# Computational science:
# A (fixable) failure of software engineering

Harold Thimbleby

See Change Fellow in Digital Health

Current Address: 62 Cyncoed Road, Cardiff, CF23 5SH, Wales
harold@thimbleby.net
ORCID 0000-0003-2222-4243

## Abstract

*Background:* Computer code underpins modern science. While scientific models are routinely peer reviewed, the algorithms and the quality of code implementing them often avoid scrutiny; therefore the details of many scientific conclusions cannot be rigorously justified.
*Problem:* Assumptions in scientific code are hard to scrutinize as they are rarely made explicit. Additionally, both algorithms and code have bugs, unknown and accidental assumptions that have unwanted effects. Code is fallible, so any interpretation that relies on code is therefore fallible, and if the code is not clearly structured and published with adequate documentation, the code cannot be usefully scrutinized. In turn, scientific claims cannot be properly scrutinized.
*Methodology:* This paper critiques the quality of software engineering practice, particularly as relied on in leading COVID-19 pandemic research driving international public health interventions, and undertakes a pilot survey of peer reviewed computational modeling papers ($N = 32$) published in leading scientific journals.
*Results:* Scientists rarely assure the quality of code they rely on, rarely make complete code available, and rarely provide adequate documentation to understand or use their code reliably. Journals do not always have a code policy, and many have a relaxed approach.
*Solutions:* Code can be improved using software engineering. This paper argues for specific solutions:

1. Professional software engineers can help and should be involved, particularly in critical research;
2. "Software Engineering Boards" (analogous to Ethics or Institutional Review Boards) should be instigated and used;
3. Code, when used, should be considered an intrinsic part of any publication, and therefore should be formally reviewed by competent software engineers.

The Supplemental Material provides a summary of professional software engineering best practice relevant to scientific research and publication.

"Criticism is the mother of methodology."
Robert P Abelson, *Statistics as Principled Argument* [1]

**Keywords:** Computational science; Software Engineering; Reproducibility; Scrutiny

## Author summary

Harold Thimbleby PhD, FRCP (Edinburgh), Hon. FRSA, Hon. FRCP is See Change Fellow in Digital Health at Swansea University, Wales. His research focuses on human error and computer system design, particularly for healthcare. Harold won the British Computer Society Wilkes Medal. He is emeritus Gresham Professor of Geometry (a chair founded in 1597), and has been a Royal Society-Leverhulme Trust Senior Research Fellow and a Royal Society-Wolfson Research Merit Award holder. His latest book is *Fix IT: How to see and solve the problems of digital healthcare* [2]. See his web site, `www.harold.thimbleby.net`, for more details.

## 1 Introduction

Computational methods are comparatively new, powerful, and very flexible tools for scientists. Computational methods rely on programming, but unfortunately poor programming is very easy use to apparently get results with; such results are likely to be misleading. The role of computational methods can be compared to statistics, another useful scientific tool. Poor statistics is much easier to do than good statistics, and there are many examples of science being let down by naïvely planned and poorly implemented statistics. Often scientists do not realize the limitations of their own statistical skills, so careful scientists generally work closely with professional statisticians.

In good science, all statistics, methods and results have to be reported very carefully and in precise detail. For example, a statistical claim might be summarized as follows:

> "… Bonferroni adjusted estimated mean difference between intervention-arms at 8-weeks 2.52 $(95\%\ \mathrm{CI}\ 0.78, 4.27, \mathrm{p} = 0.0009)$. Between group effect size $\mathrm{d} = 0.55$ $(95\%\ \mathrm{CI}\ 0.32, 0.77)$." [3]

This typical wording briefly summarizes confidence intervals, *p* levels, and so on, to present statistical results so the paper's claims can be seen to be complete, easy to interpret, and easy to scrutinize. It is a *lingua franca.* It may look technical, but it is written in the standard and now widely accepted form for summarizing statistics.

Scientists write like this — and conferences and journals require it — because statistical claims need to be properly accountable and documented in a clear way. Speigelhalter [4] says statistical information needs to be accessible, intelligible, assessable, and usable; he also suggests probing questions to help assess statistical quality (see Supplemental Material section 4). Results should not be uncritically accepted just because they are claimed.

The skill and effort required to do statistics so it can be communicated clearly and correctly, as above, is not to be taken for granted. Scientists work closely with competent, often specialist, statisticians who engage with the research from experiment design through to analysis and publication. Further, it is assumed that statistics will be peer reviewed, and that review will often lead to improvement.

Scientists accept that statistics is a distinct, professional science, itself subject of research and continual improvement. Among other implications of the depth and progress of the field of statistics, undergraduate statistics options for general scientists are insufficient training for rigorous work in science — their main value, perhaps, is to help scientists to understand the value of collaborating with specialist statisticians. Collaboration with statisticians is particularly important when new types of work are undertaken, where the statistical pitfalls have not already been well-explored.

Except in the most trivial of cases, all numbers and graphs, along with the statistics underlying them will be generated by computer. Indeed, computers are now very widely used, not just to calculate statistics, but to run the models (the term is defined below), do the data sampling and processing, and even to operate the sensors that generate the data that is analyzed. Computers are used to run human-participant surveys, such as web-based surveys. The data — including the databases and bibliographic sources — and code to analyze it is all stored and manipulated on computers. Computers even help with the word processing and typesetting of research.

In short, computers, data, and computer code are central to modern science. However, using code raises many critical questions: formats, backup, cyber-vulnerability, version control, integrity checking (e.g., managing human error), auditing, debugging and testing, and more. Software code, like statistics, is also subject to unintentional bias [5, 6]. All these issues are non-trivial concerns requiring technical expertise to manage well.

A common oversight in scientific papers is to present a model, such as a set of differential equations, but omit how that model is transformed into code that generates the results the paper summarizes; if so, the code may have problems that cannot be identified as there is no specification to reference it to.

Failure to properly document and explain computer code undermines the scientific value of the models and the results they generate, in the same way as failure to properly articulate statistics undermines the value of any scientific claims. Indeed, as few papers use code that is as well-understood and as well-researched as standard statistical procedures (such as Bonferroni corrections), the scientific problems of poorly reported code are widespread.

We would not believe a statistical claim that was obtained from some *ad hoc* analysis with a new fangled method devised just for one project — instead, we demand statistics that is recognizable, even traditional, so we are assured we understand what has been done and how reliable results were obtained. An interesting overlap with statistical and software engineering sloppiness is the many papers that just disclose as part of their methodology that they used a particular statistical package (e.g., "Data analyses were performed using SAS 9.2 (SAS Institute, Cary, North Carolina, USA)," yet *how* those analyses might have been performed is not discussed. The problem is that admitting using SAS 9.2 or any other named system does not help scrutiny, as such systems can do anything with the data. A reviewer, if nobody else, needs to actually examine the statistical code itself and its documentation to assess whether the analysis presented in the paper is appropriate and suffciently reliable.

It is recognized that to make critical claims, models need to be run under varying assumptions [7], yet somehow it is easy to overlook that the code that implements the models also needs to be carefully tested under varying assumptions to uncover and fix bugs and biases. Being able to understand (at least in principle) the exact code used in implementing a model is critical to having confidence in the results that rely on it. Unfortunately code is rarely considered a substantial part of the science to which it contributes.

This paper reviews a selection of papers in leading international journals, and finds that both papers and journal policies take code for granted. This paper then argues that, just as is routine for statistics, code and results from code (and the data it is run on) need to be discussed and presented in a way that properly assure belief in any claims derived from using them. Specifically, code should be developed and discussed in a sufficiently professional, rigorous, and recognizable way that is able to support clear presentation and scrutiny. Developing justifiably reliable code is the concern of the field of *software engineering*, which will be discussed further below, as well as more substantially in this paper's Supplemental Material).

This paper shows that unreliable computational dependencies in science are widespread. Furthermore, code is rarely published in any useful form or professionally scrutinized, and therefore the code itself does not contribute to furthering reliable science, for instance through replication or reproduction. In short, the quality of much modern science seems to be undermined because the code it relies on is rarely of adequate quality *for the uses to which it is put.*

This paper explores the extent of these software engineering problems in published science. The paper additionally suggests some ways forward. The Supplemental Material is an integral part of the suggested solutions. In particular, the Supplemental Material section 4 summarizes Speigelhater's uncontroversial statistics probes and draws explicit analogies for the critical assessment of the quality of scientific code.

## (1.a) The role of code in science and scientific publication

For the purposes of this paper, models map theory and parameters to describe phenomena, typically to make predictions or to test and refine the models. With the possible exception of theoretical research, all but the simplest models require computers to evaluate; indeed even theoretical mathematics is now routinely performed by computer systems.

Whereas the mathematical form of a model may be concise and readily explained, even a basic computational representation of a model can easily run to thousands of lines of code, and its parameters — its data — may also be extensive. The chance that a thousand lines of hand-written code is error free is negligible, and therefore good practice demands that checks and constraints should be applied to improve its reliability. How to do this well is the concern of software engineering, and is discussed throughout this paper.

While scientific research may rely on relatively easily-scrutinized mathematical models, or models that seem in principle easy to mathematize, the models that are run on computers to obtain the results published are sometimes not disclosed, and even when they are (certainly in all cases reviewed later in this paper) they are long, complex, inscrutable and (our survey shows) lack adequate documentation. Therefore the models are very likely to be unreliable *in principle.* If code is not well-documented, this is not only a problem for reviewers and scientists reading the research to understand the intention of the code, but it also causes problems for the original researchers themselves: how can they understand its thinking well enough (e.g., a few weeks or months later) to maintain it correctly if has not been clearly documented? Without documentation, including a reasoned case to assure that the approach taken is sound [8], how do researchers, let alone reviewers, know exactly what they are doing?

Without substantial documentation it is impossible to scrutinize code properly. Consider just the single line "`y = k*exp(x)`" where there can be *no* concept of its correctness *unless* there is also an explicitly stated relation between the code and the mathematical specifications. What does it mean? What does `k` mean — is it a basic constant or the result of some previous complex calculation? Does the code mean what was intended? What are the assumptions on `y` and do they hold invariantly? Moreover, as code generally consists of thousands of such lines, with numerous inter-dependencies, plus calling on many complex libraries of support code, it is inevitable that the *collective* meaning will be unknown. A good programer would (in the example here) at least check that `k` and `x` are in range and that `k*exp` was behaving as expected (e.g., in case of overflow).

Without explicit links to the relevant models (typically mathematics, depending on the claims), it is impossible to reason whether any code is correct, and in turn it is impossible to scientifically scrutinize results obtained from using the code. Not providing code and documentation, providing partial code, or providing code without the associated reasoning is analogous to claiming “statistical results are significant” without any discussion of the relevant methods and statistical details that justify making such a claim. If such an unjustified pseudo-statistical claim was made in a scientific paper, a reviewer would be justified in asking whether a competent experiment had even been performed. It would be generous to ask the author to provide the missing details so the paper could be better reviewed on resubmission.

Contrary to the views expressed in the present paper, some authors have asserted that the purpose of code is to provide insight into models, rather than precise (generally numerical) analyses summarising data [9] — code can also be used to analyze and critique scientific theories directly. If code is inadequate, “insights” it provides will be flawed, and flawed in unquantified and unknown ways. Indeed, none of the papers sampled (described below in section (3.a)) claimed their papers were using code for insight; all claimed, explicitly or implicitly, that their code outputs were integral to their peer reviewed results.

Clearly, like statistics, programming (coding) can be done poorly and reported poorly, or it can be done well and reported well — and any mix between these extremes. The question is whether it matters, *when* it matters, and when it does, *what* can be done to *appropriately* help improve the quality of code (and discussions about the code) in scientific work.

## (1.b) The normal scientific emphasis on data

Data has been at the center of science, certainly since the earliest days of astronomy collecting planetary and other information. Today it is widely recognized that lack of accessible and usable data that has already been collected limits the progress of science. Low quality data and poor access to data causes reproducibility problems, an increasingly recognized problem — in 2015 it was estimated that $28B a year is spent on preclinical research alone that is not reproducible [10].

There are many current activities to proceduralize and standardize the more effective curation and use of data, such as the FAIR principles (Findable, Accessible, Interoperable and Reusable — both for machines and for people) for scientific data management and stewardship [11], and in the development of journal and national funder policies. For example, the 2022 update to the US National Institutes of Health data (not code) policies [12] is described as a “seismic mandate” by *Nature* [13] in its attempt to improve reproducibility and open science.

On the whole, these cost estimates and initiatives under-play or ignore the role of software as a critical part of at least some scientific endeavor: code has become the new laboratory for almost all science. The role of code specifically in modeling is discussed throughout this paper; without bespoke code, proposed models (unless very abstract) cannot make a quantifiable contribution to the literature. More generally, much data is embedded in code, and in the limit code and data are indistinguishable (see Supplemental Material). Furthermore, code has additional problems of versions and compatibility beyond those of data, for example suitable compilers to run old code may no longer be available, and — worse — programming systems may silently produce different results when used on different computers.

In general, without proper management of code — for example to detect and report version control differences — sharing code may even be counter-productive. (The data and code shared with the present paper includes cryptographic checksums; if somebody reproducing the work described here does not obtain the same checksums, then there are problems that need investigation before relying on the reproducibility of the data.)

## (1.c) The deceptive simplicity of code

It is a fallacy that programming is easy [2]. More correctly, toy programming is easy, but real programming is very difficult.

An analogy helps. Building houses is very easy — indeed, many of us have built toy Lego houses. Obviously, though, a Lego house is not a *real* house. It is not large enough or strong enough for safe human habitation! This is obvious because we can see Lego houses. In contrast to Lego, computer programs are generally invisible, and therefore the engineering problems within them are also invisible. Thus the "programming is easy" cliché is deceptive — programming appears easy *because* professional standards of building software are ignored, because people cannot see the reasons why they are needed. It is like sincerely appreciating a child's Lego building because we are not worried about subsidence, load bearing, electric shock, fire risks, water ingress, or even planning regulations.

Certainly, building is much easier and faster when such technical details are ignored, as anyone who has experienced a cowboy build may attest.

Unlike building (the Code of Hammurabi dates to around 1755BC), programming is a very new discipline, and the problems of poor programming are not widely appreciated. Relevant professional standards are not enforced. Problems for the reliability of science arise when legitimate doodling with software drifts into claiming scientific results that do not have the reliable engineering structures underpinning them, let alone the properly developed and documented archived code, to justify them. In many countries, you cannot even start to build without first having plans approved, but who writes plans for software?

Since programming appears to be so easy, developing code has low status. The developers of code are rarely acknowledged in scientific papers. The implicit reasoning is: if programming is easy, then its intellectual contribution to science is negligible, so it is not even worth citing it or acknowledging the technical contributors to it. While this view prevails, the vicious cycle is that the low status means software development is done casually, which reinforces the low status.

In reaction to this vicious cycle, there is a growing movement to cite code correctly [14], because code *is* important, particularly for reproduction, testing and extension of any scientific work.

Scientists have been collecting and analyzing data for millennia, and curating data is taken seriously as a part of normal science and peer reviewed publication. Journal policies widely require appropriate discussion of data (much like they require appropriate discussion of statistics). Journals often require archiving data in standard formats so it can be accessed for further scientific work, such as reproduction.

Unfortunately, very few scientists and even fewer journals realize that data and code are theoretically and in practice indistinguishable (see Supplemental Material). Given that data and code are equivalent and interchangeable, it follows that publishing policies on data handling — and the reasons for those policies — should also apply at least as strictly to code.

## (1.d) The central role of code is ignored

It is important that experiments and analysis, such as statistical analysis, are performed reliably and ethically. There are many protocols and journal policies that enforce good practice, for example a journals often require adherence to PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) [15] for any paper performing a systematic review of the literature. Yet PRISMA, like many policies, ignores the central role of computers, and ignores the software engineering principles that assure computation is reliable and reliably reported.

There are many more styles of scientific investigation than systematic reviews, of course, but as PRISMA has had considerable development and is widely required by journals, it will serve as an example of overlooking the benefits of code in doing science.

PRISMA "was designed to help systematic reviewers transparently report why the review was done, what the authors did, and what they found," which sounds reasonable enough. PRISMA covers the review process carefully. For example, the authors should report the number of papers they included in their review. Perhaps $N = 1093$. This number is then written into their draft paper, likely in several places. As the authors reads and revises their paper, submit it and respond to peer reviewers, it is likely that the number of papers included in the survey changes, or other details may change too. The authors now have a maintenance problem: where are the numbers that have changed, and what should they now be changed to? Doing a search-and-replace, whether automated or by hand, is fraught with difficulties. What happens if 1093 is used for some other purposes as well? What happens if some of the 1093 values were written as $1,093$ and are not noticed? Then there are the Human Factors, where slips and errors will happen in this process anyway. Similar iterative revision cycles happen with any paper, not just with systematic reviews. Typos, slips during cut-and-paste, and other errors are common. They are problems across all science.

If a computer program is involved in the process (it generally is for a systematic review) then the value of $N$ could easily be stored in a file where the paper typesetting process can access it. For example, if LaTeX is the system of choice, the analysis could generate, say,

```
\newcommand{\N-papers-reviewed}{1093}
```

so that when and wherever the author writes `\N-papers-reviewed` in their paper's text, the typeset paper says 1093 or whatever the correct value happens to be at the time. If so, then *whenever* the survey is updated, the value cited in the paper is *correctly* updated without any further intervention from the author.

In general, not just numbers, but any data, text, graphs or tables, etc, from the analysis can be reliably inserted into a paper automatically.

A special case of this process is the automatic handling of bibliographies and citations: what used to be a tedious manual process is now very commonly automated and for exactly the same reasons, that is, simultaneously improving ease of use and reliability. Note that automatic bibliographies insert both free text and citation numbers: data values are arbitrary. Moreover, automatic bibliography systems also provide many useful validation checks on the data, for instance that a journal article citation actually specifies a journal name.

Increasingly, many analysis systems have themselves introduced features to automate keeping papers up to date and for reliably referring to correct values (what Software Engineers would call maintaining invariants with the data). R (e.g., using RMarkdown) and Mathematica are popular examples of such general purpose analysis systems that provide powerful features to help authors more easily produce reliable papers. In particular, they make it completely trivial to read and edit a paper, revise the analysis, redraw tables and diagrams etc, and obtain a correctly revised paper with no further manual work. Crucially, this encourages the author to explore the analysis and competing analyses more than usual, so their confidence in the final analysis and quoted results is much higher than usual. Furthermore, RMarkdown and Mathematica also have powerful mechanisms to create slide shows, and therefore the guarantees of automatic correctness transfer to presentations about the research as well.

An important consequence of using a code-based automated process is that it removes barriers to revising and improving papers. Before such methods were used, it was tedious and risked introducing new errors if one undertook to update research and papers. When the process is automated, however, there are no such barriers.

Of course errors may still occur, but there is an important and dramatic improvement. Each time a paper is processed, the *same* code is re-used. So each revision and proof reading of the paper helps detect those errors, and once any error is detected and corrected it will not arise again. Moreover, since the same potential error, if any, is repeatedly proof read and checked (it occurs in each iteration of the paper, and perhaps in many places in each version of the paper), this reduces the chances that the error is missed. This is a very different situation to the manual processing of data, where every time a paper is edited or revised, the data may be corrupted accidentally in new ways and in new places.

The present paper performed a **small** pilot survey of **32 papers** sampled from **3 journals**, **9** of which used code repositories. The repositories were most recently accessed on **8 February 2022** for automatic analysis for generating the summary data reported in this paper. All those facts, highlighted in bold here but also used throughout this paper and in the Supplemental Material, are ensured correct by the author using an automatic process (discussed in full in the Supplemental Material). One should note, as emphasized above, that as these specific values as used above also appear in many places and in many different contexts in the present paper, this significantly reduces the Human Factors problems of noticing errors.*

PRISMA says nothing about how to ensure the final results of a survey are correctly and reliably presented in a paper, despite this being one of PRISMA's explicit motivations. Ironically, many journal publishing policies (including *PLOS ONE*) forbid using such techniques (see the Supplemental Material for further discussion). Again, such rules reinforce the fallacy that code is trivial and unimportant.

