharmaceuticals that were already approved for consumption. (Loukine et al, 2012) These two papers represent very different examples of software intensive science: One, a system is capable of generating theoretical insights and law-like relationships from a data set while the other makes dramatic progress on a specific practical question of great importance. Examples like these indicate that across a broad swathe of scientific endeavor, from highly theoretical to applied science, inquiry itself is no longer purely a matter of individual or collective *human* effort. Across the sciences, software-intensive systems are increasingly driving the direction of research and in some cases are already beginning to displace human researchers. Unlike previous improvements in scientific technology, computers not only extend our capacities, but are taking on at least some of the cognitive aspects of theoretical work in the sciences. From a broader cultural perspective, science has already taken the first steps toward a post-human future. In recent years philosophers of science have begun to follow science fiction writers in working to unpack the many important questions associated with this transformation.

There is an ongoing debate among philosophers of science concerning the implications of this new way of doing science. Disagreement has centered on topics such as whether computational modeling challenges our philosophical understanding of scientific ontology, whether it changes the relationship between theory and experiment in some significant way, whether we will be forced to rethink the semantics of theories, etc. Philosophical reflection on these questions is in its early days and has been

[2] All commonly used programming languages contain some implementation of this schemata (Boolos, Burgess, and Jeffrey 2002).

met with skepticism from traditional philosophers of science. Recently, for example, Roman Frigg and Julian Reiss have rejected the claim that these developments raise interesting new philosophical problems or questions. (2009) Frigg and Reiss correctly point out that much of the excitement among philosophers is diminished when we recognize that these apparently new challenges are not unique to computational models and simulations. They argue that many of these issues have variants that arise elsewhere in the history of philosophy of science. We agree with their judgment that many of the issues raised in the philosophy of science with respect to the role of computational technologies have been addressed to some extent in earlier work. However, in our view, philosophers of science who have been drawn to the topic are correct to see something philosophically significant in recent developments. While we do not agree with Frigg and Reiss that the existing literature in philosophy of computational modeling and simulation is "the same old stew", we accept the general point that the philosophical topics that have been addressed *to date* are not unique to computational models and simulations.

We will provide a brief overview of some of the existing philosophical literature before explaining why we focus on software. In our view, the distinctive character of software intensive systems – their high conditionality – is an aspect of the transformation of modern science that has not been addressed properly in the literature in the philosophy of science to date.

Philosophers of science began to take note of the growing significance of software – particularly insofar as it figures in the realization of computational models, simulations, agent based models, and Monte Carlo methods -- in the early 1990s. Early philosophical engagement with computational models and simulations tended to be critical. So, for example, Oreskes, Belitz and Shrader-Frechette (1994) urged policy makers to avoid relying on evidence generated by software-based computational modeling techniques and challenged their scientific legitimacy. By contrast, Humphreys (1994), Winsberg (1999), and others recognized that for a range of urgent and fascinating scientific questions, computational models and simulations are often our only viable research tools. By now, it has become clear that in contexts where cognitive, economic, ethical, political, or practical barriers would otherwise loom large, software will inevitably figure in scientific inquiry for the foreseeable future. (Symons and Boschetti 2013, Boschetti et. al 2012).

Because of the centrality of the concept of explanation in philosophy of science, it is unsurprising that most discussions of software in the literature have focused on determining whether and in what ways explanations derived from computational models differ from those provided by other kinds of scientific models. (See for example Humphreys 1994, Guala 2002; Parker 2008; Winsberg 2010) In addition to debates over the nature of explanation, there has been some attention to the role of error and uncertainty in computational modeling. (Winsberg 2010) Computational models, like all scientific models, are subject to error and uncertainty. Like much of ordinary science, computational models depend on

idealizations and like most work in the so-called special sciences, the generalizations that result from modeling are subject to ceteris paribus clauses (Symons and Boschetti 2013; Batterman 2009) In these respects, computational modeling is continuous with the traditional scientific enterprise and naturally enough, philosophers have approached computational methods with the inherited set of distinctions and questions from traditional philosophy of science. However, as we will argue below, high conditionality changes the landscape for philosophers of science in interesting ways.

It is a relatively uncontroversial fact that the centrality of computational models in scientific explanation means that modern scientific prediction and explanation are more reliant on software with high conditionality. It is also clear that the existing philosophical literature in philosophy of science has not addressed this aspect of computational models and simulations. One challenge for us in this paper is to explain why software with high conditionality makes software intensive science (hereafter SIS) qualitatively different from non-software intensive science (hereafter NSIS). We will concentrate on the epistemic implications of high conditionality in software as it figures in modern science and why this difference should matter to philosophers of science.