## (1.e) Bugs, code and programming

Critiques of data and model assumptions are very common [17, 18] but program code is rarely mentioned. Yet data and program are formally equivalent (see further discussion in Supplemental Material, section 3). Program code has as great an affect on results as the data. Code is harder to scrutinize, which means errors in code can have more subtle effects on results.

It should be noted that almost all code contains "magic numbers" — that is, data masquerading as code. This common practice ensures that published data is very rarely all of the data because it omits the magic numbers. This emphasizes the need for repositories to require the inclusion of code so all data, including that embedded in the code, is actually available.

Bugs can be understood as discrepancies between what code is ought to do and what it actually does. Many bugs cause erroneous results, but bugs may be "fail safe" by causing a program to crash so no incorrect result can be delivered. Better, contracts and assertions are essential defensive programming technique that block compilation or block execution with incorrect results; they turn bugs into safe termination. None of the code examined for this paper includes any such techniques.

If code is not documented it cannot be clear what it is intended to do, so it is not possible to detect and eliminate bugs. Indeed, even with good documentation, *intentional bugs* will remain, that is, code that correctly implements the wrong things [2, 19] — they are bugs that were intended but were ideas based on mistaken

*For example, the author types 1093, but makes a transposition slip and writes 1039. Since the author thinks they know what they wrote, when they proof read 1039 they misread it (this is a very common problem with misspellings and grammatical ambiguities). These errors (transposition and misreading, etc) do not happen consistently each time, so the more often 1039 occurs in the paper *if generated automatically*, the more likely the errors will be addressed. See [16] for an actual example.

ideas (students and inexperienced programmers make intentional bugs all the time). For instance, in numerical modeling, using an inappropriate method can introduce errors that are not "bugs" in the narrow sense of incorrectly implementing what was wanted (e.g., ill-conditioning), but are bugs in the wider sense of producing incorrect results — that is, what was intended was wrong.

Random numbers are widely used in computational science, e.g., for simulation or for randomizing experiments. Misuse of random numbers (e.g., using standard libraries without testing them) is a common cause of bugs [20].

## (1.f) Human Factors of programming

People would generally not make coding errors if there were aware they were making errors. Unfortunately programming is a very demanding activity, which causes tunnel vision (also known as loss of situational awareness). The consequence is programmers focus on "the" problem as it appears in the code, and are unaware they are not considering wider issues. The correctness, generality, and usability of a program are therefore often unintentionally sacrificed to making code work at all.

Confirmation bias is a standard Human Factors problem [2] (which scientists have many techniques to avoid), which encourages us to perform tests that show our programs work. Instead, we should be rigorously testing ways in which programs can fail. This is exactly the same issue pointed out by Popper [21]: scientists should experiment to find reasons why hypotheses are false, and indeed use simple hypotheses that are testable. Software is really no more than a collection of sophisticated hypotheses.

Standard Human Factors mitigations for these problems include team working, with appropriate precautions to manage authority gradients (where the Human Factors oversights of the leader influence the team). Many computerized mitigations are also available — strong typing, code analyzers, formal methods, and so on. See the Supplemental Material.

Following the Dunning-Kruger Effect [22, 23], programmers over-estimate their programming skills because they do not have the skills to recognize their lack of knowledge, specifically in the present case, knowledge of basic software engineering. Dunning and Kruger go on to say,

> "People usually choose what they think is the most reasonable and optimal option [ …] The failure to recognize that one has performed poorly will instead leave one to assume that one has performed well; as a result, the incompetent will tend to grossly overestimate their skills and abilities. [ …] Not only do these people reach erroneous conclusions and make unfortunate choices, but their incompetence robs them of the metacognitive ability to realize it."

Unlike many skills (skating, brain surgery, …) programming, typical of much engineering, is one where errors can go unnoticed for long periods of time — things seem to work nicely right up to the moment they fail. The worse programmers are, the more trivial bugs they tend to make, but trivial bugs are easy to find so, ironically, being a poor programmer *increases* one's self-assessment because debugging seems very productive. It is easy for poor programmers and their associates to believe they are better than they actually are.

It sounds harsh to call programmers incompetent, but challenged with the complexity of programs and the complexity of the domains programs are applied in, we are all incompetent and succumb to the limitations of our cognitive resources, suffering blindspots in our thinking [2]. We *all* make mistakes we are unaware of. If we do not have the benefit of professional qualifications that have assessed us objectively, we generally have a higher opinion of our own competence than is justified. Moreover, if we do not work in a diverse team, nobody will ever point this out, so the potential problems it causes will never be addressed.

Everyone is subject to Human Factors (including the author of the present paper, e.g., as discussed in [16]): for instance, the standard cognitive bias of confirmation bias encourages us to look for bugs when code fails to do what is expected and then debug it to produce better results, but if code generates expected results not to bother to debug it further. This of course tends to make code increasingly conform to prior expectations, whether or not those expectations are scientifically justified. Typically, there was no prior specification of the code, so the code should be right, especially after all the debugging to make it "correct"! Thus coding routinely suffers from HARKing (Hypothesizing After the Results are Known [24]), a methodological trap widely recognized in statistics.

Computers themselves are also a part of the problem. Naïvely modifying a program (as may occur during debugging) typically makes it more complex, more *ad hoc*, and less scrutable. Programs can be written so that it is not possible to determine what they do or how they do it (whether by deliberate obfuscation, as in malware, or accidentally), except by running them, if indeed it is possible to exactly reproduce the necessary context to do so [25]. The point is, introducing bugs should be avoided so far as possible in the first place, and programs should routinely have assertions and other methods to detect those bugs that are introduced (see this paper's Supplemental Material for more discussion of standard programming methodologies).

## 2 State of the art in pandemic modeling

A review of epidemic modeling [26] says, "we use the words 'computational modelling' loosely," and then, curiously, the review discusses exclusively mathematical modeling, implying that for the authors, and for the peer reviewers, there is no conscious role for code or computation as such. It appears that the new insights, advances, rigor, and problems that computers bring to research are not considered relevant.

A systematic review [18] of published COVID models for individual diagnosis and prognosis in clinical care, including apps and online tools, noted the common failure to follow standard TRIPOD guidelines [27]. (TRIPOD guidelines ignore code completely.) The review [18] ignored the mapping from models to their implementation, yet if code is unreliable, the model *cannot* be reliably used, and cannot be reliably interpreted. It should be noted that flowcharts, which the review did consider, are programs intended for direct human use. Flowcharts, too, should be designed as carefully as code, for exactly the same reason: it is hard to program reliably.

A high-profile 2020 COVID-19 model [28, 29] uses a modified 2005 computer program [30, 31] for H5N1 in Thailand; it did not model air travel or other factors required for later western COVID-19 modeling. The 2020 model forms part of a series of papers [29–31] none of which provide details of their code.

A co-author disclosed [32] that the code was thousands of lines long and was undocumented C code. As Ferguson, the original code author, noted in an interview,

> "For me the code is not a mess, but it's all in my head, completely undocumented. Nobody would be able to use it …" [33]

This comment was made by a respected, influential world-leading scientist, with many peer-reviewed publications, a respectable *h*-index[†] of 93. Ferguson should be well

[†] *h*-index: the largest value of $h$ such that at least $h$ papers by the author have each been cited at least $h$ times. The figure cited for Ferguson was obtained from Google Scholar on 20 January 2022. (Typical $h$ values vary by discipline.)

aware of the standards of coding used in at least his own field. This comment, quoted above, is therefore likely to be representative of the standards of the field as a whole.

Ferguson's admission is tantamount to saying that the published scientific findings are not reproducible.[‡] This is problematic, especially as the code would have required many non-trivial modifications to update it for COVID-19 with different assumptions; moreover, the code would have had to have been updated very rapidly in response to the urgent COVID-19 crisis. If this C code had been made available for review, the reviewers would not have known how to evaluate it without the relevant documentation. It is, in fact, hard to imagine how a large undocumented program could have been repeatedly modified over fifteen years without becoming incoherent. If code is undocumented, there would be an understandable temptation to modify it arbitrarily to get desired results; worse, without documentation and proper commenting, it is methodologically impossible to distinguish legitimate attempts at debugging from merely fudging the results. In contrast, if code is properly documented, the documentation defines the original intentions (including formally using mathematics to do so), and therefore any modifications will need to be justified and explained — or the theory revised.

The programming language C which was used is not a dependable language; to develop reliable code in C requires professional tools and skills. Moreover, C code is not portable, which limits making it available for other scientists to use safely (C notoriously gets different results with different compliers, libraries, or hardware). The Supplemental Material discusses these issues further.

Ferguson, author of the code, says of the criticisms of his code, "However, none of the criticisms of the code affects the mathematics or science of the simulation" [35]. Indeed. The problem the current paper is raising is that the theoretical epidemiology may be fine, but if it is not mapped into code that correctly implements the models, then the program's output cannot be relied on without independent evidence. In science, this is the normal requirement of *reproducibility.*

The code in [28, 29] has been "reproduced," as reported in *Nature* [35, 36], but this so-called reproduction merely confirmed the code could be run again and produced comparable results (compared, apparently, to an Excel spreadsheet!). That can be achieved at this low level of sophistication is hardly surprising, regardless of the quality of the code. There was no scientific insight that merits the use of the word "reproduction." If reproducibility is to be a useful scientific criterion, an *independently* developed model needs to produce equivalent results (called $N$ version programming, a standard software engineering practice [37]) like public health surely requires — as, indeed, Ferguson's own influenza paper [38] argues. Meanwhile, it is a shame that using software for scientific models has enabled the bar to reproducibility, as understood by journals such as *Nature*, to be lowered to a mechanical level that is only sufficient to detect some forms of dishonesty, as opposed to methodological limitations, which is the point of reproducing work.

Because of the recognized importance of the Ferguson paper, a project started to document its code [39].[§] Documenting the code now may describe what it does, *including* its bugs, but it is unlikely to explain what it was intended to have done. If nothing else, as the code is documented, bugs will be found, which will then be fixed (refactoring), and so the belatedly-documented code will not be the code that was used in the published models. It is well-known that documenting code helps improve it, so it is surprising to find an undocumented model being used in the first place. The

[‡]A constructive discussion of software engineering approaches to reproducibility can be found in [34].

[§]The system is open source, available at URL `github.com/mrc-ide/covid-sim` version (19 July 2021). It is 25 kLOC (thousands of lines of code) written in C++ with Python, R, sh, YML/JSON, etc, not C.

revised code has now been published, and it has been heavily criticized (e.g., [40]), supporting the concerns expressed in the present paper.

Some epidemiology papers (e.g., [41]) publish models in pseudo-code, a simplified form of programming. Pseudo-code looks deceptively like real code that might be copied to try to reproduce it, but pseudo-code introduces invisible and unknown simplifications. Pseudo-code, properly used, can give a helpful impression of the overall approach of an algorithm, certainly, but pseudo-code alone is not a surrogate for code: using it is arguably even worse than not publishing code at all. Pseudo-code is not precise enough to help anyone scrutinize a model; copying pseudo-code introduces bugs. An extensive criticism of pseudo-code, and discussion of tools for reliable publication of code can be found elsewhere [42]. The Supplemental Material provides further discussion of reproducibility.

## 3 Science beyond epidemiology

Epidemiology has a high profile because of the current COVID pandemic. There are many other areas of computational science that are equally if not more critical, and many will have longer-lasting impact. Climate change modeling is one such example.

A short 2022 summary of typical problems of software engineering impacting science generally appears in *Nature* [43], describing diverse and sometimes persistent problems encountered during research in cognitive neuroscience, psychology, chemistry, nuclear magnetic resonance, mechanical and aerospace engineering, genomics, oceanography, and in migration. The paper [43] unfortunately makes some misleading comments about the simplicity of software engineering, e.g., "If code cannot be bug-free, it can at least be developed so that any bugs are relatively easy to find."

Guest and Martin [44] in another 2022 paper promote the use of computational modeling, arguing that through writing code, one debugs scientific thinking. Psychology, Guest and Martin's focus, has an interesting relationship with software, as computational models are often used to model cognition and to compare results with human (or animal) experiments [44]. In this field, the computation does not just generate results, but is used to explicitly explore the assumptions and structures of the scientific frameworks from which the models are derived. Computational models can even be used to perform experiments that would be unethical on live participants, for instance involving lesioning (damaging) artificial neural networks. It should be noted that such use of cognitive models is controversial — on the one hand, the software allows experiments to be (apparently) precisely specified and reproduced, but on the other hand in their quest for psychological realism the models themselves have become very complex and it is no longer clear what the science is precisely (e.g., ACT-R, one widely-used theory for simulating and understanding human cognition, has been under development since 1973 and is now a 120 kLOC Common LISP and Python system [45]; and of course any paper using ACT-R would require additional code on top of the basic framework).

The psychology paper [44] demonstrates building an example computational model from scratch to illustrate their own framework of computational science. In fact their example model has no psychological content: a simple numerical test is performed, but the psychology of why the result is counterintuitive — the psychological content — is not explored. Be that as it may, they develop a mathematical specification and discuss a short Python program they claim implements it.

The Python code is presented without derivation or discussion, as if software engineering is trivial. The program listed in the paper certainly runs without obvious problems (ignoring typographical errors due to the journal's publishers), but ironically the Python does not implement the mathematical specification explicitly provided for it,[¶] thus unintentionally undermining the argument of the paper.

One might argue the bug is trivial (the program prints `False` when it should print `b`), but to dismiss such a bug would be comparable to dismissing a statistical error that says $p = $ `False` which would be nonsense — if a program printed that, one would be justified in suspecting the quality of the entire program and its analyses. Inadvertently, it would seem, then, that the paper shows that just writing code does not help debug scientific thinking: instead, code must first be derived in a rigorous way and actually be correct (at least when finished). Otherwise, computational modeling with inadequate software engineering will very likely introduce errors into scientific thinking.

Code generally for any field of scientific modeling needs to be carefully documented and explained because all code has tacit assumptions, bugs and cybersecurity vulnerabilities [5, 6, 43] that, if not articulated *and properly managed*, can affect results in unknown ways that may undermine any claims. People reading the code will not know how to obtain good, let alone better results, because they do not know exactly what was intended in the first place. The problem is analogous to the problem of failing to elaborate statistical claims properly: failure to do so suggests that the claims may have unknown limitations or even downright flaws.

Even good quality code has, on average, a defect every 100 lines — and such a low rate is only achieved by experienced industrial software developers [46]. World-class software can attain maybe 1 bug per 1,000 lines of code. Code developed for experimental research purposes will have higher rates of bugs than professional industrial software, because the code is less well-defined and evolves as the researchers gain new insights into their models. In addition, and perhaps more widely recognized, code — especially but not exclusively mathematical code — can be subject to numerical errors [47]. It is therefore inevitable that typical modeling code has many bugs (reference [37] is a slightly-dated but very insightful discussion). Such bugs undermine confidence in model results.

Only if there is access to the *actual* code and data (in the specific version that was used for preparing the paper) does anyone know what researchers have done, but merely making code available (for instance, providing it in their Supplemental Material with papers, putting it in repositories, or using open source) is not sufficient. It is noteworthy that some papers disclosed that they had access to special hardware.

Some COVID-19 papers (e.g., [48]) make unfinished, incomplete code available. While some (e.g., [48, 49]) do make what they call "documented" code available, they provide no more than superficial comments. This is *not* documentation as properly understood. Such comments do not explain code, explain contracts, nor explain algorithms. Some readers of the present paper may not recognize these technical software terms; contracts, for instance, originated in work in the 1960s [50], and are now well-established practice in reliable programming (see the Supplemental Material for a checklist of many relevant, professional software engineering concepts and resources).

If full code is made available, it may be technically "reproducible," but the scientific point is to be able to understand and challenge, potentially refute, the findings; to do that, much more is required than merely being able to run the code [21, 51].

Even if a computer can run it, badly-written code (as found in *all* the research reviewed in the present paper and indeed in computer science research itself [52]) is inscrutable. Only if there is access to *adequate* documentation can anyone know what the researchers *intended* to do. Without all three (code, data, adequate documentation), there are dangers that a paper simplifies or exaggerates the results reported, and that omissions, bugs and errors in the code or data, generally unnoticed

[¶]More precisely: the program has a bug, and/or the specification given is wrong or too abstract.

| | | |
|---|---|---|
| 3 | Journals | |
| 32 | Papers: | |
| | 6 | *Lancet Digital Health* |
| | 12 | *Nature Digital Medicine* |
| | 14 | *Royal Society Open Science* |
| 264 | Published authors | |
| 341 | Published journal pages | |
| July 2020 | Sample month | |

Table 1: Overview of peer-reviewed paper sample.

| | | | |
|---|---|---|---|
| Number of papers sampled relying on code | 32 | 100% | |
| **Access to code** | | | |
| Have some or all code available | 12 | 38% | |
| Some or all code in principle available on request | 8 | 25% | |
| No code available | 12 | 38% | |
| **Evidence of basic good software engineering practice** | | | |
| Evidence program designed rigorously | 0 | 0% | |
| Evidence source code properly tested | 0 | 0% | |
| Evidence of any tool-based development | 0 | 0% | |
| Team or open source based development | 0 | 0% | |
| Other methods, e.g., independent coding methods | 1 | 3% | |
| **Documentation and comments** | | | |
| Substantial code documentation and comments | 2 | 6% | |
| Comments explain some code intent | 3 | 9% | |
| Procedural comments (e.g., author, date, copyright) | 10 | 31% | |
| No usable comments | 17 | 53% | |
| **Repository use** | | | |
| Code repository (e.g., GitHub) — 1 was empty | 10 | 31% | |
| Data repository (e.g., Dryad or GitHub) | 9 | 28% | |
| **Adherence to journal code policy (if any)** | | | |
| Papers published in journals with code policies | 26 | 81% | |
| Clear breaches of code policy (if any) | 11 | 42% | ($N = 26$) |

Table 2: Summary of survey results.

by the paper's authors and reviewers, will have affected the results they report [42].

Making outline code (including pseudo-code) available without proper documentation and without discussing its limitations is unethical: it encourages others to reproduce and build on poor work.

## (3.a) A pilot survey of peer-reviewed research relying on code

The problems of unreliable code are not limited to COVID-19 modeling papers, which, understandably, were perhaps rushed into publication. For instance, a 2009 paper reporting a model of H5N1 pandemic mitigation strategies [53] provides no details of its code. Its Supplemental Material, which might have provided code, no longer exists.

Almost all scientific papers rely on generic computer code (for statistics or for plotting graphs, and so on) but what is of interest is whether, and, if so, to what extent, code developed *specifically* as part of or to support a research contribution can be understood by colleagues and scientifically scrutinized. We therefore undertook a pilot survey of papers in leading peer reviewed journals.

| Github repository and paper citation | PDF paper | Repository code & data | | |
|---|---|---|---|---|
| | Number of pages | Number of files | Code kLOC | Data bytes |
| `AI-CDSS-Cardiovascular-Silo` [121] | 6 | 206 | 143 | 64 Mb |
| `blast-ct` [142] | 8 | 44 | 2 | 87 Mb |
| `covid-sim` [29] | 20 | 229 | 25 | 734 Mb |
| `lactModel` [132] | 13 | 20 | 2 | 165 kb |
| `LRM` [134] | 22 | 125 | 8 | 2 Mb |
| `manifold-ga` [140] | 7 | 11 | 1 | — |
| `MetricSelectionFramework` [113] | 17 | 44 | 4 | 236 kb |
| `PENet` [117] | 9 | 115 | 8 | 3 Mb |
| `philter-ucsf` [120] | 8 | 1,987 | 13 | 32 Mb |
| `PostoperativeOutcomes_RiskNet` [119] | 10 | 1 | — | — |
| `SiameseChange` [123] | 9 | 5 | 1 | 1 kb |
| **Average** ($N = 11$) | 11 | 253 | 19 | 84 Mb |

Citation numbers $> 79$ can be found in the Supplemental Material
Repository clones downloaded and automatically summarized 8 February 2022

**Table 3**: Sizes of repositories, with approximate sizes of code (in kLOC) and data for all available GitHub repositories reviewed in the survey, plus `covid-sim` [29] for comparison. Sizes are approximate because in all repositories code and data are conceptually interchangeable (an issue explained in the Supplemental Material), so choices were made in the survey to avoid double-counting. Many repositories rely on downloading additional code and data, which is not counted in the table as the additional required material is not in the repository cited in the paper. At the time of cloning and checking all repositories in February 2022, paper [119] still had nothing in its repository except a single file still saying "...code coming soon...," despite 32 months having already elapsed since the submitted paper had claimed the code could be accessed in its repository.