While this paper addresses the epistemic consequences of high-conditionality, we recognize that this is not the only aspect of SIS that deserves the attention of philosophers. However, before making progress on questions concerning the explanatory role of SIS or its implications for scientific ontology, we believe that it is necessary to get clear on the basic differences between SIS and NSIS. Once we understand the role of high conditionality in SIS, important differences between SIS and an NSIS that have been missed in the literature to date are brought into clear focus.

In order to show how high conditionality in computational modeling changes the epistemic status of inquiry we will introduce a measure of conditionality that distinguishes SIS from NSIS, we then provide a brief description of several kinds of errors and problems that unavoidably arise in SIS as a result of this conditionality. Given that we sometimes have no alternative but to rely on computational models or simulations, the challenge is to understand and, to the extent possible, mitigate the distinctive kinds of problems associated with this kind of conditionality. The philosophical import of this project derives from our claim that high conditionality is the source of a significant difference in the epistemic characteristics of SIS and NSIS. A rough initial sketch of this difference goes as follows: While code for scientific models in software intensive systems contains high conditionality, this was not a feature of science that held much significance prior to the late-Twentieth Century. Oversimplifying somewhat at this stage, we can contrast the high conditionality of modern scientific models with the unconditional statement of a scientific law in traditional theories. Traditionally, the statement of a scientific law, like Newton's for gravitation or Maxwell's for electromagnetism has been understood to take the form of unconditional equation or a generalization. While the precise characterization of scientific law is highly

controversial, philosophers of science have generally pointed to a few basic features that they share in common. Most importantly, scientific laws should be inductively confirmed and should support the relevant kinds of counterfactual statements. The generality of scientific laws is another important virtue, while their role in explanation and counterfactual reasoning is a significant source of intellectual satisfaction. Maxwell's equations, for example, provide a statement of the relationship between electrical and magnetic fields that allows us to understand and predict (approximately) how changes in one affect changes in the other. Maxwell's equations tell us what the relationship between the measured values of the fields and thereby unifies previously disconnected aspects of experience. They do this while simultaneously supporting satisfying explanations and permitting us to reason about counterfactual situations in illuminating ways.

Of course, many scientific laws can be (and frequently are) paraphrased as conditionals, so for example, "All *A*s are *B*s" can be read as "(For all *x*) if *x* is an *A* then it is a *B*." However, the scope of universal quantifier and the role of the variable in a scientific law are quite different than in a conditional schema in a piece of software. As we shall show, high conditionality in software programs can be understood in terms of path complexity. We will explain the concept of path complexity below and will show that NSIS has minimal path complexity while the kind of high conditionality that are associated with SIS leads to extremely high levels of path complexity. In addition to the difference in path complexity between SIS and NSIS, conditional statements in software programs usually lack the relevant kind of generality and do not support the kinds of counterfactual explanations that the paradigm cases of unconditional equations or generalizations do.

We do not intend to imply that scientific laws play a role in NSIS that is analogous with the role played by lines of code in SIS. Nevertheless, this rough sketch of the difference between high conditionality that we will show is built into SIS and the relatively low conditionality of NSIS should be enough to introduce our project here. We will provide a formally precise characterization of the nature and implications of high conditionality in SIS below. We hope to demonstrate that the difference is significant and that it underlies the intuitive sense that something fundamentally new is taking place in contemporary scientific practice.

## Software intensive systems, software conditionality, and the limits of testing

Computer scientists call a system *software intensive* if "its software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole" (IEEE 2000). Clearly, by this definition, much of contemporary science involves software intensive systems. Insofar as computational

methods have changed the character of scientific inquiry since the 1970s, we contend that this change results from the pervasive and indispensible role that software now plays in much of science in the industrialized world. We can begin to understand this change by comparing NSIS and SIS. Once we understand the distinctive features of SIS, we will be in a position to identify the effects of high conditionality on contemporary scientific practice.

How should we understand the notion of an "essential influence" as used in the IEEE definition of "software-intensive"? In the following, we propose a precise characterization of a "software-intensive system" in terms of a measure of path complexity. In the broadest use of the term, a *software system* is the collection of processes and artifacts, abstract or concrete, that are essentially or by fiat associated with a sequence S of instructions written in some computer language L (ISO/IEC 2008). In this sense, a software system includes computer programs written in L, together with all of the design, test, implementation, and operational processes and artifacts associated with S. Here, unless otherwise noted, however, we will restrict the term *software system* to mean S, together with any machine model A (Waite and Goos 1984, especially Chap. 3) that satisfies the semantics of the language in which S is written. We understand the semantics of the language to be specified in model theoretic terms along the lines presented in, for example, Chang and Keisler (1990).

We will define a *path* (Diestel 1997) in a software system to be a non-repeating sequence of instructions/program-statements in a computer program that could, for some set of conditions, be executed on the machine model A. The *path complexity* of a software system S, as we use that term, is the number of possible paths in S. The number of paths in a program increases with the presence of conditional statements. By way of illustration, suppose a software system included the following instructions:

Line 5: If (“mass is less than 10 units” is true) then go to line 100, else go to line 6.
Line 6: Print “mass is greater than or equal to 10 units”
.
[list of other instructions]
.
Line 100: Set mass to 0 units

In this example there are two paths. In one of these paths, the system skips the intermediate lines between line 5 and line 100. Those intermediate instructions are not executed. Otherwise the system proceeds to execute the instructions on line 6. Note that the conditionality at line 5 generates two paths. Line 5 is an example of a *binary conditional statement* -- a conditional statement that defines two possible

paths. On average, in actual software systems, there is approximately one binary conditional statement (such as line 5) per 10 lines of software. Thus, each binary conditional statement like line 5 doubles the number of paths in a software system. Two binary conditional statements in a software system will result in 4 paths in the system, three binary conditional paths will result in 8 paths in the system, four will in 16, and so on. It is worth noting that this definition of complexity differs from *McCabe complexity*, which is the number of independent paths in a software system (McCabe 1976).[3] Path complexity, as we have defined it here, captures the space of possible ways that the software system could run.

So, what role does path complexity plays in the distinction between NSIS and SIS? Let's first consider a 1000-line (instruction) software program that has a binary conditional statement (such as line 5 in the example above) every 10 lines on average. The number of paths through such a program, and hence its path complexity, is $2^{1000/10} = \sim 10^{30}$. In general, the path complexity of a program of M lines that has a binary decision branch on average every N lines is $2^{M/N}$, where M > N. A 1000-line program is extremely short by contemporary scientific software standards. Almost all modern scientific software systems are larger than 50,000 lines, and 100,000-line systems are common enough (Center for Systems and Software Engineering 2013; Horner 2003). Clearly, SIS involves programs with extremely high path complexity.

Our next task is to show why this high path complexity is philosophically significant to the practice of SIS. We will begin by examining some of the consequences of high path complexity in SIS for the epistemology of science. One way to evaluate the epistemic status of SIS is to consider the degree of confidence we should assign to the results of a system with high conditionality. A natural way of thinking about the appropriate degree of confidence we would give to these systems is in terms of their reliability. We can be more confident in the output of a system if we can judge it to be reliable. Given the important role of SIS, even in terms of the practical role of its outputs, we need some assurance of their reliability independently of philosophical questions.

How can we determine the reliability of software systems given their typical high path complexity? In general, we can be confident of what a software system does ***only*** for those conditions under which it has been tested, or more precisely, only for those paths that have been tested. Of course, not all paths need to be tested in all practical applications. For example, we would not need to test the behavior of an airbag control system in a car for speeds over 300 miles per hour. Nevertheless, it is important to recognize that we know nothing about the behavior of the system in these untested cases. Many paths will be deemed irrelevant to the operation of the system for good principled and practical reasons, for example, the limitations of automotive performance in the case above.

[3] Technically, path complexity is the least upper bound of the number of paths in a software system. We are grateful to George Crawford for helpful discussion of this point.

Given high path complexity, we face profound restrictions on what we can claim to know about a software system. Suppose, for example, we could define a test case for, execute, record, and analyze 1 path per second through a 1000-line system that has on average, one binary branch per ten lines. Note that in general this rate is humanly unattainable because we must first design test cases and analyze test results. Returning, for example to our case of the 1000- line system with a path complexity of $\sim 10^{30}$, let's compute the time it would take to exhaustively exercise all paths, i.e. test, all paths in the system. The estimated lifetime of the universe is ~10 billion years, or about $10^{17}$ seconds. So it would take $\sim 10^{30}/10^{17} = \sim 10^{13}$ lifetimes of the universe to sequentially exercise each path through a typical 1000-line software system that contains one binary branch, on average, per ten lines. Exhaustive path testing for typical software systems is therefore intractable. Even within the space of relevant behaviors for applications there will be untested paths in those systems.

In practical terms, as we saw above, a typical 300-line program with one binary branch per 10 lines, on average, would have a path complexity of $2^{30}$, or roughly $10^8$. Statistically speaking, therefore, a 300-line program is the largest "typical" program whose paths can be exhaustively tested in 5-10 years. This is a reasonable upper-bound on the time that would be spent testing a software system given typical funding cycles in the sciences.