### (3.a.*i*) Methodology

There are many ways to do good science. Normal science routinely involves either performing new research and reporting it in the literature, or performing reviews — such as systematic reviews — of the existing literature to form a consensus view of an existing body of research. Typically, research offers new theories or results, points out previous errors, or uses statistical or qualitative methods to reduce the variability in previously reported results. In any case, the intention is to provide reliable knowledge and to bolster consensus [54].

In the present case, we have identified limitations in the methodology of earlier scientific contributions, throwing serious doubt on their soundness and reliability; furthermore, we identified critical misunderstandings in the literature citing those papers, including misunderstanding the nature of code reproducibility. (Shortly, we will present some ways forward, in section 4.)

It is possible, if not likely (e.g., given the report in *Nature*, cited above [43]), that such limitations are more widespread than the specific examples considered so far, which are anyway limited primarily to epidemiology, and to COVID research performed under extreme pandemic pressure. We now therefore undertake a small, manageable stratified sample of peer reviewed literature to explore whether the issues are more widespread.

A sample of 32 recent papers covering a broad range of science were selected. The papers were selected from the leading journals *Lancet Digital Health* ($N = 6$), *Nature Digital Medicine* ($N = 12$) and *Royal Society Open Science* ($N = 14$).

The two journals *Nature Digital Medicine* and *Lancet Digital Health* were selected as leading specialist science journals in an area where correctness of scientific modeling has safety-critical implications, and *Royal Society Open Science* was selected as a leading general science journal. All papers sampled are Open Access, although for some papers some or all of the associated data has no or restricted access, in some cases despite the relevant journal policies on code. Table 1 is an overview of the sample.

Papers were selected from the journals' July 2020 then new online listings where the paper's title implied that code had been used in the research. Commentary, correspondence, and editorials were excluded. The sample is intended to be large enough and objective enough to avoid the selection bias in the papers that motivated the current paper (the sample excludes the motivating papers discussed above as they were not published in the sampled journals), so that the sample may be considered to fairly represent what the editorial and the broader peer review community in leading journals considers to be good practice for computationally-based science. The selection criterion selected papers where the title implies the authors themselves considered code to be a significant component of the scientific contribution, and, indeed, all sampled papers relied on and assumed the quality of code used in their research.

This convenience sample may be considered to be small given the importance of the research questions and relative to the diversity and huge number of scientific papers,[‖] but …

1. the selected journals are leading peer-reviewed scientific journals that set the standards for scientific publishing practice generally (although the sample shows that code policies are not always enforced);

2. as will be clear from the following discussion, there is little variation across the sample, which implies that a larger sample would not have been productively more insightful (this view is consistent with the multi-disciplinary reports in [43], mentioned in section 3);

3. the survey is not intended to be a formal, systematic sample of scientific research in general, but is intended to be sufficient to dispel the possibility that the issues described above earlier in this paper are isolated practice unique to a few papers in epidemiology, perhaps an idiosyncrasy of a few authors in a particular field, or perhaps due to an initial chance selection bias (e.g., the Ferguson papers were reviewed above because of Ferguson's public profile and the importance of dependable pandemic research, but they might have just happened to be software engineering outliers);

4. the code/data policies of the 3 journals condoned at the time of the sample *and continue to condone* poor practice at the time of writing the present paper (February 2022) — for specific details and further explanation of the problems, see Supplemental Material section 5;

5. the fact that the specifically identified problems are elementary errors in software engineering (see the discussion in section 4) suggests more sophisticated analysis is not required;

6. finally, the present paper's LaTeX source, as well as all documented code and data, are available from a repository, which provides a convenient framework for easily refining or developing the research as may be desired (see details at the end of this paper).

[‖]Using Google Scholar it is estimated that over 40,000 papers meeting the title criteria were published in the month of July 2020.

The 32 papers surveyed cover a range of specialities, and it is unlikely that non-specialists can properly assess the code from the point of view of the specialism, not least because many of the papers sampled require specialist code libraries (and in the right combinations of versions) to be run that not everyone will have or be able to install. Code quality was therefore assessed by reading it — due to the paper authors' complex and/or narrative interpretation of data, code, data and hardware/operating system dependencies, no assessment could realistically be made whether the code provided actually reproduced a paper's specific claims. Indeed, if we trust the papers that their code was actually run and provides the results as reported, then running their code (when provided in full) would merely check the paper/code consistency but will not assess the quality or reliability of the code. Indeed, in most scientific papers there are layers of expert scientific work, interpretation and abstraction, lying between the computational models and the report in the paper.

The present paper's full methodology, data and analysis, as well as many rigorous software engineering and Human Factors techniques to help improve reliability and reproducibility used in its preparation, are provided in the Supplemental Material.

### (3.a.*ii*) Summary of results

The sample selection criteria necessarily identified scientific research with software engineering contributions.

No evidence of verification and validation was seen. There was only one example of very basic software engineering methods, namely independent coding, and even then the independent code used for testing was not uploaded to the paper's code repository, so the independent testing is not available for reviewers or readers of the paper. (See Supplemental Material for more details.)

There was no evidence of any critical assessment of code, suggesting that scientists writing papers take it for granted that their code works as they intend. No competent programmer would take it for granted that their code was correct without following rigorous methods, such as formal methods, regression testing, test driven design, etc. (See the Supplemental Material for a list of standard methods.)

Much code depended on specific software versions, specific libraries, and substantial manual intervention to compile it. All code (where actually provided) was sufficiently complex that, if it was to be used or scrutinized, required more substantial documentation than was provided.

On the whole, on the basis of the sample evidence, scientists do not make their code *usably* available, and rarely provide adequate documentation (see table 2).

With the one minor exception, no papers reported anything on any software engineering methodologies, which is astonishing given the scale of some of the software effort supporting the papers (table 3). The papers themselves, typically only a few published pages, are very brief compared to the substantial code they rely on (see table 3).

With the one exception, none of the papers used any specific software engineering methods, such as open source [55] or other standard methodologies provided in the Supplemental Material, to help manage their processes and help improve quality. Although software stability [56] is a relatively new concept, understood as methodologies, such as portability, to provide long-term value of software, it is curious that none of the papers made any attempt at stability (however understood) despite the irony that all the papers were published in archival journals.**

**Reason this paper does not directly assess the quality of software in the surveyed papers include: many papers did not provide complete software; it was not possible to find correct versions of all software systems to run the models; also, no papers provided adequate test suites so that correct operation of software could be confirmed objectively.

*Nature Digital Medicine* and *Royal Society Open Science* have clear data and code policies (see Supplemental Material section 5), but actual publishing practice falls short: 11 out of the 26 papers (42%) published in them and sampled in the survey manifestly breach their code policies. In contrast, *Lancet Digital Health*, despite substantial data policies, has no code policy at all to breach. The implication is that the fields, and the editorial expertise of leading journals, misunderstand and dismiss code policies — they (or their editors and reviewers) are technically unable to assess them. This lack of expertise is consistent with the limited awareness of software engineering best practice that is manifest in the published papers (and resources) themselves.

Code repositories were used by 10 papers (31%), though one paper in the survey claimed to have code on GitHub but there was no code in the repository, only the comment "Code coming soon…" (checked at the time of doing the review, then double-checked as detailed in the references in the Supplemental Material, as well as most recently on 8 February 2022 while checking table 3): in other words, the repository had never been used and the code could never have been looked at, let alone reviewed.[††] This is a pity because GitHub provides help and targeted warnings and hints like "No description, website, or topics provided […] no releases published." The lack of code is ironic: the paper concerned [119] has as its title "*Development and validation* of a deep neural network model […]" (our emphasis), yet it provides no code or development processes for the runnable model it claims to validate, so nobody else (including referees) can check any of the paper's specific claims.

The sizes of all GitHub repositories are summarized in table 3 (since many papers not using GitHub do not have all code available, non-GitHub code sizes are not easily compared and are not listed).

Overall, there was no evidence that any code had been developed carefully, let alone by using recognized professional software engineering methods. In particular, no papers in the survey provide any claims or evidence of effective testing, for instance with evidence that tests were run on clean builds. While it may sound unrealistic to ask for evidence on software quality in a paper written for another field of science, the need is no less than the need for standard levels of rigor in statistics reporting, as discussed in the opening of this paper.

Data repositories (the Dryad Digital Repository, Figshare or similar) were used by 9 papers to provide structured access to their data. Unlike GitHub, which is a general purpose repository, Dryad has scientifically-informed guidelines on handling data, and all papers that used Dryad provided more than just their raw data — they provided a little, sometimes substantial, documentation for their data. At the time of writing, Dryad is not helpful for managing code — its model appears to be founded on the requirement that once published papers must refer to exactly the data they used, so further refinements on the data (or code) are taboo, even with version control.

Key findings from the survey are summarized in tables 2 and 3, and are discussed in greater detail in the Supplemental Material.

## 4 Discussion

Effective use or access to quality code is evidently not routine in science. Using structured repositories that provide suggestions for and which encourage good practice (such as Dryad and GitHub), and requiring their use, would be a lever to improve the quality and value of code and documentation in published papers. The evidence suggests that, generally, some but not all manually developed code is uploaded to a

[††] GitHub records show that it had not been deleted after paper submission.

repository just before submitting the paper in order to “go through the motions.” In the surveyed papers there is no clear evidence (over the survey period) that any published code was maintained using the repositories.

The Supplemental Material provides a summary of the analysis and full citations for all papers in the sample.

## (4.a) A call to action

Computer programs are the laboratories of modern scientists, and should be used with a comparable level of care that virologists use in their laboratories — lab books and all [51] — and for the same reasons: computer bugs accidentally cultured in one laboratory can infect research and policy worldwide. Given the urgency of rigorously understanding COVID-19, any epidemic for that matter, it is essential that epidemiologists engage professional software engineers to help develop reliable laboratory methodologies. For instance, code lab books can be generated and signed easily.

Software used for informing public health policy, medical research or other medical applications is *critical software*. Professional critical software development, as used in aviation and the nuclear power industry, is (put briefly) based on *correct by construction*: [57] effectively, design it right first time, supported by numerous rigorous techniques, such as formal methods, to manage error. (See extensive discussion in this paper’s Supplemental Material.) Not coincidentally, these are *exactly* the right methods to ensure code is both dependable and scrutable. Conversely, not following these practices undermines the rigor of the science.

An analogous situation arises in ethics. There are some sorts of research that are ethically unacceptable, but few people have the objectivity and ethical expertise to make sound ethical judgements, particularly when it comes to assessing their own work. Misuse of data, exploiting vulnerable people, and not obtaining informed consent are typical ethical problems. National funders, and others, therefore require Ethics Boards to formally review ethical quality. Medical journals will not publish research that has not undergone appropriate ethical review.

Analogously, and supplementing Ethics Boards, Software Engineering Boards would authorize as well as provide advice to guide the implementation of high-quality software engineering. Just as journals require conflicts of interest statements, data availability statements, and ethics board clearance, we should move to epidemic modeling papers — and in due course, all scientific papers — being required to include Software Engineering Board clearance statements as appropriate. Software Engineers have a code of ethics that applies to *their* work in epidemic modeling [58].

The present paper did not explore data, because in almost all cases the code and data were explained so poorly and archived so haphazardly it would be impossible to know whether the author’s intentions were being followed.[‡‡] Some journals have policies that code is available (see the Supplemental Material), but they should require that code is not just available in principle but *actually works* on the relevant data. Ideally, the authors should test a clean deployed build of their code and save the results. Presumably a paper’s authors must have run their code successfully on *some* data (if any, but see section 3 in the Supplemental Material) at least once, so preparing the code and data in a way that is reproducible should be a routine and uncontentious part of the rigorous development of code underpinning *any* scientific claims. This requirement is no more unreasonable than requesting good statistics, as discussed in the opening of the paper. And the solution is the same: relevant experts — whether

[‡‡]For the present paper, all the code, data, analysis and documents are available for download in a single zip file.

statisticians or software engineers — need to be routinely available and engaged with the science. SEBs would be a straight forward way of helping achieve this.

There need to be many Software Engineering Boards (SEBs) to ensure convenient access and oversight, potentially at least one per university. Active, professional software engineers should be on these SEBs; this is not a job for people who are not qualified and experienced in the area or who are not actively connected with the true state of the art. There are many high-quality software companies (especially those in safety-critical areas like aviation and nuclear power) who would be willing and competent to help.

Open Source generally improves the quality of software. SEBs will take account of the fact that open source software enables contributors to be located anywhere, typically without a formal contractual relationship with the leading researchers. Where appropriate, then, SEBs might require *local* version control, unit testing, static analysis and other quality control methods for which the lead scientist and software engineer remain responsible, and may even need to sign off (and certainly be signed off by the SEB). Software engineering publishers are already developing rigorous badging initiatives to indicate the level of formal review of the quality of software for use in peer reviewed publications [59]. See this paper's Supplemental Material for further suggestions.

A potential argument against SEBs is that they may become onerous, onerous to run and onerous to comply with their requirements. A more mature view is that SEBs need their processes to be adaptable and proportionate. If software being developed is of low risk, then less stringent engineering is required than if the software could cause frequent and critical outcomes, say in their impact on public health policy for a nation. Hence SEBs processes are likely to follow a risk analysis, perhaps starting with a simple checklist. There are standard ways to do this, such as following IEC 61508:2010 [60, 61] or similar. Following a risk analysis (based on safety assurance cases, controlled documents and so on, if appropriate to the domain), the Board would focus scrutiny where it is beneficial without obstructing routine science.

A professional organization, such as the UK Royal Academy of Engineering ideally working in collaboration with other national international bodies such as IFIP, should be asked to develop and support a framework for SEBs. SEBs could be quickly established to provide direct access to mature software engineering expertise for both researchers and for journals seeking competent peer reviewers. In addition, particularly during a pandemic, SEBs would provide direct access to their expertise for Governments and public policy organizations. Given the urgency, this paper recommends that *ad hoc* SEBs should be established for this purpose.

SEBs are a new suggestion, providing a supportive, collaborative process. Methodological suggestions already made in the literature include open source and specific software engineering methodologies to improve reproducibility [34, 55]. While [62] provides an insightful framework to conceptualize approaches and compare their merits, there is clearly scope for much further research to provide an evidence base to motivate and assess appropriate interventions to help scientists do more rigorous and effective software engineering to support their research and publishing. These and further issues are discussed at greater length in the Supplemental Material.

### (4.b) Action must be interdisciplinary

Code is only part of science, and only one critical factor in the wider reproducibility crisis: SEBs must work with — and be engaged by — other reproducibility initiatives.

Relying on Software Engineering Boards (SEBs) *alone* would continue one of the current besetting problems about the role of code. The conventional view is that scientists do the hard work compared to the "easy" coding work[†] so they just need to tell programmers what to do. This is the view expressed by Landauer in his classic book *The Trouble with Computers* [63, 64], where he argues that the trouble with computers, which he explores at some length, is that we need to spend more effort in working out what computers should do (i.e., do the science better) and then just tell programmers to do *that*.

On the contrary, competent software engineers have insights into the logic, coherence, complexity, and computability of what they are asked to do, and often that logic needs refining or optimizing. In other words, the software engineers can bring important insights back into the science, hence improving or changing the questions and assumptions the science relies on. This insight was widely recognized in the specialist area of numerical computation: "here is a formula I want you to just code up" … "but that is ill-conditioned, there is no good answer." Ideally, then, it is not just a sequential process of science → code → results, but an iterative cycle of collaboration and growing mutual understanding.

In short, the way SEBs work and are used is crucial to their success. Software engineers can help improve the science, so it is not just a matter of asking a SEB whether some coding practices (like documentation) are satisfactory, but whether the SEB has insights into the science itself too. Most effectively, this requires interdisciplinary working practices (science plus software engineering) with mutual respect for their contributing expertise.

## (4.c) SEBs could be enforced

It is routine for funders and journals to require appropriate ethical processes and ethical statements, typically specifying data security and confidentiality, appropriate handling of vulnerable participants, consent, and so on. It would be easy for funders and journals to require equivalent types of statements on software engineering.

## (4.d) Suggestions for further work

Although this study of software engineering in scientific research (specifically, in peer reviewed publications) was necessarily interpretive in nature, the corpus of data (namely, the selected 32 papers) was rigorously gathered so that it could be later updated or extended in size or scope. However, the insights appear to be general.

Further work to extend the scope of the survey beyond the basic requirements of the present paper is of course worthwhile, but the following cases (listed in the next few paragraphs) suggest that the problem is widespread. We argue, then, that we should be focusing effort on avoiding problems and considering proposed solutions (see section (4.a)), not just assessing the problems highlighted in this paper with increasing scale or rigor.

1. The journal *The Lancet* published and then subsequently retracted a paper on using hydroxychloroquine as a treatment for COVID [65]. The paper was found to rely on fraudulent data [66, 67]. *The Lancet* subsequently tightened its data policies [68], for instance to require that more than one author must have directly accessed and verified the data reported in the manuscript. Curiously, the original (now retracted) paper declares

   > "… all authors participated in critical revision of the manuscript for important intellectual content. MRM and ANP supervised the study. All

[†]The programming is easy fallacy is refuted in sections (1.c) & (1.d).

> authors approved the final manuscript and were responsible for the decision to submit for publication."

which seems to suggest that several original authors of the paper would have been happy to make the new declarations — and, of course, if there is fraud (as was established in this case) it seems likely that authors who make the new declarations of accessing and verifying data are unlikely to make reliable declarations.

*The Lancet* still has no code publication policy (see the Supplemental Material), and for more than one author to have "direct access" to the data they are very likely to access the data through the same code. If the code is faulty or fraudulent, an additional author's confirmation of the data is insufficient, and there is at least as much reason for code to be fraudulent (not least because code is much harder to scrutinize than data). Code needs more than one author to check it, and ideally reviewers independent of the authors so they do not share the same assumptions and systems (for instance shared libraries, let alone potential collusion in fraud).

2. In 2020 the *Journal of Vascular Surgery* published a research paper [69], which had to be retracted on ethical grounds [70, 71]: it was a naïve study and the editorial process was unaware of digital norms. Notably, the paper fails to provide access to its anonymized data (with or without qualification), and fails to define the data anonymization algorithm, and also fails to even mention the code that it developed and used to perform its study. The journal's data policy is itself very weak (the authors "should consider" including a footnote to offer limited access to the data) and, despite basic statistics policies, it has no policy at all for code (see Supplemental Material section 5). Ironically, the retracted article [69] is still online (as of August 2020) with no reference to any editorial statement to the effect that it has been retracted, despite this being trivial — and necessary — to achieve in the widely-accessed online medium.

3. Medical research often aims to establish a formula to define a clinical parameter (such as body mass index, BMI) or to specify an optimal drug dose or other intervention for treatment. These formulas, for which there is conventional clinical evidence, are often used as the basis for computer code that provides advice or even directly controls interventions. Unfortunately a simple formula as may be published in a medical paper is *never* sufficient to specify code to implement it safely. For example, clinical papers do not need to evaluate or manage user error when operating apps, and therefore the statistical results of the research will be idealistic compared to the outcomes using an app under real conditions — which is what the clinical research is supposedly for. A widespread bug (and its fix) that is often overlooked is discussed in [72]; the paper includes an example of a popular clinical calculator (based on published clinical research) that calculated nonsense, and potentially dangerous, results. The paper [73] summarizes evidence that such bugs, ignored by the clinical research literature, are commonplace in medical systems and devices.

## 5 Conclusions

We need to improve the quality of software engineering that supports science. While this paper was originally motivated by Ferguson's public statements (e.g., [32, 33]), the wider evidence reviewed shows that current coding practice makes for poor science. In a pandemic, scientific modeling, such as epidemiological modeling, track and trace [74], modeling COVID mutation pressures against vaccine shortages and delays between vaccinations [75], etc, drive public policy and have a direct impact on quality of life.

The challenge to scientific research is to manage software development to reduce the unnoticed and unknown impacts of bugs and poor programming practices. Computer code should be explicit, accessible (well-structured, etc), and properly documented. Papers should be explicit on their software methodologies, limitations and weaknesses, just as Whitty expressed more generally about the standards of science [7]. Professional software methodologies should not be ignored.

Unfortunately, while programming is easy, and is often taken for granted, programming *well* is very difficult [2]. We know from software research than ordinary programming is very buggy and unreliable. Without adequately specified and documented code and data, research is not open to scrutiny, let alone proper review, and its quality is suspect. Some have argued that availability of code and data ensure research is reproducible, but that is naïve criterion: computer programs are easy to run and reproduce results, but being able to reproduce something of low quality does not make it more reliable [21, 42, 76].

Software Engineering Boards (as introduced in this paper) are a straight forward, constructive and practical way to support and improve computer-based science. This paper's Supplemental Material summarizes the relevant professional software engineering practice that Software Engineering Boards would use, including discussing how and why software engineering helps improve code reliability, dependability, and quality.

## Supporting information

**Acknowledgments** The author is very grateful for comments from: Ross Anderson, Nicholas Beale, Ann Blandford, Paul Cairns, Rod Chapman, José Corréa de Sà, Paul Curzon, Jeremy Gibbons, Richard Harvey, Will Hawkins, Ben Hocking, Daniel Jackson, Peter Ladkin, Bev Littlewood, Paolo Masci, Stephen Mason, Robert Nachbar, Martin Newby, Patrick Oladimeji, Claudia Pagliari, Simon Robinson, Jonathan Rowanhill, John Rushby, Susan Stepney, Prue Thimbleby, Will Thimbleby, Martyn Thomas, and Ben Wilson provided very helpful comments.

**Ethics** This article presents research with ethical considerations but does not fall within the usual scope of ethics policies.

**Data and code access** There is a full discussion of the methodology of this paper and its benefits in the Supplemental Material, section 3. The Supplemental Material also presents all raw data in tabular form. All material is available for download at URL `github.com/haroldthimbleby/Software-Enginering-Boards` (which has been tested in a clean build, etc). The data is encoded in JSON. JavaScript code, conveniently in the same file as the JSON data, checks (with 30 possible types of error report) and converts the JSON data into LaTeX number registers and summary tables, etc, thus making it trivial to typeset all results reliably in this paper and in its Supplemental Material *directly* from the automatic data analysis (e.g., see table 2 in this paper). The Supplemental Material describes how *PLOS ONE* and other journal publishing policies unfortunately do not support such automatic approaches, undermining the benefits of reliable reproducibility.

In addition, a standard CSV file is generated in case this is more convenient to use, for instance to browse in Excel or other spreadsheet application.

**Author contribution** Harold Thimbleby is the sole author. An preliminary outline of this paper, containing no supplementary material or data, was submitted to the UK Parliamentary Science and Technology Select Committee's inquiry into UK Science, Research and Technology Capability and Influence in Global Disease Outbreaks, under reference LAS905222, 7 April, 2020. The evidence, which was not peer reviewed and is only available after an explicit search, briefly summarizes the case for Software Engineering Boards, but without the detailed analysis and case studies of the literature, etc, that are in the present paper. It is available to the public [77, 78].

**Competing interests** The author declares no competing interests.

**Funding** This work was jointly supported by See Change (M&RA-P), Scotland (an anonymous funder), by the Engineering and Physical Sciences Research Council [grant EP/M022722/1], and by the Royal Academy of Engineering through the Engineering X Pandemic Preparedness Programme [grant EXPP2021\1\186]. The funders had no involvement in the research or in this paper.

## References

1. Abelson RP. Statistics as Principled Argument. Lawrence Erlbaum Associates; 1995.
2. Thimbleby H. Fix IT: How to see and solve the problems of digital healthcare. Oxford University Press; 2021.
3. Richards D, Enrique A, Eilert N, Franklin M, Palacios J, Duffy D, et al. A pragmatic randomized waitlist-controlled effectiveness and cost-effectiveness trial of digital interventions for depression and anxiety. Nature Digital Medicine. 2020;3(85). doi:10.1038/s41746-020-0293-8.
4. Speigelhalter D. The Art of Statistics: Learning from Data. Pelican Books; 2019.
5. Shneiderman B. Opinion: The dangers of faulty, biased, or malicious algorithms requires independent oversight. Proceedings National Academy of Sciences. 2016;113(48):13538–13540. doi:10.1073/pnas.1618211113.
6. Friedman B, Nissenbaum H. Bias in Computer Systems. ACM Transactions on Information Systems. 1996;14(3):330–347. doi:10.1145/230538.230561.
7. Whitty CJM. What makes an academic paper useful for health policy? BMC Medicine. 2015;13:301. doi:10.1186/s12916-015-0544-8.
8. Habli I, Alexander R, Hawkins R, Sujan M, McDermid J, Picardi C, et al. Enhancing COVID-19 decision making by creating an assurance case for epidemiological models. BMJ Health & Care Informatics. 2020;27(e100165):1–5. doi:10.1136/bmjhci-2020-100165.
9. Kelly D, Sanders R. Assessing the Quality of Scientific Software. First International Workshop on Software Engineering For Computational Science and Engineering [79]; 2008. Available from: URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.526.5076`.
10. Freedman1 LP, Cockburn IM, Simcoe TS. The Economics of Reproducibility in Preclinical Research. PLOS Biology. 2015;13(6):e1002165. doi:10.1371/journal.pbio.1002165.

11. Wilkinson MD, Dumontier M, Aalbersberg IJ, Appleton G, Axton M, Baak A, et al. The FAIR Guiding Principles for scientific data management and stewardship. Scientific Data. 2016;3(160018):1–9. doi:10.1038/sdata.2016.18.

12. of Health NI. Final NIH Policy for Data Management and Sharing. vol. NOT-OD-21-013. Office of The Director, National Institutes of Health; 2020; effective January 25, 2023. Available from: URL `grants.nih.gov/grants/guide/notice-files/NOT-OD-21-013.html`.

13. Kozlov M. NIH issues a seismic mandate: share data publicly. Nature. 16 February 2022;.

14. Katz DS, Hong NPC, Clark T, Muench A, Stall S, Bouquin D, et al. Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved] Previously titled: "The importance of software citation". F1000Research. 2021;9(1257). doi:10.12688/f1000research.26932.2.

15. Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, et al. The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. Systematic Reviews. 2021;10(89). doi:10.1186/s13643-021-01626-4.

16. Thimbleby H. Human Factors and missed solutions to Enigma design weaknesses. Cryptologia. 2016;40(2):177–202. doi:10.1080/01611194.2015.1028680.

17. Sayburn A. Covid-19: Experts question analysis suggesting half UK population has been infected. BMJ. 2020;368:m1216. doi:10.1136/bmj.m1216.

18. Wynants L, Van Calster B, Bonten MMJ, Collins GS, Debray TPA, De Vos M, et al. Prediction models for diagnosis and prognosis of covid-19 infection: Systematic review and critical appraisal. BMJ. 2020;369(m1328). doi:10.1136/bmj.m1328.

19. Jackson D. The Essence of Software. Princeton University Press; 2021.

20. Knuth DE. The Art of Computer Programming (Seminumerical algorithms). vol. 2. 3rd ed. Addison-Wesley; 1998.

21. Popper KR. Conjectures and Refutations: The Growth of Scientific Knowledge. 2nd ed. Routledge; 2002.

22. Kruger J, Dunning D. Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments. Journal of Personality and Social Psychology. 1999;77(6):1121–1134. doi:10.1037/0022-3514.77.6.1121.

23. Nuhfer E, Fleisher S, Cogan C, Wirth K, Gaze E. How Random Noise and a Graphical Convention Subverted Behavioral Scientists' Explanations of Self-Assessment Data: Numeracy Underlies Better Alternatives. Numeracy. 2017;10(1):4. doi:10.5038/1936-4660.10.1.4.

24. Kerr NL. HARKing: Hypothesizing after the results are known. Personality and Social Psychology Review. 1998;2(3):196–217. doi:10.1207/s15327957pspr0203_4.

25. Thimbleby H, Anderson SO, Cairns P. A Framework for Modelling Trojans and Computer Virus Infection. Computer Journal. 1999;41:444–458. doi:10.1093/comjnl/41.7.444.

26. Heesterbeek H, Anderson RM, Andreasen V, Andreasen V, Bansal S, Angelis DD, et al. Modeling infectious disease dynamics in the complex landscape of global health. Science. 2015;347(6227):265–270. doi:10.1126/science.aaa4339.

27. Moons KG, Altman DG, Reitsma JB, Ioannidis JP, Macaskill P, Steyerberg EW, et al. Transparent Reporting of a multivariable prediction model for Individual Prognosis or Diagnosis (TRIPOD): Explanation and elaboration. Annals of Internal Medicine. 2015;162(1):W1–73. doi:10.7326/M14-0698.

28. Adam D. Modelling the pandemic: The simulations driving the world's response to COVID-19. Nature. 2020;580:316–318. doi:10.1038/d41586-020-01003-6.

29. Ferguson NM, Laydon D, Nedjati-Gilani G, Imai N, Ainslie K, Baguelin M, et al.. Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand; 16 March 2020. Available from: URL `www.imperial.ac.uk/media/imperial-college/medicine/sph/ide/gida-fellowships/Imperial-College-COVID19-NPI-modelling-16-03-2020.pdf`.

30. Ferguson NM, Cummings DAT, Fraser C, Cajka JC, Cooley PC, Burke DS. Strategies for mitigating an influenza pandemic. Nature. 2005;437:209–214. doi:10.1038/nature04017.

31. Ferguson NM, Cummings DAT, Fraser C, Cajka JC, Cooley PC, Burke DS. Strategies for mitigating an influenza pandemic. Nature. 2006;442:448–452. doi:10.1038/nature04795.

32. Ferguson N. Tweet; 22 March 2020. Available from: URL `twitter.com/neil_ferguson/status/1241835454707699713`.

33. Leake J. Neil Ferguson interview: No 10's infection guru recruits game developers to build coronavirus pandemic model. The Sunday Times. 29 March 2020;.

34. Hinsen K. Software Development for Reproducible Research. Computing in Science & Engineering. 2013;15(4):60–63. doi:10.1109/MCSE.2013.91.

35. Chawla DS. Critiqued coronavirus simulation gets thumbs up from code-checking efforts. Nature. 8 June 2020;582:323–324.

36. Scheuber A, van Elsland SL. Codecheck confirms reproducibility of COVID-19 model results; 1 June 2020. Available from: URL `www.imperial.ac.uk/news/197875/codecheck-confirms-reproducibility-covid-19-model-results`.

37. Hatton L, Roberts A. How accurate is scientific software? IEEE Transactions on Software Engineering. 1994;20(10):785–797. doi:10.1109/32.328993.

38. Halloran ME, Ferguson NM, Eubank S, Longini IM, Cummings DAT, Lewis B, et al. Modeling targeted layered containment of an influenza pandemic in the United States. Proceedings of the National Academy of Sciences. 2008;105(12):4639–4644. doi:10.1073/pnas.0706849105.

39. Ferguson N. Tweet; 22 March 2020. Available from: URL `twitter.com/neil_ferguson/status/1241835456947519492`.

40. Richards D, Boudnik K. Neil Ferguson's Imperial model could be the most devastating software mistake of all time. The Telegraph. 16 May 2020;

41. Zlojutro A, Rey D, Gardner L. A decision-support framework to optimize border control for global outbreak mitigation. Nature Scientific Reports. 2019;9(2216). doi:10.1038/s41598-019-38665-w.

42. Thimbleby H, Williams D. A tool for publishing reproducible algorithms & A reproducible, elegant algorithm for sequential experiments. Science of Computer Programming. 2018;156:45–67. doi:10.1016/j.scico.2017.12.010.

43. Perkel JM. How to fix your scientific coding errors. Nature. 2022;602:172–173. doi:10.1038/d41586-022-00217-0.

44. Guest O, Martin AE. How Computational Modeling Can Force Theory Building in Psychological Science. Perspectives on Psychological Science. 2021;16(4):789–802. doi:10.1177/1745691620970585.

45. ACT-R Research Group. ACT-R; 2022. Available from: URL `act-r.psy.cmu.edu/software`.

46. Ladkin PB, Littlewood B, Thimbleby H, Thomas M. The Law Commission presumption concerning the dependability of computer evidence. Digital Evidence and Electronic Signature Law Review. 2020;17. doi:10.14296/deeslr.v17i0.5143 (NB See URL `journals.sas.ac.uk/deeslr/article/view/5143` until the DOI is resolved.).

47. Hamming RW. Numerical Methods for Scientists and Engineers. Dover Publications Inc.; 1987.

48. Kissler SM, Tedijanto C, Goldstein E, Kissler SM, Tedijanto C, Goldstein E, et al. Projecting the transmission dynamics of SARS-CoV-2 through the postpandemic period. Science. 2020;doi:10.1126/science.abb5793.

49. Verity, R, Okell, C L, Dorigatti, Winskill IP, et al. Estimates of the severity of coronavirus disease 2019: A model-based analysis. Lancet. 2020;doi:10.1016/S1473-3099(20)30243-7.

50. Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM. 1969;12(10):576–580. doi:10.1145/363235.363259.

51. Schnell S. Ten Simple Rules for a Computational Biologist's Laboratory Notebook. PLoS Computational Biology. 2015;11(9):e1004385. doi:10.1371/journal.pcbi.1004385.

52. Thimbleby H. Give your computer's IQ a boost — *Journal of Machine Learning Research*. Times Higher Education Supplement. 9 May, 2004;.

53. Sander B, Nizam A, Garrison Jr LP, Postma MJ, Halloran ME, Longini Jr IM. Economic evaluation of influenza pandemic mitigation strategies in the us using a stochastic microsimulation transmission model. Value Health. 2009;12(2):226–233. doi:10.1111/j.1524-4733.2008.00437.x.

54. Ziman JM. Reliable Knowledge: An Exploration of the Grounds for Belief in Science. 1st ed. Cambridge University Press; 1979.

55. Fomel S. Reproducible Research as a Community Effort: Lessons from the Madagascar Project. Computing in Science & Engineering. 2015;17(1):20–26. doi:10.1109/MCSE.2014.94.

56. Salama M, Bahsoon R, Lago P. Stability in Software Engineering: Survey of the State-of-the-Art and Research Directions. IEEE Transactions on Software Engineering. 2021;47(7):1468–1510. doi:10.1109/TSE.2019.2925616.

57. Woodcock JCP, Larsen PG, Bicarregui JC, Fitzgerald JS. Formal methods: Practice and experience. ACM Computing Surveys. 2009;41(4). doi:10.1145/1592434.1592436.

58. ACM. Code of Ethics and Professional Conduct. ACM; 2020. Available from: URL `www.acm.org/code-of-ethics`.

59. ACM. Artifact Review and Badging — Current. vol. Artifact Review and Badging Version 1.1. ACM; 2020. Available from: URL `www.acm.org/publications/policies/artifact-review-and-badging-current`.

60. Redmill F. Understanding the Use, Misuse and Abuse of Safety Integrity Levels. In: Lessons in System Safety, Eighth Safety-critical Systems Symposium; 2000.Available from: URL `homepages.cs.ncl.ac.uk/felix.redmill/publications/1%20SILs.pdf`.

61. IEC 61508:2010 CMV Commented version, Functional safety of electrical/electronic/programmable electronic safety-related systems; 2010. Available from: URL `webstore.iec.ch/publication/22273`.

62. Stol KJ, Fitzgerald B. The ABC of Software Engineering Research. ACM Transactions on Software Engingeering and Methodology. 2018;27(3). doi:10.1145/3241743.

63. Landauer TK. The Trouble with Computers: Usefulness, Usability, and Productivity. MIT Press; 1995.

64. Thimbleby H. The Trouble with Computers: Usefulness, Usability, and Productivity (by Thomas K. Landauer). Computational Linguistics. 1996;22(2):265–276.

65. Mehra MR, Desai SS, Ruschitzka F, Patel AN. RETRACTED: Hydroxychloroquine or chloroquine with or without a macrolide for treatment of COVID-19: A multinational registry analysis. The Lancet. 2020; p. 1–10. doi:10.1016/S0140-6736(20)31180-6.

66. Servick K, Enserink M. A mysterious company's coronavirus papers in top medical journals may be unraveling. Science. 2020;doi:10.1126/science.abd1337.

67. Servick K. COVID-19 data scandal prompts tweaks to elite journal's review process. Science. 2020;doi:10.1126/science.abe8656.

68. The Editors. Learning from a retraction. The Lancet. 2020;396:799. doi:10.1016/S0140-6736(20)31958-9.

69. Hardouin S, Cheng TW, Mitchell EL, Raulli SJ, Jones DW, Siracuse JJ, et al. Prevalence of unprofessional social media content among young vascular surgeons. Journal of Vascular Surgery. 2020;72(2):667–671. doi:10.1016/j.jvs.2019.10.069.

70. Baumann J. #MedBikini Backlash Exposes Research Ethics Boards' Digital Gaps. Bloomberg Law. 29 July, 2020;.

71. The Editors (of *Journal of Vascular Surgery*). Editors' Statement Regarding "Prevalence of unprofessional social media content among young vascular surgeons". FaceBook. 2020;.

72. Thimbleby H, Cairns P. Interactive numerals. Royal Society Open Science. 2017;4(4):160903. doi:10.1098/rsos.160903.

73. Zhang Y, Masci P, Jones P, Thimbleby H. User Interface Software Errors in Medical Devices. Biomedical Instrumentation & Technology. 2019;53(3):182–194. doi:10.2345/0899-8205-53.3.182.

74. Thimbleby H. The problem isn't Excel, it's unprofessional software engineering. BMJ. 2020;371(m4181). doi:10.1136/bmj.m4181.

75. Wadman M. Could too much time between doses drive the coronavirus to outwit vaccines? Science. 2021;doi:10.1126/science.abg5655.

76. Benureau FCY, Rougier NP. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific. Frontiers in Neuroinformatics. 2017;11(69). doi:10.3389/fninf.2017.00069.

77. House of Commons Science and Technology Committee. The UK response to covid-19: Use of scientific advice; 16 December 2020. Available from: URL `committees.parliament.uk/publications/4165/documents/41300/default`.

78. Thimbleby H. Written Evidence Submitted by Harold Thimbleby to The UK response to covid-19: Use of scientific advice. C190005. House of Commons Science and Technology Committee [77]; 29 April 2020. Available from: URL `committees.parliament.uk/work/91/default/publications/written-evidence/?SearchTerm=thimbleby`.

79. Carver JC. First International Workshop on Software Engineering for Computational Science & Engineering. Computing in Science & Engineering. 2009;11(2):7–11. doi:10.1109/MCSE.2009.30.

# Supplemental Material

## Computational science: A (fixable) failure of software engineering

Harold Thimbleby, harold@thimbleby.net

February 24, 2022

1063

# 1 Further issues for Software Engineering Boards (SEBs)

## (1.a) Brief definition

Software Engineering Boards, henceforth SEBs, will be used to help and assure that critical code, including epidemic modeling, is of high standard, to provide assurance for scientific papers, Government public health and other policies, etc, that the code used is of appropriate quality for its intended uses.

Further details of the SEB proposal is in the main paper. Here we raise further issues for SEBs (additional to those covered in the main paper's introduction to SEBs), potential limitations and possible responses that can be addressed over time:

1. Until there are national qualifications, nobody — certainly nobody without professional training in software — really knows just how bad (or good) they are at software engineering.
2. When code is taken seriously, concerns may be raised on programmers' contributions to research, intellectual property rights, and co-authoring [80]. Software engineering is a hard, creative discipline, and getting epidemiological (and other scientific) models to work is generally a significant challenge, on a par with the setting up and exploring the mathematical models themselves. Often software engineers will need to explore boundary cases of models, and this typically involves hard technical mathematics [47]. Often the software engineers will be solving entirely new problems and contributing to the research. How this is handled needs exploring. How software engineers are appropriately credited and cited for their contributions also needs exploring.
3. SEBs require policies on professional issues such as membership, transparency, and accountability.
4. There should be a clear separation between the SEB members' activities as part of the Board, and their other activities, including professional advice, code development, or training (which is likely to be in demand from the same people who require formal approvals from the SEBs).
5. Professional Engineering Bodies have a central role to play in professionalism, ranging from education and accreditation to providing professional structures and policies for SEBs. For example, should and if so how should the programming skills taught to computational scientists (epidemiologists, computational biologists, economists, computational chemists, …) be accredited?
6. In the main paper, SEBs are viewed as a constructive contribution to good science, specifically helping improve the quality of epidemiological modeling. More generally, SEBs will have wider roles, for instance in overseeing software subject to medical device regulation [2].
7. SEBs may fruitfully collaborate with other engineering disciplines to share and develop best practice. For example, engineers in other

domains (e.g., civil engineers) routinely sign off projects, yet, on the other hand, they often overlook the quality of software engineering their projects implicitly rely on — for the same reasons as the scientific work discussed in this paper overlooks the dependence on quality software.