The foregoing considerations allow us to define and distinguish NSIS and a SIS in practical terms. We define a SIS to be a system that uses a software system that has path complexity of at least $10^8$ (one hundred million). An NSIS, by contrast, is a system that does not use a software system that has a path complexity of at least $10^8$. At first blush, the $10^8$ path threshold in our NSIS/SIS distinction might seem arbitrary. However, the bounding considerations we discuss above show that it has the right order of magnitude. $10^8$ is a clear general practical upper bound on the path complexity of code all of whose paths can be exercised during testing.

Once a software system becomes a part of practice in a scientific domain, our knowledge of what the system, and more generally, anything that depends essentially on what that system does, is subject to the testing limitations explained above. As we shall see below, insofar as software is untested, its epistemic status in inquiry is fundamentally different from the unconditional equations of NSIS.

Scientific theories that do not rely heavily on software contain little if any conditionality and hence, unlike the execution of a typical software system, an NSIS will have few to no branches. For example, in their canonical forms the classical theory of electromagnetism (Eyges 1972) and Newton's theory of motion (Newton 1726) contain no conditionality. Such theories are traditionally regarded as paradigm cases of what scientific understanding should look like. There is only one path through such theories; their path complexity is 1. Arguably, a large part of the rational satisfaction that we derive from NSIS theories may be due to their lack of conditionality and the resulting lack of path complexity.

Against these considerations, Paul Humphreys has objected that some applications of Newton's laws have a complexity that is at least comparable to that of software systems, and if so, there is a sense in which at least some applications of Newton's theory are at least as complex as any software system.[4] This is true. However the character of the complexity in these cases is quite different from an epistemic perspective. For example, let's imagine a Newtonian "random walk" of N steps, in which each step there is a choice between just two directions. The set of such paths will contain $2^N$ elements, it seems in a way that is strictly analogous to the way that a software system could contain $2^N$ paths. Thus, an objection along the lines proposed by Humphreys might continue, the complexity created by the inclusion of software in science is not a distinctive order of complexity. And if so, the measure of complexity as we have presented it does not capture a significant difference between NSIS and SIS. In general, objections of this kind assert that we are in precisely the same conditions with respect to testing and error in NSIS cases like Newton's theory of motion as we are in SIS cases with high path complexity. However, as we will show below, this class of objections misses the epistemic difference in the roles complexity plays in NSIS and SIS.

In order to explain the epistemic difference between high path complexity in SIS and NSIS it is helpful to consider how an epistemic agent could cope with error in inquiry. In the case of NSIS, there is a relatively straightforward approach that an agent could take. In general, the distribution of errors in an NSIS can be characterized by conventional statistical inference theory (Hogg, McKean, and Craig 2005, Chaps. 5-12), That approach requires us to randomly draw (Hogg, McKean, and Craig 2005, Df. 5.1.1) a sample from the population of interest, then apply statistical tests to the sample to assess the probability that specific hypotheses about the population hold (e.g., whether the population has a normal distribution). Consider the "random walk" example above: We draw a random sample from the trajectories produced by the system, determine by inspection of the statements in the system that there is no conditionality that could produce trajectories that could be statistically unlike those we have sampled, then use conventional statistical inference theory to test whether it is likely the population of trajectories has the distribution (in this case, a "random walk" distribution) we expect. Typically, the sample size required to test a hypothesis of interest in such a case is small -- on the order of 100.

The situation in SIS is considerably less friendly to sampling because of the challenge posed by high conditionality. This is because the high conditionality of SIS requires us to draw random samples that are so large we cannot sample of the population of interest in a way that is sufficient to evaluate H at a confidence level (say, 90%; Hogg, McKean, and Craig 2005, Section 5.4) in which we have an interest. Here's why. Nothing prohibits the errors in an SIS from being independent (Chung 2001, Section 3.3). If so, the Central Limit Theorem (CLT; Chung 2001, Chap. 7) tells us that the distribution of those errors

[4] Personal correspondence.

could converge to a normal distribution (Hogg, McKean, and Craig 2005, Df. 3.4.1) as the size of the system increases. The dimensionality, D, of the space of paths in a software system is path complexity of that system. Theory tells us that the sample size K required to assess the hypothesis

(H) The distribution of errors is a normal distribution.

at a given confidence level, scales *exponentially* as D, and thus, exponentially as the path complexity, that is[5]

$K = a^D$ Eq. 1

where a is a parameter that is determined by the confidence level at which we wish to assess H. For example, if D = 1, and we wish to assess (H) at the 90% confidence level, , then a = 100 and K = 100. If all else remains the same and D = 2, then K = 10,000.