8. Clearly, at least while this paper's concepts are tested and mature, SEBs will need to collaborate closely with research organizations, journals, and funding agencies in order to develop incremental developments to policies and processes that will be most effective, and which can be introduced most productively over time to the scientific community at large. Funding agencies may wish to support such strategic work, as they have previously funded one-off projects such as [81].

There are other ideas to help make SEBs work, but it is clear they are part of the solution. We must not let perfection be the enemy of the good. SEBs don't need to be perfect on day one, but they do need to get going in some shape or form to start making their vital contribution.

## (1.b) Relationships of SEBs to Ethics Boards

1. Although SEBs may start with a checklist approach, like Ethics Boards generally do, it cannot be assumed that people approaching SEBs know enough about software engineering to perform adequate software assessments when there is any risk (as there is in public policy, medical apps, and so on). SEBs may also provide mentoring and training.

2. Unlike Ethics Boards, which provide hands-off oversight, SEBs should provide professional advice, perhaps providing training or actually helping hands-on develop appropriately reliable software. During a pandemic SEBs would be very willing to do this, but in the long run it is not sustainable as voluntary labour, so all research, particularly medical research, should include support for professional software engineering.

3. Ethics Boards typically require researchers to fill in forms and provide details, which is a feasible approach as researchers know if they are doing experiments on children, for instance, so the forms are relatively easy to fill in (if often quite tedious). On the other hand, few healthcare and medical researchers understand software and programming, so they are *not* able to fill in useful software forms on their own. SEBs need to know how well engineered the software really is, not how good its developers *think* it is. As typical programs are enormous, SEBs are either going to need resources to evaluate programs, or they will need to supervise independent bodies that can do it for them.

4. SEBs should have a two-way collaboration with Ethics Boards.

    - SEBs have to deal with ethical concerns, and how they may be implemented in code. One of the papers [129] in the survey (discussed later in this Supplemental Material) is a case in point, as is the growing cross-fertilization between AI and ethics (e.g., [82]).

- Ethics Boards also have to deal with software, and it is clear that they often fail to do this effectively. The case of the retraction of a peer reviewed articles for *The Lancet* [66, 67, 68] and the *Journal of Vascular Surgery* [69, 70, 71], discussed in the main paper, are cases in point.

5. Like some Ethics Boards, SEBs might become, or be perceived as becoming, onerous and heavy handed — as if the Board is not interested in ethics but only in following a bureaucratic pathway. It seems essential, then, that SEBs have (and perhaps are chaired by) experienced, practicing, professional software engineers to avoid this problem.

### (1.c) SEBs are necessary but not sufficient

The main paper provides evidence and argues that SEBs (or equivalent) are necessary to help improve the quality of science, specifically science relying, explicitly or implicitly, on tools or methods based in software.

SEBs address the problems identified at the laboratory end of doing science; they do not address the processes of review, editorial control, and action based on claimed results. As shown in the review of 32 papers, only some journals have code policies, and the policies are not enforced. In other words, improving the professionalization of software engineering has to proceed from doing science, which the paper covers, to the downstream issues of review and publication. SEBs may work with journals, funding agencies and even international standards agencies to improve broader awareness of professional software engineering, but this is a topic the present paper has not addressed. It needs doing.

## 2 Software engineering best practice

### (2.a) Introduction and standard references

This Supplemental Material provides more explanations and justification for following standard software engineering practices that support reliable modeling, reliable research, and, most generally, reliable science.

The reader is referred to standard textbooks for more information (e.g., [83, 84]), as well as to specialized texts that are more specifically addressed to software engineering in science (e.g., [81]).

The book *Why Programs Fail* [85] is a very good practical guide to developing better code, and will be found very accessible. Humphrey [86] outlines a thorough discipline for anyone wanting to become a good programmer. Improvement is such an important activity, Humphrey has also published a book to persuade managers of the benefits [87]. Further suggestions for background reading can be found throughout this section.

### (2.b) Essential components of best practice

Software Engineering includes the following topics, which are discussed at more length below:

**(0)** Requirements
**(1)** Formal methods
**(2)** Defensive programming
**(3)** Using dependable programming languages
**(4)** Open source and version control
**(5)** Rigorous testing
**(6)** Good documentation and record keeping
**(7)** Usability
**(8)** Reusing quality solutions
**(9)** Simplicity
**(10)** Compliance with standards
**(11)** Effective multidisciplinary teamwork
**(12)** Continuous Professional Development (CPD)
**(13)** Security and other factors

## (0) Without defining requirements, not enough skilled effort will be put into designing and implementing reliable software — or excess effort will be wasted

It is not always necessary to program well if the code to be produced is for fun, experimenting, or for demonstrations. On the other hand, if code is intended for life-critical applications, then it is worth putting more engineering effort into it. The first step of software engineering, then, is to assess the requirements, specifically the reliability requirements of the code that is going to be produced.

In practice, requirements and expectations change. Early experimental code, developed informally, may well be built on later to support models intended to inform public policy, for instance. Unfortunately, prototypes may impress project leaders who then want to rush into production software because, it seems, "it obviously works." Fortunately, best practice software engineering can be adopted at any stage, particularly by using *reverse engineering*. In reverse engineering, one carefully works out (generally partly automatically) what has already been implemented. This specification, carefully reviewed, is then used as the basis for a more rigorous software engineering process that implements a more reliable version of the system.

## (1) Without formal methods, there is no rigorous and checked specification of a program, so nobody — including its developers — will know exactly what it is supposed to do

In the physical world, to do something as simple as design and build a barbecue, you would need to use elementary mathematics to calculate how many bricks to buy. To build something more substantial, such as block of flats, you would need to use structural engineering (with certified structural engineers) to ensure the building was safe. Although programming lends itself to mathematical analysis, it is surprising that few programmers use explicit mathematics at all in the design and implementation of software.

The type and use of mathematics used in software engineering is formal methods. Not using formal methods ensures the resulting code is unsafe and unreliable. Of particular relevance to scientific modeling: there must be an explicit use of formal methods to ensure mathematical models (such as differential equations) are correctly implemented in code (and to understand the any limitations of doing so).

Formal methods require sophisticated knowledge of logic [57], as well as practical knowledge of using appropriate formal methods tools (Alloy, HOL, PVS, SPARK, and *many* others). Using the right tools is essential for reliable programming, because the tools do quickly and reliably what, done by hand, would be slow and error-prone. Standard tools cover verification, static analysis of code, version control, documentation, and so on — this paper explains why some of these activities are essential for reliable programming below.

Crucially, tools are designed to catch common human errors that we are all prone to. Many tools are designed to avoid common human errors arising *in the first place*; notably, the MISRA C toolset simply stops the developer using the most error-prone features of normal C, and hence improves the quality of programming with little effort.

Many programming languages and programming environments have integrated features that support formal methods. For example, Hoare's triples [50] (and formal thinking based on similar ideas) are readily supported by assertions, as either provided explicitly in a programming language or through a simple API. In particular, assertions readily support contracts, an important rigorous way of programming: assertions allow the program, the programming language, or tools (as the case may be) to automatically (and hence rigorously) check essential details of the program. Hoare's original 1969 paper [50] is very strongly recommended because it is a classic paper that has stood the test of time; in the 1960s it was leading research, but now it can be read as an excellent introduction, given how the field of software engineering has advanced and become more specialized and sophisticated over the decades since. Hoare is also a very good writer.

Formal methods have the huge advantage that they "think differently" and therefore help uncover design problems and bugs that can be found in no other way. Because formal methods are logical, mathematical theories (safety properties, and so forth) can be expressed and checked (often automatically); this provides a very high degree of insight into a program's details, and hence supports fault tolerance (e.g., redundancy). Ultimately, formal methods provides good reasons to believe the quality of the final code — that it does what it is supposed to do. Unfortunately, because formal methods are mathematical, few programmers have experience of using them. Fortunately tools are widely available to help use formal methods very effectively.

### (2) Without defensive programming, any errors — in data, code, hardware, or in use — will go unnoticed and be uncorrected

Defensive programming is based on a range of methods, including error checking, independent calculation (using multiple implementations written by independent programmers), assertions, regression testing, etc. Notoriously, what are often unconsciously dismissed as trivial concerns frequently lead to the hardest to diagnose errors, such as buggy handling of "well-known, trivial" things like numbers [72]. The great advantage of defensive programming is that it detects, and may be able to recover from, bugs that have been missed earlier in the development process (such as typos in the code). Defensive programming requires professional training to be used effectively, for example it is not widely known that some choices of programming language make defensive programming unnecessarily hard [88].

A special case of defensive programming appropriate for pandemic modeling is mixing methods. Do not rely on one programming method, but mix methods (e.g., different numerical methods) to use and compare multiple approaches to the modeling.

Interestingly, the only paper reviewed that claimed to do any independent testing [126] failed to include any testing in its data or code repository, so the testing itself — the essential quality assurance of the code — is not open to scrutiny (e.g., the code and the "independent" code are likely to contain common code, data, and common bugs).

## (3) Using inappropriate programming languages undermines reliability

Many popular languages are popular because they are easy to use, which is not the same as being reliable to use. The fewer constraints a language imposes, the easier it *seems* to be to program in, but the lack of constraints means the language cannot provide the checks stricter languages do. C, for instance, which is one of the languages widely used for modeling [32, 89], is not a good choice for a reliable programming language — it has many intrinsic weaknesses that are well-known to professionals, but which frequently trap inexperienced programmers. (This is not the place for a review [88] but Excel is even worse in this regard.) In particular, C is not a portable language (unless extreme care is taken), which means models will work differently on different types of computer. SPARK Ada is one example of a much more appropriate programming language to use. SPARK Ada also has the advantage that most Ada programmers are better qualified than most C programmers.

## (4) Version control and open source organizes and helps software development

It is appreciated that the models may change and be adapted as new data and insights become available. Changing models makes it even harder to ensure that they are correct, and thus emphasizes the relevance of the core message this paper: we have to find ways to make computer models more reliable, inspectable, and verifiable. Version control keeps a record of what code was used when, and enables reconstruction of earlier versions of code that has been used. Version control is supported by many tools (such as Git, Subversion, etc).

If version control is not used, one has no idea what the current program actually is. Version control is essential for *reproducibility*: [34, 76] it enables efforts to duplicate work to start with the exact version that was used in any published paper, provided that the published paper discloses the version and a URL for the relevant repository. Note that version control should also be used for data and web site data used by code, otherwise the results reported are not replicable.

If results cannot be reproduced, has anything reliable been contributed? When a modeling paper presents results from a model, it is important to reproduce those results without using the same code. Better still, research should be reproduced without sharing libraries or APIs (for example, results from a model using R might be reproduced using Mathematica —

this is a case of $N$ (where, in this case, $N = 2$) version Programming [37]). Reproducing the same results relying on the same codebase tells you little. The more independent reproductions of results the greater the evidence for belief in the implications.

Clearly, with the transformations a program from avian flu in Thailand [30] to COVID-19 in the United States and in Great Britain [29] taking place over many years, version control would have been very helpful to keep proper track of the changes. Note that professional version control repositories also provide secure off-site back up, ensuring the long-term access to the code and documentation — this would avoid loss of Supplemental Material problems, as occurred in [53].

Most version control systems would, in addition, enable open source methods so the code could be shared — and reviewed — by a wider community. Open source is not a panacea, however; it raises many trade-offs. Particularly for world-wide concerns like pandemic modeling, it increases diversity in the software developers, and fosters a diverse scientific collaboration. Open source can raise people's standards — some countries [90, 91] are using Excel models to manage COVID-19, and open source projects properly implemented would help these people enormously.

Open source raises important licensing and management questions to ensure the quality of contributions. A salutary open source case is NPM, where lawyers from a company called Kik triggered Azer Koçulu, that is, a *single* programmer, to remove all his code from a repository. This caused problems to many thousands of JavaScript programmers worldwide who could no longer compile anything — ironically, including Kik itself [92].

Critically in the case of epidemic modeling, open source democratizes the model development and interpretation, and enables properly-informed public debate. Note that many (if not most) successful open source projects have had a closed team of highly dedicated and directly employed developers [55].

## (5) Without professional testing, there is no acceptable evidence that a program works under real conditions

In poorly-run software development it is very easy to miss bugs, because the flawed thinking that inserted bugs in the code is going to be the same flawed thinking with the same misconceptions that tries to detect them. Rigorous testing includes methods like fault injection. Here, the idea is that if testing finds no bugs, that may be because the testing is not rigorous enough rather than that the program actually has no bugs. Fault injection inserts random bugs, and then testing gives statistical insights into the number of bugs in a program (depending on how many deliberate bugs it successfully finds).

It is very tempting to test code while it is being built, save some or all of the code on a repository, but forget to check that the code has not changed out of recognition of the earlier tests — tests should be saved so that modified code can easily be tested again. For example, if a test reveals a bug, the bug should be fixed *and* the test needs to be re-run to check the fix worked (and did not introduce other bugs previously eliminated).

It is important that code is saved and then downloaded to a clean site, confirmed it is consistent, and a new build made (preferably by an independent tester), which is then re-tested. If this procedure (or equivalent) is not followed, there is no assurance that the code made available with the paper is complete and works reliably.

There are many other important testing methods [83, 84, 37].

## (6) Without documentation and record keeping, nobody — least of all the programmer — knows what code is supposed to do or how to get it to do it

Documentation covers internal documentation (how code works), developer (how to include it in other programs), configuration (how to configure and compile the code in different environments), external documentation (how the code is used), and help (documentation available while using the program).

For critical projects, such as for pandemic modeling, all documentation (including software) should be formally controlled, typically digitally signed and backed up in secure repositories. One would also expect a structured assurance case to be made, both to help the authors understand and complete their own reasoning and to help reviewers scrutinize it [8].

For purely scientific purposes, perhaps the most important form of internal is internal documentation: how to understand how and why the code works. This is different from developer documentation, which is how to *use* the code in other programs. For example, code for solving a differential equation needs explaining — what method does it use, what assumptions does it have? In contrast, the developer documentation for differentiation would say things like it solves ordinary differential equations with parameters $e$ for the function $f$ with the independent variable $x$ in the interval $[u, v]$, or whatever, but *how* it solves equations is of little interest to the developer who just needs to use it. How code works — internal documentation — is essential for the epidemiologist, or more generally any scientist. An example of a simple SIR epidemiological model's internal documentation can be found at URL `http://www.harold.thimbleby.net/sir`

There are many tools to help manage documentation (Javadoc, Doxygen, …). Literate programming is one very effective way of documenting code, and has been used for very large programming projects [93]. Literate programming has also been used directly to help publish clearer and more rigorous papers based on code [42] — a paper that also includes a wider review of the issues.

Documentation should be supplemented by details of algorithms and proofs of correctness (or references to appropriate literature). All the documentation needs to be available to enable others to correctly download, install and correctly use a program — and to enable them, should they wish, to repurpose it reliably for their own work. In addition, documentation requires specifications and, in turn, *their* documentation.

A important role of documentation is to cover configuration: how to get code to work — without configuration, code is generally useless. The most basic is a README file, which explains how to get going; more useful approaches to configuration include make files, which are programs that do the configuration automatically.

Without proper record keeping, code becomes almost impossible to maintain if programmers leave the project. Note that computer tools can make record keeping, laboratory books etc, trivial — if they are used.

## (7) If code is not usable, even if it is "correct" it will not be used and interpreted correctly

Usability is an important consideration: [94, 95] is the program usable by its intended users so they can obtain correct results? Often the programmers developing code know it so well they misjudge how easy it will be for anyone else to use it — this is a very serious problem for the lone programmer (possibly working in another country) supporting a research team. Usability is especially important when programs are to be used by other researchers and by non-programmers, including epidemiologists.

In publishing science, an important class of user includes the scientists and others who will use or replicate the work described. When code used in research is non-trivial, it is essential that the process of successfully downloading code and configuring it to run is made as usable as possible. Typically so-called makefiles are provided, which are shell scripts or apps that run on the target machine, establish its hardware and other features, then automatically configure and compile the code to work on that machine. Makefiles typically also provide demo and test runs and other helpful features. Other approaches to improve usability are zip files, so every relevant file can be conveniently downloaded in one step, and using standard repositories, such as GitHub which allow new forks to be made, and so on.

## (8) Without using existing solutions (libraries, APIs, etc) reinventing code merely reinvents bugs

Reusing quality code (mathematical functions, database operations, user interface features, connectivity, etc) avoids having to develop it oneself, saves time and avoids the risks of introducing new bugs. The more code that is reused, the more likely many people will have contributed to improving it — for example, reusing a standard database package will provide Atomicity, Consistency, Isolation, and Durability (so-called ACID properties) without any further work (nor even needing to understanding what useful guarantees these basic properties ensure).

Note that reusing code assumes the originators of the code followed good software engineering practice — particularly including good documentation; equally, if the code being developed building on it follows good software engineering practice, it too can be shared and further improved as it gets more exposure. Its quality improves through having scrutiny by the wider community, and in successful cases, leading to consensus on the best methods. Indeed, reuse, scrutiny, and consensus are the foundations of good science.

Anticipating reuse during program development is called *flexibility*, where various programming techniques can greatly enhance the ease and reliability of reuse [96].

A special case of reuse is to use software tools to help with software development. The tools (if appropriately chosen) have been carefully developed and widely tested. Tools enable software developers to avoid or solve complex programming problems (including maintenance) repeatedly and with ease.

## (9) Poor programmers often fix bugs rather than the causes of bugs: complexity and obfuscation

When a program doesn't quite do what is wanted, it is tempting to add more features or variables, or to treat the problem as an "exception" and program around it — which inserts more code and, almost certainly, more bugs. This way lies over-fitting, a problem familiar from statistics (and machine learning). Programs can be made over-complex and they can then do anything; an over-complex program may seem correct by accident. Instead, the hallmarks of good science are that of parsimony and simplicity; if a simple program can do what is needed it is more likely to be correct. A simpler program is easier to prove correct, easier to program, and easier to debug. A special case of needing simplicity is when fixing bugs: instead of fixing bugs one at a time, one should be fixing the *reasons* why the bugs have happened. Generally, when bugs are fixed, programmers should determine *why* the bugs occurred, and thence repair the program more strategically.

## (10) International standards have been developed to support critical software development

To ensure adherence to best practice and, importantly, to avoid being unaware of relevant methodologies, professional software development projects adopt and adhere to relevant standards, such as ISO/IEC/IEEE 90003:2018 [97]. However, for safety-critical models or models of national policy significance, much stronger standards such as aviation software standards, such as RTCA DO-178C/EUROCAE ED-12C [98], commonly called DO-178C, will be more appropriate. Publications should then cite the standards to which their computer models comply.

Note that medical device regulation, which has its own standards, is lagging behind professional software engineering practice, and currently provides no useful guidance for critical software development [2].

## (11) Effective multidisciplinary teamwork is essential because no individual has the capacity to develop non-trivial reliable software

As this long list illustrates, Software Engineering is a complex and wide-ranging subject. Software engineering cannot be done effectively by individuals working alone (for instance, code review is impossible for individuals to perform effectively), even without considering the complexities of the domain the code is intended for (in the present case, including pandemic modeling, mathematical modeling, public health policy, etc). Multidisciplinary teamwork is essential.

Modern software is complex, and no one person can have the skills to understand all relevant aspects of all but the most trivial of programs. Furthermore, programming is a cognitively demanding task, and causes loss of situational awareness (that is, cognitive "overload" making one unable to track requirements beyond those thought to be directly related to the specific task in hand). The main solution to both problems is teamwork, to bring fresh insights, different mindsets and skills to the task.

Peer review of code is an essential teamwork practice in reliable program development: [99, 84] it is easy to make programming mistakes that one is unaware of, and an independent peer review process is required to help identify such unnoticed errors.