Now to show that we have drawn a random sample from the population P of software paths in a given software system, we must *serially* test (because we must exhaustively compare each element of the sample with all other elements of the sample) whether the combination of results is a random sample. A lower bound on the time in which any information (a "signal", in the the sense of Reichenbach 1958) can be obtained from a material comparing system is the time it takes light to traverse the classical electron radius ($\sim 3 \times 10^{-15}$ m). Given Eq. 1, therefore, even if we could test, in the time it takes light to traverse the radius of an electron ($\sim 10^{-23}$ second), whether a given element of a candidate sample is part of a random sample, the time it would take to serially verify a large enough random sample, to be such, in the smallest software system in an SIS, would be $\sim 100^{100,000,000}$ seconds. This implies that in general it is not practically possible to draw a random sample, tested to be such, from a software system in an SIS, in order to assess whether (H) is likely to be correct at the 90% confidence level.

Similar arguments can be constructed from the above rubric for statistical hypotheses other than (H), e.g., whether the mean of the distribution of errors has some specific value *v*.

There is thus, in general, no practical way to bound the behavior of a SIS simply because there is no known practical way to characterize the distribution of errors in software in that SIS. Without such a characterization, we are forced to rely on a much more conservative, and expensive, criterion of "knowing" – testing – than in the case of an NSIS.

[5] For a derivation, see Hogg, McKean, and Craig 2005 (Sections 2.6 and 9.4).

Because there are never more than a few arbitrary conditional paths through theories like Maxwell's or Newton's, in contrast – the paradigms of NSIS – we can exercise (i.e., in the same sense as above, at least minimally "test") all paths in an NSIS.

It might be objected to the foregoing that it is unreasonable to demand that all paths in the software in a SIS be tested before that program can be used in research. This point is well taken, but it does not in general change the limitations of our knowledge of what a given software system does.. In friendly cases, we may be able to certify the reliability of scientific software for specific applications under very specific conditions -- those that are identical to the conditions of our test cases. Given that this is the correct (and only feasible) approach to take to testing scientific software, however, we are still left with the problem of characterizing, and avoiding the execution of, untested software paths.

It has sometimes been suggested that there is a general software development methodology, which, if rigorously followed, would help to greatly reduce the error-frequency in software.(ISO/IEC 2008; Black, Veenendaal, and Graham 2012). Although there is some truth to this suggestion, it says much less than it might at first seem. There is no known effective, or even generally accepted theory or method for producing correct software (Graham, Clancy, and DeVaney 1973). This is unlike NSIS, in which we *do* have generally accepted -- indeed, mandated -- method and theory. (Kuhn 1970, Sections II-V)

Our concerns have practical as well as theoretical import. In practice, the severity of problems generated by untested parts of software depends on the consequences of that software's behavior. For example, the software in a pacemaker might induce fibrillation if the pacemaker happens to enter an untested path. Consequently, in the case of pacemaker software, we have no choice but to test all possible paths. Because of this, most pacemaker software has very low path complexity. However, experimental pacemakers that can be controlled over the Internet, do have SIS-level path complexity. Systems of this kind introduce new risks.

Similarly, given United States' commitment to observe the Comprehensive Nuclear Test-Ban Treaty (United Nations 2006), the US must rely heavily on simulation to verify the safety and efficacy of nuclear weapons (National Coordination Office 2013). In this case, the software is a proxy for the weapons proper, requiring very high confidence that the software reflects all behavior of such weapons in which one has an interest (Gustafson 1998). In contrast, if the software is simply a "throw-away" experiment for summing, say, the first N integers, we might not care what it does if, say, we accidentally input real-valued data into that software.

One might be tempted to further refine the NSIS/SIS path-complexity threshold, but it is more than difficult to imagine what of consequence could come from such an effort. In any case, nothing in this paper would be sensitive to such a refinement.

No matter how we decide to restrict our tests the restrictions on the reliable domain of application must be well-understood and appreciated by the users of the software.

## Error and uncertainty unique to SIS

The fundamental role played by complexity due to high conditional plays in a SIS changes the epistemic landscape (that is, our ability to know the distribution of errors in the software in SIS) of science in ways indicated above. Now there are certain kinds of errors that arise -- some intrinsically, others, with high frequency -- in a SIS that are unique to software, and that will occur under unknown conditions, because in general not all paths in a typical software program can be exercised. In addition to the kinds of modeling-approximation errors mentioned in the Introduction of this paper there are at least two general types of software-specific error:

precision (reproducibility) problems that arise essentially as a function of program size, and

intrinsic numeric sources of error.