Almost all software will be used by other people, and user interface design is the field concerned with developing usable and effective software. A fundamental component of user interface design is working with users and user testing: without engaging users, developers are very likely to introduce quirks that make systems less usable (often less safe) than they should be. In short, users have to be brought into the software team too.

## (12) Computing technologies are advancing rapidly, and best practice in software engineering is continually evolving

As computing technology continues to develop rapidly — especially as new programming tools and systems are introduced — best practice in software engineering is also rapidly evolving. Continuous Professional Development (CPD) is essential.

Ironically, the more organized CPD the more likely the content itself will lag behind. There is an argument for two-way links between universities (and other research organizations), research science developers, including enabling developers to undertake part-time research degrees. Research degrees teach not just current best-practice but also how to stay abreast of the relevant technologies and literature as it develops.

The UK's Software Sustainability Institute is one initiative that is making important contributions [100, 101], and its web site will no doubt remain timely and up to date in a way that this paper cannot.

Note that CPD is not just a matter of learning current best practice, but a continual process as best practice itself continually evolves. In software engineering, a current (as of 2021) initiative concerns reproducible code artifacts and badging papers to clearly show the approaches they take [59], and this will in due course have a direct impact on software engineering standards in other fields.

## (13) Other factors ...

Of course, there are many other factors to be considered for the professional development of critical code, such as using appropriate methods to ensure cybersecurity [102, 103], particularly while also being able to up- and download secure updates.

For pandemic modeling specifically, understanding the limitations of numerical methods (in particular, how numerical methods are affected by the choice of programming language and style of programming) is critical.[1] Hamming [47] is considered a classic, but there is a huge choice available.

For reasons of space, the present paper does not discuss the issues raised by AI, nor the many very important, non-trivial social and professional concerns, which have complex implications for software engineering practice, such as managing programming teams, data ethics, privacy, legal liability [104], or software as a matter in law, as in disputes over model results or disputes over ownership of code [105].

[1]For example [135] was noticed to use literal numbers at too high a precision for the chosen language, where conformant implementations use IEEE 754 double precision 64-bit floating point. Such an error typically has an undefined impact on results, and unfortunately is easy to overlook as the program almost certainly ignores the error when running.

## 3 Code, data and publication

All computer systems are in principle equivalent to Turing Machines, and Turing Machines make no distinction between program and data. It is possible to define Turing Machines that do separate program code and data, but as soon as a Universal Turing Machine is constructed, its data *is* code. Indeed, Universal Turing Machines are a theoretical abstraction of virtual machines, which are used widely in practical computing. Java, for instance, runs in a virtual machine, so any Java program code (and any data it uses) is in fact merely *all* data to the Java virtual machine. At another extreme, $\lambda$-calculus is purely program source code, yet $\lambda$-calculus is equivalent to Turing Machine computation. Therefore, even the "pure" programs of $\lambda$-calculus also represent data.

These elementary theoretical considerations underly an important practical fact: there is no fundamental difference between code and data, and no distinction that is relevant for scientific publication purposes.

There is no code/data distinction one can imagine that cannot easily, even accidentally, be circumvented. In other words, a journal's data policies and code policies should be the identical — and the conventionally stricter data policies should also apply to code. It is baffling that some journals have data policies that are weaker than their data policies; it is certainly indefensible to have no code policies at all.

Significant cyber-vulnerabilities result from there being no difference between code and data. For example: an email arrives, which brings data to a user. The user opens an attachment, perhaps a word processor text document, which is more data. The word processor runs macros in the text document — but now it is code. The macros move data onto the user's disc. The data there then runs as code, and corrupts the user's data across the disc — which includes both data and code stored in files. And so on. Each step of a computer virus infection crosses over non-existent "boundaries" between data and code [106].

This section's discussion may sound like arcane and irrelevant pedantry, but these issues are at the very foundations of Computer Science.[2] If we ignore or misunderstand these basic things — or overlook them in policies and procedures — bugs and irreproducibility are the inevitable (and confusing) consequence.

The main paper points out that data is often embedded in code using "magic numbers." Let's now explain how.

A simple fragment of program code might say

```
x = 324+sin(theta*pi/180);
```

This is clearly all source code, but the number 324 above is likely to be some sort of relevant data, though it might be a physical constant whose value does not depend at all on *this* experiment. The next hard-coded value mentioned in the calculation is difficult to categorize: is the value of $\pi$ empirical data or is it part of a standard formula? Some programming languages like Mathematica treat $\pi$ as an exact mathematical constant

[2]Many of the foundational issues were explored thoroughly by Christopher Strachey and others in the 1960s; Strachey's classic lectures are reprinted in an accessible 2000 publication [107]. Being originally a very old paper this classic introduction is much easier to read than many more recent discussions of the foundations of Computer Science.

(e.g., Mathematica calculates $\tan \pi/4 = 1$ exactly), but $\pi$ is *also* definitely an inexact empirical value.[3]

The point is, the distinctions between data, program and even mathematical constants are purely a matter of perspective.

Unfortunately, there is data that is extremely easy to overlook (and therefore very hard to manage) because it is embedded in arbitrary ways in code. You may assume that the function `sin`, as used in the calculation example above, is the standard trigonometric function for calculating sines (and because of the $\pi$ in the expression, you assume `theta` is degrees and `sin` is taking radians as its parameter type) but almost all programming languages allow `sin` to be any function whatsoever. Confusingly, even if it calculates sines, it is generally a different function when the code is run on a different computer producing numbers that are not exactly the same.

It is impossible to tell.

## (3.a) When magic numbers become magic code

Data often controls the flow of code. For example, data summarizing patients may include their gender, but the program processes males and females differently. Then data becomes code.

Arbitrary numbers appearing in code are obviously magic numbers, but code often conceals the magic numbers of data by "programming them away" during the coding process.

For example, the magic number 324 was explicit in the line of code shown above, but if somewhere else the program says

```
if evenQ(324) then A; else B;
```

many programmers would optimize this to `A`, because they know the condition is true because of their assumptions. This now seems to be a more efficient program because it has avoided a test (which a modern complier would have optimized away anyway). Unfortunately, the previously explicit dependency of the code on the magic number 324 has completely disappeared.

Obviously this example seems trivial, but it illustrates that programmers do some of their programming while writing code, and many assumptions disappear completely and have no representation in the final code. More complex code will have many facts hard wired into the code — so in fact the code contains data. Code can even read in formulas from data and compile them to perform further calculations, and so on.

This is one reason bugs — effectively incorrect assumptions — are so hard to find, because they have no concrete form in the final program.

## (3.b) When data is code

Many computer programs blur the simplistic code/data distinctions deliberately, to create virtual machines. Data is then run on the virtual machine as program. Many programs provide standard features to do this, such as LISP's and JavaScript's `eval` functions. Henderson's book [108] builds an elegant Pascal program to run *any* LISP program as data, and then shows that the LISP program can run itself running other programs, so it is now

[3] A record set on 19 August 2021, the most accurate value of $\pi$ then known was 62,831,853,071,796 digits URL `www.fhgr.ch/en/specialist-areas/applied-future-technologies/davis-centre/pi-challenge`

its own code and *its* data — despite being purely data to the Pascal program. There are numerous advantages to doing this, including: the Pascal program is not just reading data, but structured data that must conform to the rules of LISP; the LISP running itself runs faster than the original Pascal running LISP, even though the Pascal virtual machine is still doing it in the recursive case; LISP is a much more powerful language than Pascal, so a virtual machine can be used to escape the barriers of a limited implementation such as Pascal. In short, any distinctions between code and data are impossible to maintain.

AI and Machine Learning are further examples of exploiting data as code. Typically a program learns from a training set of data, and then processes future data differently depending on what it has learned. In other words, the original data becomes a model which is now code.

## (3.c) Exploiting code as data for more reliable science

In the present paper, we knowingly built on this blur between data and code. However, what we did was not unusual except in our explicit and rigorous approach to managing and summarizing data reliably in the paper.

The paper and its Supplemental Material are typeset in LaTeX, a popular typesetting language. LaTeX not only has text (as you are reading right now) but it also has code. For example, "LaTeX" was typeset by running the code for a macro called `\LaTeX`, which then calculated how to position the letters as they are wanted. When $\pi$ was written above, the code that generated what you read actually said `$\pi$` — so is this data that just says $\pi$ or is it code that tells the computer to change character sets from Latin to Greek, and then uses `\pi` as a program variable name to select a particular glyph from the data about typesetting Greek characters? The distinctions are all a bit moot. In other words, the publication itself is data to a LaTeX program, and within that data it includes further programs. Indeed, LaTeX is run on a virtual machine, in exactly the same way that Henderson's LISP is, and doing so provides the same advantages.

The data for this paper's survey was itself originally written as literal text in LaTeX: it meant that LaTeX could process it to produce a typeset table (as in the Supplemental Material above). As the extent of the data grew, it rapidly became apparent that LaTeX is a poor choice to manage structured data. A simple JavaScript program was written to convert the LaTeX data into JSON (which is much more readable than LaTeX) and also generate CSV files that can be processed in standard office software such as Excel, which some readers may prefer. In fact, examining and comparing the same data in the contrasting formats, this typeset file, in JSON, and in Excel (reading the generated CSV) provided multiple different perspectives of the data that increased redundancy and confidence that the data was correct and correctly handled.

It is important to note that using such techniques is quite routine in science publication, though often pre-existing tools are used to streamline the process (and to ensure that it is more widely understood). The paper [124], for example, in addition to using a typesetting system for publication, also placed its code in a repository using R Markdown [109], a programming environment based on R designed for generating and documenting lab books — almost the polar opposite of LaTeX, which is designed for publication but can be used for programming.

Finally note that what may look like magic numbers used throughout the present paper (such as the 32, as in “32 papers were evaluated”) are all in fact named, calculated and placed *in situ* directly from computations performed on the JSON paper’s data.

## 4 The Speigelhalter trustworhiness questions

David Speigelhalter is concerned how statistics is often misused and misunderstood. In his *The Art of Statistics* [4] Speigelhalter brings together his advice for making reliable statistical claims: they need to be accessible, intelligible, assessable, and usable — the claims need to be properly accountable. Speigelhalter proposes ten questions to ask when confronted with any claim based on statistical evidence. Some of his questions are quite general, and might be applied to any sort of scientific claims, but all have analogous questions that could be addressed to software code or papers relying on code — analogues are suggested in **bold** below.

What might seem like dauntingly technical software issues are no more demanding than the basic statistical issues that are regularly acceded to; failing to ask them is as risky as dismissing statistical scrutiny.

### (4.a) How trustworthy are the numbers?

1. *How rigorously has the study been done?* For example, check for ‘internal validity,’ appropriate design and wording of questions, pre-registration of the protocol, take a representative sample, using randomization, and making a fair comparison with a control group.

   ▶ **How rigorously has the software engineering been done? Section 2 in the Supplemental Material provides a list of important issues that must be addressed for any reliable software.**

   ▶ **“Internal validity” assumes that there is evidence the programmers had uncertainty in the code’s reliability and checked it. Were different methods used and compared, or was all confidence put into a single implementation? What internal consistency checks does the implementation have? Were invariants and assertions defined and checked?**

2. *What is the statistical uncertainty/confidence in the findings?* Check margins of error, confidence intervals, statistical significance, multiple comparisons, systemic bias.

   ▶ **How are the claims presented that give us confidence in the code that they are based on? Are there discussions of invariants, independent checks for errors, and so on? Again, Supplemental Material section 2 provides further discussion of such issues.**

3. *Is the summary appropriate?* Check appropriate use of averages, variability, relative and absolute risks.

   ▶ **If the claims are exploratory, weaker standards of coding can be used; if the claims are a basis for critical decisions, then there should be evidence of using**

**appropriate software engineering (such as defensive programming) to provide appropriate confidence in the results claimed.**

## (4.b) How trustworthy is the source?

4. *How reliable is the source of the story?* Consider the possibility of a biased source with conflicts of interest, and check publication is independently peer-reviewed. Ask yourself, 'Why does this source want me to hear this story?'

    ▶ **The source of many science stories is the output of running some code. How reliable is this code? What evidence is there that the code was well-engineered so its reliability can be trusted?**

    ▶ **What evidence is there of rigorous (e.g., code review and tool-based) independent methods being used to manage coding bias?**

5. *Is the story being spun?* Be aware of the use of framing, emotional appeal through quoting anecdotes about extreme cases, misleading graphs, exaggerated headlines, big-sounding numbers.

    ▶ **Be wary of AI and ML which may have been trained by chance or specifically (if not deliberately) to get the results described.**

6. *What am I not being told?* This is perhaps the most important question of all. Think about cherry-picked results, missing information that would conflict with the story, and lack of independent comment.

    ▶ **Cherry picking with code is often unconscious and is very common: when running code produces the "cherries" for a paper it is tempting to stop testing the code and just assume it is running correctly. So, what evidence is there that the code was rigorously developed and cherry picking avoided?**

## (4.c) How trustworthy is the interpretation?

7. *How does the claim fit with what else is known?* Consider the context, appropriate comparators, including historical data, and what other studies have shown, ideally in a meta-analysis.

    ▶ **Is there any discussion of the code and how does it compare with other peer-reviewed publications using code used for similar purposes?**

8. *What's the claimed explanation for what has been seen?* Vital issues are correlation v. causation, regression to the mean, inappropriate claim that a non-significant result means 'no effect,' confounding attribution, prosecutor's fallacy.

- ▶ **These are all good statistical questions. The software engineering analogy is: are the claims backed up by a sufficiently detailed discussion of the algorithms and software engineering that justify the appropriateness of the chosen software implementation? The Supplemental Material list in section 2 provides examples of expected explanations for the trustworthiness of running some code.**

9. *How relevant to the story is the audience?* Think about generalizability, whether the people being studied are special case, has there been an extrapolation from mice to people.

   - ▶ **Generalizability is equivalent to is the code available, easy to understand and use for more general purposes — including further work and checking the reproducibility of the claims being made?**

10. *Is the claimed effect important?* Check whether the magnitude of the effect is practically significant, and be especially wary of claims of 'increased risk.'

## Additional references for Supplemental Material

References numbered 1–79 appear in the reference list in the main paper.

[80] Vancouver Group. Uniform requirements for manuscripts submitted to biomedical journals. JAMA. 1997;277(11):927–934. doi:10.1001/jama.1997.03540350077040.

[81] Stepney S, Polack FAC, Alden K, Andrews PS, Bown JL, Droop A, et al. Engineering Simulations as Scientific Instruments: A Pattern Language. Springer; 2018.

[82] Misselhorn C. Artificial Morality. Concepts, Issues and Challenges. Social Science and Public Policy. 2018;55:161–169. doi:10.1007/s12115-018-0229-y.

[83] Sommerville I. Software Engineering. 10th ed. Pearson; 2015.

[84] Knight J. Fundamentals of Dependable Computing for Software Engineers. CRC Press; 2012.

[85] Zeller A. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann; 2006.

[86] Humphrey WS. PSP: A Self-Improvement Process for Software Engineers. Addison Wesley; 2005.

[87] Humphrey WS. Winning with Software: An Executive Strategy. Addison-Wesley Professional; 2001.

[88] Thimbleby H. Heedless Programming: Ignoring Detectable Error is a Widespread Hazard. Software — Practice & Experience. 2012;42(11):1393–1407. doi:10.1002/spe.1141.

[89] Chao DL, Halloran ME, Obenchain VJ, Longini Jr IM. FluTE, a Publicly Available Stochastic Influenza Epidemic Simulation Model. PLOS Computational Biology. 2010;6(1):e1000656. doi:10.1371/journal.pcbi.1000656.

[90] Abir M, Nelson C, Chan EW, Al-Ibrahim H, Cutter C, Patel K, et al. RAND Critical Care Surge Response Tool: An Excel-Based

Model for Helping Hospitals Respond to the COVID-19 Crisis. RAND Corporation; 2020. Available from: URL `www.rand.org/pubs/tools/TLA164-1.html`.

[91] Alvarez NM, Gonzalez-Gonzalez E, Trujillo-de Santiago G. Modeling COVID-19 epidemics in an Excel spreadsheet: Democratizing the access to first-hand accurate predictions of epidemic outbreaks. MedRxiv. 2020;doi:10.1101/2020.03.23.20041590.

[92] Schlueter IZ. Blog: kik, left-pad, and npm. NPM Blog; 23 March 2016. Available from: URL `blog.npmjs.org/post/141577284765/kik-left-pad-and-npm`.

[93] Knuth DE. Literate programming. vol. 27. Center for the Study of Language and Information Publication Lecture Notes; 1992.

[94] Shneiderman B, Plaisant C, Cohen M, *et al.* Designing the User Interface: Strategies for Effective Human-Computer Interaction. 6th ed. Pearson; 2016. Available from: URL `www.cs.umd.edu/hcil/DTUI6`.

[95] Thimbleby H. Press On: Principles of Interaction Programming. MIT Press; 2007.

[96] Hanson C, Sussman GJ. Software design for flexibility: How to avoid programming yourself into a corner. MIT Press; 2021.

[97] ISO/IEC JTC 1/SC 7 Software and systems engineering Committees. Software engineering — Guidelines for the application of ISO 9001:2015 to computer software. International Organization for Standardization (ISO); 2015. Available from: URL `www.iso.org/standard/74348.html`.

[98] RTCA Committee SC-205. DO-178C — Software Considerations in Airborne Systems and Equipment Certification. RTCA; 2011. Available from: URL `my.rtca.org/NC__Product?id=a1B36000001IcmqEAC`.

[99] Baum T, Leßmann H, Schneider K. The Choice of Code Review Process: A Survey on the State of the Practice. In: Lecture Notes in Computer Science. vol. 10611 of Product-Focused Software Process Improvement: 18th International Conference; 2017. p. 111–127.

[100] Brett A, Croucher M, Haines R, Hettrick S, Hetherington J, Stillwell M, et al. State of the Nation Report for Research Software Engineers. Research Software Engineer Network; 2017. Available from: URL `zenodo.org/record/495360#.Xyfuyi2ZOCM`.

[101] Software Sustainability Institute. Web site; 2020. Available from: URL `software.ac.uk`.

[102] Anderson R. Security Engineering. 3rd ed. Wiley; 2020.

[103] Shostack A, Zurko ME. Secure Development Tools and Techniques Need More Research That Will Increase Their Impact and Effectiveness in Practice. Communications of the ACM. 2020;63(5):39–41. doi:10.1145/3386908.

[104] Schneier B. Click Here To Kill Everybody — Security and Survival in a Hyper-connected World. W. W. Norton & Company, Inc; 2018.

[105] Mason S, Seng D. Electronic Evidence. 4th ed. Humanities Digital Library; 2017.

[106] Thimbleby H, Anderson SO, Cairns P. A Framework for Modelling Trojans and Computer Virus Infection. Computer Journal. 1999;41(7):444–458. doi:10.1093/comjnl/41.7.444.

[107] Strachey C. Fundamental Concepts in Programming Languages. Higher-Order and Symbolic Computation. 2000;13:11–49. doi:10.1023/A:1010000313106.
[108] Henderson P. Functional Programming: Application and Implementation. Computer Science. Prentice-Hall International; 1980.
[109] Xie Y, Allaire JJ, Grolemund G. R Markdown: The Definitive Guide. Chapman & Hall/CRC; 2020.
[110] Fuster R. The `calculator` and `calculus` packages: Arithmetic and functional calculations inside LaTeX. TUGboat. 2012;33(3):265–271.
[111] Hurst P. Data sharing and mining. Royal Society. 2022;doi:10.25504/FAIRsharing.dIDAzV.

# 5 Summary of pilot survey

## (5.a) Assessment criteria and methods

A survey sampled of recent papers that were published online in July 2020, accepted for publication after peer review in 3 high-profile, highly competitive leading peer-reviewed journals, namely *Lancet Digital Health* ($N = 6$), *Nature Digital Medicine* ($N = 12$) and *Royal Society Open Science* ($N = 14$). Papers were selected from the journals' July 2020 new online listings where the paper's title implied that code had been used in the research. Commentary, correspondence and editorials were excluded. The sample represents what the editorial and the broader peer review community considers to be good practice.