We briefly describe these two classes of errors.

**Precision and the T experiments**

In all modern computing languages, we can (approximately) represent any real value, R, in a canonical normalized form (sometimes called "scientific notation" form),

$$R = M.SS\ldots \times 10^{K} \qquad (C)$$

where

M is an integer between 1 and 9, inclusive
SSS… is a finite sequence of non-negative integers
K is a non-zero integer

Let Z be a digit in M.SSS …. The number of *significant digits* in an instance S of form (C) is the number of digits in M.SSS…, counting from the left, such that the values obtained under repeated determination (i.e., by repeated measurement or computation) of S lie in the range determined by $Z \pm 1$. The number of significant digits is a reproducibility measure (Halmos 1950) closely related the variance of a probability distribution and to a statistical confidence interval. (Hogg, McKean, and Craig 2005) In the jargon of computer science, the number of significant digits is called the *precision* of R.

In the absence of comprehensive precision management, the precision generated by scientific software degrades rapidly as a function of program size. In a very significant empirical study of this phenomenon, Hatton (1997) demonstrated that ~15 randomly chosen scientific software systems each containing at least 50,000 lines of code, distributed across diverse scientific application domains, yielded results with at most one significant digit – regardless of application domain and authorship. Hatton conjectured that this striking error-rate invariant arose from potentially diverse sources ranging from failure to model error distributions in inputs, to failure to manage error-propagation within computation.

Whether a single-significant-digit result is acceptable typically depends on the application. Let's suppose, for example, that the temperature of the coolant in a nuclear reactor must be kept between 500 and 525 degrees Fahrenheit if the reactor is to operate safely. If the control software that manages this temperature has only one significant digit, then the actual temperature control is at best $500 \pm 100$ degrees, which would be a disaster.

More surprising still were results Hatton obtained by analyzing the number of significant digits in the outputs of software systems composed entirely of *randomly generated* sequences of instructions. As the size of such "random"-programs goes to "infinity", he discovered, the number of significant figures in the outputs of these randomly generated sequences asymptotically approaches 1 (Hatton 2013). Programs as small as 15,000 lines of code clearly exhibit this tendency.

What should we infer from such results? Surely not that it is impossible to build a program that can produce more than one significant figure. Rather, it is clear warning that in absence of aggressive error management, as a function of program size, programs will devolve to little more than noise generators.

A naïve attempt to fix this kind of problem – adding error-checking code to a program – is a devil's trade. The error-checking code itself must be tested, so in general adding error-checking code to existing software adds more de facto untestable paths to the code.

Alternately, it has been suggested that it is possible to create software systems that automatically test other software systems (Fewster and Graham 1999). In this scheme, a "test harness" running on a computer replaces human labor in software testing. Using automated testing, it would seem that software systems of arbitrary size could be tested at the speed a computer can run, thereby obviating, if not

collapsing in principle, the NSIS/SIS distinction. Unfortunately, there are at least two problems with this proposal.

First, a test harness is a software system which itself must be tested. The testing of the test harness is subject to the same limitations of software testing mentioned in the NSIS/SIS distinction. There may be, of course, special cases in which the test harness adds no path complexity. These are special cases, however, not general solutions to the NSIS/SIS complexity problem.[6]

Second, there is no general way to automate the creation of test harnesses for all software testing regimes. This implies that the test-harness approach at the very least is not a general way of eliminating the NSIS/SIS distinction, even though it might be a labor-leveraging approach for testing *some* software systems.

**Types of numerical error in software**

There are at least three distinctive kinds of numerical errors in software: those arising from mapping real-valued quantities into digital discrete-state machines, errors arising from approximation of specific numerical operations, and numerical type-conversion errors. We sketch these in turn.

**Mapping real-valued quantities to discrete-state machines.** Many scientific theories are expressed in terms of real-valued quantities (quantities whose magnitudes are real numbers). However, because digital computing machinery has no way of representing real-valued quantities exactly (technically, computing machinery can represent, at best, integers), computing representations of real values are integer approximations of real values. At the simplest level, these approximations give rise to round-off and truncation errors.

**Errors arising from approximations of numerical operations**. Operations on approximated real values, *as operations*, generate distinctive kinds of errors. Any arithmetic operation, such as multiplication, on two approximated real-valued operands, generates error that is a function of that operation as such, and of the uncertainty in the operands. If not comprehensively managed, sequences of simple arithmetic operations (which typically occur in modern computing regimes at the rate of $10^9 – 10^{18}$ per second) on approximations of real operations can yield results that contain no information, solely as a consequence of lack of error management of these operations.