The selection process will have certainly missed some papers that use code, but the criterion selects papers where the wording of the title indicates that the authors consider code to be a component of the scientific contribution. Indeed, all sampled papers used code in their research. Although there is unavoidable subjectivity in the paper evaluations and uncontrolled bias from using a single evaluator (the author of this paper), it is hoped that using a sample of 32 papers from 3 diverse journals is sufficient to randomize errors so that they largely cancel out, and the overall trends as discussed in this paper are reliable. It should be noted that, except where a paper provides a URL to a code repository, much code was disorganized so possibly not all code was reviewed because it was too hard to find (some emails to authors have not been responded to).

Since almost every scientific paper relies on generic computer code (calculating statistics, plotting graphs, storing and manipulating data, accessing internet resources, etc), the baseline of papers using code was not assessed. Papers whose title indicated their contribution included or relied on bespoke code were selected, and all those clearly relied heavily on their own specifically developed code. Papers that may have relied on bespoke code but whose titles made no such implication were not assessed.

Although the pilot survey is not a systematic review, following Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) [15] it is good practice to disclose details of the reviewers. In the present case, the selected papers were assessed by the author of the present paper. The study was not blinded. This is, of course, a limitation of the study. However, the reviewer is a full professor of computer science, who has taught and assessed computer software since the 1970s, using moderated and peer-reviewed processes for undergraduate and postgraduate computing degrees, and is well aware assessing code quality has been a lively topic in software engineering for decades (there is now international standard ISO/IEC 9126, updated to ISO 25000). The author has written legal documents analyzing software for criminal cases involving faulty software. The author has approximately 528 published papers in computer science.

The evaluations performed for the present paper are at a trivial level where sources of bias should have a negligible effect, particularly given that the overall conclusion is consistent across both the diverse sample and the computational science-based papers cited in the paper that did not form part of the selected sample. As said in the body the paper: *the fact that the specifically identified problems are elementary errors in software engineering (see the discussion in* [main paper] *section 4) suggests more sophisticated analysis is not required.*

In any case, as stated in the main paper, the full dataset and analysis code is available at

URL `github.com/haroldthimbleby/Software-Enginering-Boards`

and the reviewed papers are (unless retracted) still available online for independent assessment.

There is considerable debate over what good commenting practice is, but this is because comments have many roles — from helping students to get marks in assessments, asserting intellectual rights, reminding the developer of things to do, managing version control, to explaining the code to third parties. Different programming languages also develop "cultures" and tool-based systems that encourage different approaches for comments (examples include R Markdown, Mathematica Notebooks, JavaDoc, Haskell's Haddock, and so on). For scientific code, however, the explanatory role is critical, and this is what was assessed in the present survey. It is notable that no such tool-based approach to code or documentation was used in any code reviewed.

The completeness or executability of code was not assessed, although if code was obviously incomplete this was noted. Whether code runs as claimed is a matter of research integrity, which is beyond the scope of this survey. What is relevant to the study is whether the code is described in sufficient detail that the methods used can be scrutinized. Obviously being able to run the code will help, but clarity in documentation and comments is critical. It is more like "can we see the critical pages from your lab book so we understand what you did?" rather than "can we have a free run of your laboratory, even though we don't understand the details of the science?"

As an informal survey, intended to establish whether the issues in epidemic modeling were more widespread, and given the very poor level of documentation found in scientific code, it was not felt necessary to have independent or blind assessment.

The data was recorded in JSON (JavaScript Object Notation), which is a simple standard data format. A typical entry in the data file looks like this (with long field values truncated for clarity):

```
{
  accessed: "14 July 2020",
  doubleChecked: "17 January 2021",
  authors: "Callahan A, Steinberg E, Fries JA, Gomba …,
  year: 2020,
  title: "Estimating the efficacy of symptom-based …,
  volume: 3,
  number: 95,
  journal: "Nature Digital Medicine",
  doi: "10.1038/s41746-020-0300-0",
  dataComment: "On request",
  hasCodeInPrinciple: 1,
  codeComment: "``Code is available upon request from th …,
  pages: 3
}
```

The data was entered by hand (as JSON terms), after reading and reviewing each paper in the survey. In total there are 32 data fields available for documenting papers, but not all need be used for each paper; for example, the field `hasCodeTested` defaults to `false`, so it need not be set — it is also an error to set it if another field asserts there is no code to evaluate! (A separate JSON data structure maps the data fields to English descriptions, along with default values if they are optional descriptors.)

A JavaScript program sanity checks the JSON data. The sanity checks found a few errors (e.g., it checks that if there are comments of any sort then there must be some accessible code; it checks the DOI is accessible, etc), which led to a productive double-checking of all the facts of the original papers — and correcting all the errors. Some papers that had had no code available during the first assessment had uploaded code by the time of the double-checking.[4] A field `doubleChecked` was added to supplement the original data field `accessed` to track the process of double-checking the data; the sanity checks then of course checked all `doubleChecked` fields were completed.

Note that since JSON data is JavaScript code, it was convenient to combine the data, the data sanity checking, and the analysis all in a single file. Hence, running the data generates the core human-readable information used in this paper.

The JavaScript data+program generates files from the JSON, with all the definitions; these files were then included in both the main paper and in this Supplemental Material, so when the paper or Supplemental Material is typeset all tables and specific data items are typeset automatically, consistently and reliably by LaTeX.[5] For example, the register `\dataN` is set to the value 32, which is the total number of papers assessed in the JSON data, and the macro `\journalBreakdown` is defined directly from the data to be the following text (when typeset in LaTeX):

> *Lancet Digital Health* ($N = 6$), *Nature Digital Medicine* ($N = 12$) and *Royal Society Open Science* ($N = 14$)

— which is the breakdown of the total $N = 32$ by journal name. The *exact* same text was also used in the main paper.

An interesting consequence of this automatic approach is that as the author found themselves starting to write text such as:

[4] Note that double-checking was performed by the same person as the first assessment, though with the benefit of a six month gap to bring a degree of independence.

[5] Because of its approach to automatic typesetting, this journal requires that no files are explicitly `input` into LaTeX, so a simple JavaScript program is used to recursively expand all `input` commands before submission. If the source files are available for download after this paper is accepted, they will therefore contain no `input` commands, but they will contain comments at the appropriate points in the expanded source code explaining the sources of the `input` data.

```
Code repositories were used by 10 papers …
```

it motivated extending the JavaScript data processing so that *all* specific quantities mentioned in the paper are traceable directly back to the JSON data. The phrase above is now in fact written in LaTeX in the paper as follows:

```
Code repositories were used by
\plural{\countUsesVersionControlRepository}{paper} …
```

where `\plural` automatically writes a word ("paper" in this case) in singular or plural form as required.

For each of the 22 variables used in the paper, JavaScript generates a LaTeX header file that declares and assigns a calculated value. The header file is then included in the paper using LaTeX's standard `\input` command. Here is an example of one such automatic definition:

```
\newcount \dataVariableCount
\dataVariableCount = 22
```

so the named value is then available for the author to use when the paper is typeset. The tables generated for the present paper also include data that did not require names for explicit use elsewhere in the LaTeX files, so these numbers are not counted in the specific 22 variable count.

Some of the files generated from the JSON data are Unix shell scripts. For example, details of all the papers with GitHub repositories are automatically collected into a shell script so the repositories can be cloned locally and then measured (using `awk` scripts), e.g., to generate table 3, as used in the main paper.

The full JavaScript JSON data and processing code (including the makefile) is provided in this paper's repository, as described in the main paper.

## (5.b) Detecting and defending against error

Normally, when we write a number like 10 in a paper, especially longer or more complex numbers, we will later proof read them as "the numbers we intended to write" — as remembering what we meant is easier than reading the details. Unfortunately, a sentence would likely seem to make as much sense when a number has been erroneously typed as, say, 1.0, 9, 11, or 100 — we hardly bother to pay attention because we think we know what we are reading; at least we know what we meant to write. Worse, the more often we proof read a document, the more we remember, so the better we know what we think we said, and the more casual our proof reading becomes. It is very hard to spot all of our own typos.

- The first and last errors above are examples of the very common error of "out by ten" (common partly because the correct number, 10 looks very similar to 1.0, and 10.0 also looks very similar to 100) [2].

- The middle two errors above are examples of the common error of "out by one," or "fence post errors" frequently made by mixing up counting fences or the posts (there is usually one more post than fence panel) [2].

All the discussion and examples above were generated automatically, and have been checked correct for other correct values than 10. This approach, too, considerably helps defend against common Human Factors errors. For example, if we set `\countUsesVersionControlRepository`=10 to be 2348, say, then all of the subsequent sentences that mention it will say something unexpected and so have to be more carefully proof-read, significantly reducing confirmation bias. The approach turns a possibly-hard-to-spot *single* error into *multiple* errors spread throughout the paper into different contexts, thus increasing the chances of noticing the error.

It must be emphasized that an automatically-guaranteed number that is supposed to be the same appearing in multiple different contexts is an extremely effective way of defending against common Human Factors errors. As the number is proof read, the different contexts encourage it to be read more carefully, and in different ways.

If any of the numbers used in a paper were safety critical (e.g., lives directly depend on their values) then further checks would have been made to help detect and avoid errors. LaTeX itself makes it very easy to check that numbers fall within reasonable ranges, or to have any other required safety properties. For the present paper, a potential problem is if the paper is mistakenly typeset *before* the latest JSON data has been analyzed; in which case, none of the variables, like `\countUsesVersionControlRepository`, will have been correctly set and their values could be undefined or nonsense (e.g., from a debugging run of `data.js`). Variables might be checked as follows:.

**This is *automatic* confirmation that**

$$5 \leq \texttt{countUsesVersionControlRepository} \leq 20$$

**and therefore falls within pre-defined sanity limits set for this paper.**

The corresponding error messages would not normally be printed in a paper like this — they would normally be reported by stopping a LaTeX run, that is before the paper can be distributed and cause confusion.

## (5.c) Defending against system problems

Code can become obsolete as programming languages develop and compilers are improved. Typically, compilers first warn that code is "deprecated" and then later versions reject the old code. Furthermore, when code is run on different computers, different operating systems, and with different compilers, it is common to obtain different results. Data, too, is subject to the same problems, but data standards and formats are far more stable than code standards, so "data rot" is less of a risk (but no less a problem when it occurs) than "software rot."

Additionally, errors can be the result of human slips, such as accidentally deleting a line of code or a line of data in a spreadsheet. Such corruption errors are hard to detect unless specific steps are taken to ensure the integrity of code and data [74]. Checksums are the simplest way to detect such errors, but during active research more refined techniques might be used in addition, for example checking that the number of rows of data in a spreadsheet monotonically increases. In the present paper, the JSON data is more structured than a spreadsheet matrix, and a number (as it happens, 30) of other consistency checks are imposed on the data.

To protect against version, portability and other problems, the Github repository for the present paper includes a check on software versions and a checksum check for all possibly affected files, including the data file. This does not solve the problem, but it ensures anyone developing or reproducing the paper's work will at least be forewarned of potential version or portability problems. The Github repository itself can be used to restore files that have been corrupted.

## (5.d) Problems of restrictive journal policies

Automatically generated variables are used throughout the paper and this Supplemental Material. As usual, LaTeX detects any spelling errors in the use of variables, thus helping protect the paper against typos that could otherwise mislead the paper's readers. Conveniently, LaTeX also supports sophisticated calculations itself [110], so the typeset paper can use any variable values in further calculations without going back to modify the data source file (in the present case, `data.js`). In practice this enables the author to avoid copying-and-pasting values from a data source or calculator, and then overlooking keeping them up to date with changes to the data or formula required.

For example, the caption of table 3 in the main paper calculates its "32 months" figure from the generated variables recording the repository date of cloning used to provide the data to construct the table. The number of months will of course be correctly updated if the paper's repository [119] is subsequently checked again:

> At the time of cloning and checking all repositories in February 2022, paper [119] still had nothing in its repository except a single file still saying "...code coming soon...," despite 32 months having already elapsed since the submitted paper had claimed the code could be accessed in its repository.

Of course, the data generation process itself checks that this surprising statement remains valid, and provides a warning if the wording may need revising.

Unfortunately, although using generated variables and analyses from a paper's data is a very simple technique to help make published papers more reliable, some journals and preprint servers (such as *IEEE Transactions on Software Engineering*, *PLOS ONE*, and *arXiv*) do not permit papers to be submitted using LaTeX source code that uses the standard `\input`, `\bibliography`, and other related commands. Typically they also do not support running any data collection or analysis either (which the present paper does when it clones repositories).

Another program (`expand.js`) was therefore written to recursively expand included files so the expanded version can be submitted adhering to any such restrictive policy. Of course, the expanded version now contains all variables as fixed constants, so the submitted paper is misleading and useless to other researchers if the data is modified — the effort to ensure all published numbers are automatically correct is defeated. Such restrictive publishing policies undermine reproducibility.

## (5.e) Code policies of sampled journals

It is noteworthy that none of the journals sampled permit any reliable style of managing data in published papers, such as described above in sections (5.b) and (5.d). In particular, for all the papers that had accessible code, the code included explicit (and relevant) data that was not archived *as* data in the journal repositories.

**Extract from *Royal Society Open Science* author guidelines**
— It is a condition of publication that authors make the primary data, materials (such as statistical tools, protocols, software) and code publicly available. These must be provided at the point of submission for our Editors and reviewers for peer-review, and then made publicly available at acceptance. [...] As a minimum, sufficient information and data are required to allow others to replicate all study findings reported in the article. Data and code should be deposited in a form that will allow maximum reuse. As part of our open data policy, we ask that data and code are hosted in a public, recognized repository, with an open licence (CC0 or CC-BY) clearly visible on the landing page of your dataset.

URL `royalsociety.org/journals/authors/author-guidelines/#data`

> Accessed 29 July 2020; the policy has been revised (undated, but accessed 2 February 2022) but retains the same principles; full policy now available via a DOI [111]. The policy still retains an emphasis on data accessibility, and continues a lack of awareness that code and data are equivalent and often mixed (see section 3).

**Extract from *Nature Digital Medicine* author guidelines**
— A condition of publication in a Nature Research journal is that authors are required to make materials, data, code, and associated protocols promptly available to readers without undue qualifications. [...] A condition of publication in a Nature Research journal is that authors are required to make unique materials promptly available to others without undue qualifications.

URL `www.nature.com/nature-research/editorial-policies/reporting-standards#availability-of-data`

> Accessed 29 July 2020; since updated to require [in part] "Upon publication, Nature Portfolio journals consider it best practice to release custom computer code in a way that allows readers to repeat the published results. Code should be deposited in a DOI-minting repository such as Zenodo, Gigantum or Code Ocean and cited in the reference list following the guidelines described here." (accessed 2 February 2022).

***Lancet Digital Health* author guidelines**
Journal has detailed data policies, but no code policy.

URL `marlin-prod.literatumonline.com/pb-assets/Lancet/authors/tldh-info-for-authors.pdf`

> Accessed 29 July 2020. Still no code policy (accessed 2 February 2022).

**Extract from *Journal of Vascular Surgery* author guidelines**
Journal has detailed data policies, but no code policy. While no *Journal of Vascular Surgery* papers were surveyed, the following statement on data policies is relevant for section (4.d) in the main paper:

— The authors are required to produce the data on which the manuscript is based for examination by the Editors or their assignees, should they request it. [...] The authors should consider including a footnote in the manuscript indicating their willingness to make the original data available to other investigators through electronic media to permit alternative analysis and/or inclusion in a meta-analysis.

URL `www.editorialmanager.com/jvs/account/JVS_Instructions%20for%20Authors2020.pdf`
Accessed 29 July 2020. Policy unchanged when accessed 2 February 2022.

## (5.f) Sample assessments and criteria

Assessment flags are highlighted in color to be clearer in the following tables.

Legend:

| | |
|---|---|
| $\mathsf{P}_{\mathsf{c}}$ | Journal has a code policy (see section 5.e) |
| $\mathsf{P}_{\mathsf{c\text{-}breach}}$ | Paper breaches journal code policy (see section 5.e) |
| $\mathsf{R}_{\mathsf{c}}$ | Paper uses a code repository (e.g., GitHub) |
| $\mathsf{R}_{\mathsf{c\text{-}empty}}$ | Code repository contains no code |
| $\mathsf{R}_{\mathsf{d}}$ | Paper uses a data repository (e.g., Dryad, Figshare, GitHub) |
| $\mathsf{S}_{\mathsf{NONE}}$ | No code available at all (note: code is not expected for standard models, systems or statistical methods) |
| $\mathsf{S}_{\mathsf{p}}$ | Paper says source code is available in principle |
| $\mathsf{S}_{+}$ | Paper or URL provides source code |
| $\mathsf{S}_{\mathsf{rigorous}}$ | Evidence that source code was developed rigorously |
| $\mathsf{S}_{\mathsf{tested}}$ | Evidence that source code has been run with a clean build and tested |
| $\mathsf{S}_{\mathsf{tools}}$ | Evidence of any tool-based development |
| $\mathsf{S}_{\mathsf{open\ source}}$ | Team or open source development |
| $\mathsf{S}_{\mathsf{otherSE}}$ | Other evidence of good practice; see details in summary table |
| $\mathsf{C}_{0}$ | Code has no non-trivial comments |
| $\mathsf{C}_{1}$ | Code only has trivial comments (e.g., copyright) |
| $\mathsf{C}_{2}$ | Helpful comments explaining code intent, rather than rephrasing the code |
| $\mathsf{C}_{+}$ | Code has substantial, useful comments and documentation |

| Ref | Data | Code |
|---|---|---|
| [112] | On request | "Code is available upon request from the corresponding author" (requested) $\mathsf{P_c}$ $\mathsf{S_p}$ |
| [113] | "The datasets used in the current study are available from the corresponding author upon reasonable request and under consideration of the ethical regulations" $\mathsf{R_d}$ | Matlab. Documented overview, but only trivial comments $\mathsf{P_c}$ $\mathsf{R_c}$ $\mathsf{S_+}$ $\mathsf{C_1}$ |
| [114] | "In accordance with Twitter policies of data sharing, data used in the generation of the algorithm for this study will not be made publicly available" | "Due to the sensitive and potentially stigmatizing nature of this tool, code used for algorithm generation or implementation on individual Twitter profiles will not be made publicly available" $\mathsf{P_c}$ $\mathsf{P_{c\text{-}breach}}$ $\mathsf{S_{NONE}}$ |
| [115] | "The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request."“” | "This code would be made available upon reasonable request." (requested) $\mathsf{P_c}$ $\mathsf{S_p}$ |
| [116] | Nothing available | Nothing available (despite building two voice-based virtual counselors) $\mathsf{P_c}$ $\mathsf{P_{c\text{-}breach}}$ $\mathsf{S_{NONE}}$ |
| [117] | "The datasets generated and analyzed during the study are not currently publicly available due to HIPAA compliance agreement but are available from the corresponding author on reasonable request" | Poor commenting, no documentation $\mathsf{P_c}$ $\mathsf{R_c}$ $\mathsf{S_+}$ $\mathsf{C_1}$ |
| [118] | "The dataset generated and analyzed for this study will not be made publicly available due to patient privacy and lack of informed consent to allow sharing of patient data outside of the research team" | No code available $\mathsf{P_c}$ $\mathsf{P_{c\text{-}breach}}$ $\mathsf{S_{NONE}}$ |
| [119] | "The datasets generated during and/or analyzed during the current study are not publicly available due to institutional restrictions on data sharing and privacy concerns. However, the data are available from the corresponding author on reasonable request" | Empty GitHub repository: "Code coming soon..." it says $\mathsf{P_c}$ $\mathsf{P_{c\text{-}breach}}$ $\mathsf{R_c}$ $\mathsf{R_{c\text{-}empty}}$ $\mathsf{S_{NONE}}$ |