**Numeric errors arising from type mismanagement.** Yet another kind of numeric error arises from the fact that software engineers are often enough not well versed in the approximation management that is part of the computing language they are using. In every modern programming language, one must

[6] We are grateful to Sam Arbesman for helpful discussion of this point

specify the size of machine storage allocated for a representation of numeric elements (e.g., variables and literals). Most languages provide for several different storage size-allocations (technically, *types*). For example, in both Fortran and C, we can allocate a smaller, or a larger, space to a given real ("floating-point") variable. Fortran and C enforce rules for operations (e.g., addition) that act on sets of operands that do not all have the same numeric type. Some of these rules are more or less intuitive, but some are arcane. It is uncommon for software engineers to pay much attention to numeric type differences of operands in a given operation, but inattention to such detail can lead to quite unexpected results – error by any other name. A widely used N-body gravitational simulator, *nbody6*, for example, has ~10,000 instances of mixed-numeric-type ("mixed-precision") arithmetic within its ~30,000 lines of Fortran. Not surprisingly, removing these mixed-precision operations produces software whose results are not the same as the mixed-precision version (Horner 2013).

*Numerical analysis* (Leader 2004) is the study of error propagation in numeric computation. Numerical analysis was once a required part of the undergraduate computer science curriculum, but it has mostly been relegated to a specialty in finite mathematics. As a result, the management of errors arising from chains of operations is often ignored in practice.

The above kinds of numeric error can in principle be partially mitigated under a robust theory of approximation. The severity of problems generated by the types of numeric errors mentioned above when they occur in untested parts of software depends on the consequences of that software's behavior, and can range from trivial to profound, as noted above. A general theory of the consequences of untested software paths is highly desirable, but beyond the scope of this paper.

## Conclusions

There can be little doubt that science in the industrialized world has become deeply dependent on computing technology and that this dependence has changed the practice of science in significant ways. Philosophers of science have neglected an important distinguishing feature of recent science, namely the high conditionality of software used in modern science. We have shown that the NSIS/SIS distinction can be explicitly characterized in terms of this conditionality, and more specifically, in terms of software path complexity. Software-based models can have arbitrary conditional structure, that is, can have arbitrary path complexity. In general, we have no way to bound the behavior of a software path except to explicitly test that path. Because we cannot test, or even practically characterize by conventional statistical inference theory, all possible paths in even a typical 1000-line program, it is not possible in general for us to fully characterize, or even reliably bound, the behavior of a SIS. A SIS therefore

contains, in ways an NSIS cannot, distinctive error and uncertainty modalities that present new challenges for the philosophy, and indeed, the responsible practice, of scientific inquiry.

## Acknowledgements

This work benefited from discussions with Sam Arbesman, George Crawford, Paul Humphreys, Tony Pawlicki, and Eric Winsburg. For any errors that remain, we blame the path complexity of our (biological) software.

## References

Batterman, Robert W 2009. "Idealization and modeling". *Synthese* 169 (3):427 - 446.

Black, Rex, Erik van Veenendaal, and Dorothy Graham 2012. *Foundations of Software Testing ISTQB Certification.* Cengage Learning EMEA.

Boolos G, J Burgess, and R Jeffrey 2002. *Computability and Logic* (4th ed.). Cambridge UK: Cambridge University Press.

Boschetti F, E.A. Fulton, R.H Bradbury, J. Symons 2012. "What is a Model, Why People Don't Trust Them, and Why They Should." In *Negotiating Our Future: Living scenarios for Australia to 2050*, Vol 2. Australian Academy of Science 107-119

Center for Systems and Software Engineering, University of Southern California 2013. *COCOMO II.* http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html.

Chang, C. and J.H Keisler 1990. *Model Theory*. North-Holland.

Chung Kai L. 2001. *A Course in Probability Theory*. 3rd Edition. New York: Academic Press.

Diestel, Reinhard 1997. *Graph Theory*. Springer.

Eyges, Leonard 1972. *The Classical Electromagnetic Field.* Addison-Wesley. Dover reprint. 1980.

Fewster, Mark and Dorothy Graham 1999. *Software Test Automation.* Addison-Wesley.

Frigg R and J Reiss 2009. "The philosophy of simulation: hot new issues or same old stew?" *Synthese* (169) (3), 593–613.

Graham Robert M, GJ Clancy Jr., and DB DeVaney 1973. "A software design and evaluation system". *Communications of the ACM* 16 (2), 110-116. Reprinted in E Yourdon, ed., *Writings of the Revolution.* New York: Yourdon Press, 1982. pp. 112-122.

Guala, Francesco 2002. "Models, simulations, and experiments." In *Model-Based Reasoning*. Springer, 59-74.

Gustafson, John 1998. "Computational verifiability and the ASCI Program". *Computational Science and Engineering 5*, 36-45. http://www.johngustafson.net/pubs/pub55/ASCIPaper.htm.

Halmos, Paul 1950. *Measure Theory*. D. Van Nostrand Reinhold.

Hatton, Les 1997. "The T experiments: errors in scientific software". *IEEE Computational Science and Engineering* 4, 27-38. Also available at http://www.leshatton.org/1997/04/the-t-experiments-errors-in-scientific-software/.

Hatton, Les 2013. "Power-laws and the conservation of information in discrete token systems: Part 1: General Theory". http://www.leshatton.org/Documents/arxiv_jul2012_hatton.pdf.

Hogg, Robert , Joseph McKean, and Allen Craig. 2005. *Introduction to Mathematical Statistics*. 6th edition. Pearson.

Horner, Jack 2003. "The development programmatics of large scientific codes". *Proceedings of the 2003 International Conference on Software Engineering Research and Practice,* 224-227. Athens, Georgia: CSREA Press.

_____. 2013. "Persistence of Plummer-distributed small globular clusters as a function of primordial-binary population size". *Proceedings of the 2013 International Conference on Scientific Computing*, 38-44. Athens, Georgia: CSREA Press.

Humphreys, Paul 1994. "Numerical experimentation." In *Patrick Suppes: Scientific Philosopher*. Kluwer. 103-121.

IEEE 2000. IEEE-STD-1471-2000. Recommended practice for architectural description of software-intensive systems. *http://standards. IEEE. org*.

ISO/IEC 2008. *ISO/IEC 12207:2008. Systems and software engineering — Software life cycle processes.*

Kuhn, Thomas S. 1970. *The Structure of Scientific Revolutions*. Second Edition, Enlarged. Chicago: University of Chicago Press.

Leader, Jeffrey 2004. *Numerical Analysis and Scientific Computation*. Addison Wesley.

Lounkine, Eugen 2012. "Large-scale prediction and testing of drug activity on side-effect targets." *Nature* (486) 7403, 361-367.

McCabe, Thomas 1976. "A complexity measure". *IEEE Transactions on Software Engineering* 2, 308–320. Also available at http://www.literateprogramming.com/mccabe.pdf.

National Coordination Office for Networking and Information Technology Research and Development 2013. "DoE's ASCI Program". http://www.nitrd.gov/pubs/bluebooks/2001/asci.html.

Newton 1726. *The Principia.* Edition of 1726. Trans. By A. Motte, 1848. Prometheus reprint, 1995.

Oreskes, Naomi, Kristin Shrader-Frechette, and Kenneth Belitz 1994. "Verification, validation, and confirmation of numerical models in the earth sciences." *Science* 263 (5147), 641-646.

Parker, Wendy S 2009. "II—Confirmation and Adequacy-for-Purpose in Climate Modelling." *Aristotelian Society Supplementary Volume*. Vol. 83 (1) .

Reichenbach, Hans 1958. *The Philosophy of Space and Time*. Trans. by M Reichenbach and J Freund. New York: Dover.

Schmidt, Michael, and Hod Lipson. 2009 "Distilling free-form natural laws from experimental data." *Science* 324.5923 81-85.

Symons, John (2008). Computational models of emergent properties. *Minds and Machines*, 18(4), 475-491.

Symons, John and Fabio Boschetti 2013 "How computational models predict the behavior of complex systems." *Foundations of Science*. 18, 4, 809-821

United Nations 1996. "Resolution adopted by the general assembly:50/245. Comprehensive Nuclear-Test-Ban Treaty".

Waite, William M. and Gerhard Goos. *Compiler Construction*. Springer. 1984.

Winsberg, Eric 1999. "Sanctioning models: The epistemology of simulation." *Science in Context* 12 (2), 275-292.

_____. 2003. "Simulated experiments: Methodology for a virtual world." *Philosophy of Science* 70 (1), 105-125.

Winsberg, Eric and J Lenhard 2010. "Holism and Entrenchment in Climate Model Validation." In *Science in the Context of Application: Methodological Change, Conceptual Transformation, Cultural Reorientation*. Carrier, M. and Nordmann, A., eds. Springer.