| Ref | Data | Code |
|---|---|---|
| [120] | “The i2b2 data that support the findings of this study are available from i2b2 but restrictions apply to the availability of these data, which require signed safe usage and research-only. Data from UCSF are not available at this time as they have not been legally certified as being De-Identified, however, this process is underway and the data may be available by the time of publication by contacting the authors. Requesters identity as researchers will need to be confirmed, safe usage guarantees will need to be signed, and other restrictions may apply” | Basic documentation, very little comment $\mathsf{P_c}$ $\mathsf{R_c}$ $\mathsf{S_+}$ $\mathsf{C_1}$ |
| [121] | “Not available due to restrictions in the ethical permit, but may be available on request” | Trivial comments, no documentation $\mathsf{P_c}$ $\mathsf{R_c}$ $\mathsf{S_+}$ $\mathsf{C_1}$ |
| [122] | “The data that support the findings of this study are available in a deidentified form from Cleveland Clinic, but restrictions apply to the availability of these data, which were used under Cleveland Clinic data policies for the current study, and so are not publicly available” | “We used only free and open-source software” some of which is unspecified $\mathsf{P_c}$ $\mathsf{P_{c\text{-}breach}}$ $\mathsf{S_{NONE}}$ |
| [123] | “The i-ROP cohort study data for ROP is not publicly available due to patient privacy restrictions, though potential collaborators are directed to contact the study investigators …” | Not all code on GitHub, minor comments $\mathsf{P_c}$ $\mathsf{R_c}$ $\mathsf{S_+}$ $\mathsf{C_1}$ |
| [124] | Data available on Dryad $\mathsf{R_d}$ | Code and example runs available in R Markdown $\mathsf{P_c}$ $\mathsf{S_+}$ $\mathsf{C_+}$ |
| [125] | Data directly written into program code | Basic Matlab with routine comments $\mathsf{P_c}$ $\mathsf{P_{c\text{-}breach}}$ $\mathsf{S_+}$ $\mathsf{C_1}$ |
| [126] | Data available on Dryad plus publicly available data from the 1000 genomes project. Currently (apparently) for private view $\mathsf{R_d}$ | Code available for private view, though some code available with minor comments. Paper describes using two contrasting methods to help confirm correctness, “As an additional check, I also coded the calculation of D based on a probabilistic approach, using genotype frequencies in each population to calculate the expected frequencies of each possible two-genotype combination (electronic supplementary material, table S1). Essentially identical results were obtained.” but the contrasting method is not available $\mathsf{P_c}$ $\mathsf{S_p}$ $\mathsf{S_{otherSE}}$ $\mathsf{C_2}$ |

| Ref | Data | Code |
|---|---|---|
| [127] | Data available on Dryad $R_d$ | Reasonaby commented code on Dryad, but code is not complete and presumably never checked $P_c$ $S_p$ $C_2$ |
| [128] | On request | R lightly commented $P_c$ $S_p$ $C_1$ |
| [129] | No data required | Unrunnable incomplete code fragment $P_c$ $P_{\text{c-breach}}$ $S_p$ |
| [130] | Data embedded in PDF | No code available $P_c$ $P_{\text{c-breach}}$ $S_{\text{NONE}}$ |
| [131] | Data available on Dryad $R_d$ | Some comments, some code in Matlab $P_c$ $S_p$ $C_2$ |
| [132] | Partial data on Dryad $R_d$ | Documented R, including manual $P_c$ $R_c$ $S_+$ $C_+$ |
| [133] | No data required | "We constructed a bioeconomic model for an RSSF [restricted fishing effort small-scale fishery] using game theory" for which results are discussed, yet no code is available $P_c$ $P_{\text{c-breach}}$ $S_{\text{NONE}}$ |
| [134] | Data cited, not all available | Trivial documentation $P_c$ $R_c$ $S_+$ $C_1$ |
| [135] | On Figshare $R_d$ | On Figshare, large amount of disorganised and undocumented code. Helpful features to make usable for third parties $P_c$ $S_+$ $C_1$ |
| [136] | Data on Dryad $R_d$ | No code available $P_c$ $P_{\text{c-breach}}$ $S_{\text{NONE}}$ |
| [137] | Data on various web sites | No code available $P_c$ $P_{\text{c-breach}}$ $S_{\text{NONE}}$ |
| [138] | Data on request | "The coding used to train the artificial intelligence model are dependent on annotation, infrastructure, and hardware, so cannot be released." (!) Algorithm (not source code) available on request. $S_{\text{NONE}}$ |
| [139] | Data on request | Python scripts can be requested $S_p$ |
| [140] | Unspecified location on large website requiring registration $R_d$ | Has overall documentation but poorly commented Matlab code on GitHub $R_c$ $S_+$ $C_1$ |
| [141] | Available to researchers who meet criteria for access to confidential data | Despite the paper being a "deep learning algorithm" the code is not available $S_{\text{NONE}}$ |
| [142] | Data access conditional on approved study proposal | Almost completely uncommented Python, but does have a basic setup script $R_c$ $S_+$ $C_0$ |
| [143] | Unspecified locations on several large websites | Python used and apparently GitHub, but — an oversight? — no code is available $S_{\text{NONE}}$ |

## (5.g) Summary of assessments

| | | | |
|---|---|---|---|
| Number of papers sampled relying on code | 32 | 100% | |
| **Access to code** | | | |
| Have some or all code available | 12 | 38% | |
| Some or all code in principle available on request | 8 | 25% | |
| No code available | 12 | 38% | |
| **Evidence of basic good software engineering practice** | | | |
| Evidence program designed rigorously | 0 | 0% | |
| Evidence source code properly tested | 0 | 0% | |
| Evidence of any tool-based development | 0 | 0% | |
| Team or open source based development | 0 | 0% | |
| Other methods, e.g., independent coding methods | 1 | 3% | |
| **Documentation and comments** | | | |
| Substantial code documentation and comments | 2 | 6% | |
| Comments explain some code intent | 3 | 9% | |
| Procedural comments (e.g., author, date, copyright) | 10 | 31% | |
| No usable comments | 17 | 53% | |
| **Repository use** | | | |
| Code repository (e.g., GitHub) — 1 was empty | 10 | 31% | |
| Data repository (e.g., Dryad or GitHub) | 9 | 28% | |
| **Adherence to journal code policy (if any)** | | | |
| Papers published in journals with code policies | 26 | 81% | |
| Clear breaches of code policy (if any) | 11 | 42% | ($N = 26$) |

This is exactly the same table as table 2 from the main paper, reproduced here for convenience. See section (5.a) in this Supplemental Material for details of the process that generated it.

## h References for sampled papers

[112] Callahan A, Steinberg E, Fries JA, Gombar S, Patel B, Corbin CK and Shah NH, "Estimating the efficacy of symptom-based screening for COVID-19," *Nature Digital Medicine*, **3**(95):3pp, 2020. DOI `10.1038/s41746-020-0300-0`
Accessed 14 July 2020. Double-checked 17 January 2021.

[113] Kanzler CM, Rinderknecht MD, Schwarz A, Lamers I, Gagnon C, Held JPO, Feys P, Luft AR, Gassert R and Lambercy O, "A data-driven framework for selecting and validating digital health metrics: use-case in neurological sensorimotor impairments," *Nature Digital Medicine*, **3**(80):17pp, 2020. DOI `10.1038/s41746-020-0286-7` Code URL `github.com/ChristophKanzler/MetricSelectionFramework`
Accessed 14 July 2020. Double-checked 17 January 2021.

[114] Roy A, Nikolitch K, McGinn R, Jinah S, Klement W and Kaminsky ZA, "A machine learning approach predicts future risk to suicidal ideation from social media data," *Nature Digital Medicine*, **3**(78):12pp, 2020. DOI `10.1038/s41746-020-0287-6`
Accessed 14 July 2020. Double-checked 17 January 2021.

[115] Levine DM, Co Z, Newmark LP, Groisser AR, Holmgren AJ, Haas JA and Bates DW, "Design and testing of a mobile health application rating tool," *Nature Digital Medicine*, **3**(74):7pp, 2020. DOI `10.1038/s41746-020-0268-9`
Accessed 14 July 2020. Double-checked 17 January 2021.

[116] Kannampallil T, Smyth JM, Jones S, Payne PRO and Ma J, "Cognitive plausibility in voice-based AI health counselors," *Nature Digital Medicine*, **3**(72):4pp, 2020. DOI `10.1038/s41746-020-0278-7`
Accessed 14 July 2020. Double-checked 17 January 2021.

[117] Huang S, Kothari T, Banerjee I, Chute C, Ball RL, Borus N, Huang A, Patel BN, Rajpurkar P, Irvin J, Dunnmon J, Bledsoe J, Shpanskaya K, Dhaliwal A, Zamanian R, Ng AY and Lungren MP, "PENet a scalable deep-learning model for automated diagnosis of pulmonary embolism using volumetric CT imaging," *Nature Digital Medicine*, **3**(61):9pp, 2020. DOI `10.1038/s41746-020-0266-y` Code URL `github.com/marshuang80/PENet`
Accessed 14 July 2020. Double-checked 17 January 2021.

[118] Dhruva SS, Ross JS, Akar JG, Caldwell B, Childers K, Chow W, Ciaccio L, Coplan P, Dong J, Dykhoff HJ, Johnston S, Kellogg T, Long C, Noseworthy PA, Roberts K, Saha A, Yoo A and Shah ND, "Aggregating multiple real-world data sources using a patient-centered health-data-sharing platform," *Nature Digital Medicine*, **3**(60):9pp, 2020. DOI `10.1038/s41746-020-0265-z`
Accessed 14 July 2020. Double-checked 17 January 2021.

[119] Hofer IS, Lee C, Gabel E, Baldi P and Cannesson M, "Development and validation of a deep neural network model to predict postoperative mortality, acute kidney injury, and reintubation using a single feature set," *Nature Digital Medicine*, **3**(58):10pp, 2020. DOI `10.1038/s41746-020-0248-0` Code URL `github.com/cklee219/PostoperativeOutcomes_RiskNet`
Accessed 14 July 2020. Double-checked 17 January 2021.

[120] Norgeot B, Muenzen K, Peterson TA, Fan X, Glicksberg BS, Schenk G, Rutenberg E, Oskotsky B, Sirota M, Yazdany J, Schmajuk G, Ludwig D, Goldstein T and Butte AJ, "Protected Health Information filter (Philter): accurately and securely de-identifying free-text clinical notes," *Nature Digital Medicine*, **3**(57):8pp, 2020. DOI `10.1038/s41746-020-0258-y` Code URL `github.com/BCHSI/philter-ucsf`
Accessed 14 July 2020. Double-checked 17 January 2021.

[121] Choi D, Park JJ, Ali T and Lee S, "Artificial intelligence for the diagnosis of heart failure," *Nature Digital Medicine*, **3**(54):6pp, 2020. DOI `10.1038/s41746-020-0261-3` Code URL `github.com/ubiquitous-computing-lab/AI-CDSS-Cardiovascular-Silo`
Accessed 14 July 2020. Double-checked 17 January 2021.

[122] Hilton CB, Milinovich A, Felix C, Vakharia N, Crone T, Donovan C, Proctor A and Nazha A, "Personalized predictions of patient outcomes during and after hospitalization using artificial intelligence," *Nature Digital Medicine*, **3**(51):8pp, 2020. DOI `10.1038/s41746-020-0249-z`
Accessed 14 July 2020. Double-checked 17 January 2021.

[123] Li MD, Chang K, Bearce B, Chang BY, Huang AJ, Campbell JP, Brown JM, Singh P, Hoebel KV, Erdoğmuş D, Ioannidis S, Palmer W, Chiang MF and Kalpathy-Cramer J, "Siamese neural networks for continuous disease severity evaluation and change detection in medical imaging," *Nature Digital Medicine*, **3**(48):9pp, 2020. DOI `10.1038/s41746-020-0255-1` Code URL `github.com/QTIM-Lab/SiameseChange`
Accessed 14 July 2020. Double-checked 19 January 2021.

[124] Hoffman JI, Nagel R, Litzke V, Wells DA and Amos W, "Genetic analysis of *Boletus edulis* suggests that intra-specific competition may reduce local genetic diversity as a woodland ages," *Royal Society Open Science*, **7**(200419):13pp, 2020. DOI `10.1098/rsos.200419` Code URL `datadryad.org/stash/dataset/doi:10.5061/dryad.1g1jwstrw`
Accessed 22 July 2020. Double-checked 26 January 2021.

[125] Grönquist P, Panchadcharam P, Wood D, Menges A, Rüggeberg M and Wittel FK, "Computational analysis of hygromorphic self-shaping wood gridshell structures," *Royal Society Open Science*, **7**(192210):9pp, 2020. DOI `10.1098/rsos.192210` Code URL `royalsocietypublishing.org/doi/suppl/10.1098/rsos.192210`
Accessed 22 July 2020. Double-checked 26 January 2021.

[126] Amos W, "Signals interpreted as archaic introgression appear to be driven primarily by faster evolution in Africa," *Royal Society Open Science*, **7**(191900):9pp, 2020. DOI `10.1098/rsos.191900` Code URL `datadryad.org/stash/share/ichHKrWj7hqlznOaR6NQVzITgp40dlqWvWAgAxyafiQ`
Accessed 22 July 2020. Double-checked 26 January 2021.

[127] Gordon M, Viganola D, Bishop M, Chen Y, Dreber A, Goldfedder B, Holzmeister F, Johannesson M, Liu Y, Twardy C, Wang J and Pfeiffer T, "Are replication rates the same across academic fields? Community forecasts from the DARPA SCORE programme," *Royal Society Open Science*, **7**(200566):7pp, 2020. DOI `10.1098/rsos.200566` Code URL `royalsocietypublishing.org/doi/suppl/10.1098/rsos.200566`
Accessed 22 July 2020. Double-checked 26 January 2021.

[128] Evans D and Field AP, "Predictors of mathematical attainment trajectories across the primary-to-secondary education transition: parental factors and the home environment," *Royal Society Open Science*, **7**(200422):20pp, 2020. DOI `10.1098/rsos.200422` Code URL `osf.io/a5xsz/?view_only=87ae173f775b40d79d6cd0fdcf6d4a9c`
Accessed 22 July 2020. Double-checked 26 January 2021.

[129] Beale N, Battey H, Davison AC and MacKay RS, "An unethical optimization principle," *Royal Society Open Science*, **7**(200462):11pp, 2020. DOI `10.1098/rsos.200462`
Accessed 22 July 2020. Double-checked 26 January 2021.

[130] Cherevko AA, Gologush TS, Petrenko IA, Ostapenko VV and Panarin VA, "Modelling of the arteriovenous malformation embolization optimal scenario," *Royal Society Open Science*, **7**(191992):16pp, 2020. DOI `10.1098/rsos.191992`
Accessed 22 July 2020. Double-checked 26 January 2021.

[131] Soczawa-Stronczyk AA and Bocian M, "Gait coordination in overground walking with a virtual reality avatar," *Royal Society Open Science*, **7**(200622):19pp, 2020. DOI `10.1098/rsos.200622` Code URL `datadryad.org/stash/dataset/doi:10.5061/dryad.vx0k6djnr`
Accessed 22 July 2020. Double-checked 26 January 2021.

[132] Duruz S, Vajana E, Burren A, Flury C and Joost S, "Big dairy data to unravel effects of environmental, physiological and morphological factors on milk production of mountain-pastured Braunvieh cows," *Royal Society Open Science*, **7**(200638):13pp, 2020. DOI `10.1098/rsos.200638` Code URL `github.com/SolangeD/lactModel`
Accessed 22 July 2020. Double-checked 26 January 2021.

[133] de Azevedo EZD, Dantas DV and Daura-Jorge FG, "Risk tolerance and control perception in a game-theoretic bioeconomic model for small-scale fisheries," *Royal Society Open Science*, **7**(200621):11pp, 2020. DOI `10.1098/rsos.200621`
Accessed 22 July 2020. Double-checked 26 January 2021.

[134] Abdolhosseini-Qomi AM, Jafari SH, Taghizadeh A, Yazdani N, Asadpour M and Rahgozar M, "Link prediction in real-world multiplex networks via layer reconstruction method," *Royal Society Open Science*, **7**(191928):22pp, 2020. DOI `10.1098/rsos.191928` Code URL `github.com/UT-NSG/LRM`
Accessed 22 July 2020. Double-checked 26 January 2021.

[135] Webster J and Amos M, "A Turing test for crowds," *Royal Society Open Science*, **7**(200307):12pp, 2020. DOI `10.1098/rsos.200307` Code URL `figshare.com/collections/Supplementary_information_for_Webster_J_and_Amos_M_A_Turing_Test_for_Crowds_/4859118/1`
Accessed 22 July 2020. Double-checked 26 January 2021.

[136] Zhu Y-l, Wang C-J, Gao F, Xiao Z-x, Zhao P-l and Wang J-y, "Calculation on surface energy and electronic properties of $\mathrm{CoS}_2$," *Royal Society Open Science*, **7**(191653):12pp, 2020. DOI `10.1098/rsos.191653`
Accessed 22 July 2020. Double-checked 26 January 2021.

[137] Yu B, Scott CJ, Xue X, Yue X and Dou X, "Derivation of global ionospheric Sporadic E critical frequency ($f_o$Es) data from the amplitude variations in GPS/GNSS radio occultations," *Royal Society Open Science*, **7**(200320):15pp, 2020. DOI `10.1098/rsos.200320`
Accessed 22 July 2020. Double-checked 26 January 2021.

[138] Joon-myoung K, Younghoon C, Ki-Hyun J, Soohyun C, Kyung-Hee K, Seung D B, Soomin J, Jinsik P and Byung-Hee O, "A deep learning algorithm to detect anaemia with ECGs: a retrospective, multicentre study," *Lancet Digital Health*, **2**(7):9pp, 2020. DOI `10.1016/S2589-7500(20)30108-4`
Accessed 24 July 2020. Double-checked 26 January 2021.

[139] Zhu H, Cheng C, Yin H, Li X, Zuo P, Ding J, Lin F, Wang J, Zhou B, Li Y, Hu S, Xiong Y, Wang B, Wan G, Yang X and Yuan Y, "Automatic multilabel electrocardiogram diagnosis of heart rhythm or conduction abnormalities with deep learning: a cohort study," *Lancet Digital Health*, **2**(7):9pp, 2020. DOI `10.1016/S2589-7500(20)30107-2`
Accessed 24 July 2020. Double-checked 26 January 2021.

[140] Fung R, Villar J, Dashti A, Ismail LC, Staines-Urias E, Ohuma EO, Salomon LJ, Victora CG, Barros FC, Lambert A, Carvalho M, Jaffer Y A, Noble JA, Gravett MG, Purwar M, Pang R, Bertino E, Munim S, Min AM, McGready R, Norris SA, Bhutta ZA, Kennedy SH, Papageorghiou AT and Ourmazd A, "Achieving accurate estimates of fetal gestational age and personalised predictions of fetal growth based on data from an international prospective cohort study: a population-based machine learning study," *Lancet Digital Health*, **2**(7):7pp, 2020. DOI `10.1016/S2589-7500(20)30131-X` Code URL `github.com/ki-analysis/manifold-ga`
Accessed 24 July 2020. Double-checked 26 January 2021.

[141] Sabanayagam C, Xu D, Ting DSW, Nusinovici S, Banu R, Hamzah H, Lim C, Tham Y-C, Cheung CY, Tai ES, Wang XY, Jonas JB, Cheng C-Y, Lee ML, Hsu W and Wong TY, "A deep learning algorithm to detect chronic kidney disease from retinal photographs in community-based populations," *Lancet Digital Health*, **2**(7):7pp, 2020. DOI `10.1016/S2589-7500(20)30063-7`
Accessed 24 July 2020. Double-checked 26 January 2021.

[142] Monteiro M, Newcombe VF, Mathieu F, Adatia K, Kamnitsas K, Ferrante E, Das T, Whitehouse D, Rueckert D, Menon DK and Glocker B, "Multiclass semantic segmentation and quantification of traumatic brain injury lesions on head CT using deep learning: an algorithm development and multicentre validation study," *Lancet Digital Health*, **2**(7):8pp, 2020. DOI `10.1016/S2589-7500(20)30085-6` Code URL `github.com/biomedia-mira/blast-ct`
Accessed 24 July 2020. Double-checked 27 January 2021.

[143] Liu K-L, Wu T, Chen P-T, Tsai Y M, Roth H, Wu M-S, Liao W-C and Wang W, "Deep learning to distinguish pancreatic cancer tissue from non-cancerous pancreatic tissue: a retrospective study with cross-racial external validation," *Lancet Digital Health*, **2**(7):10pp, 2020. DOI `10.1016/S2589-7500(20)30078-9`
Accessed 24 July 2020. Double-checked 27 January 2021.