Improving science that uses code

Harold Thimbleby

See Change Fellow in Digital Health, Swansea University
Email: harold@thimbleby.net

Scientists rarely assure the structure and quality of the code they rely on, and they rarely make it accessible for scrutiny. Even if available, scientists rarely provide adequate documentation to understand or use their code reliably. When code used in science avoids adequate scrutiny published science will be unreliable, because results — data, graphs, images, inferences, etc — generated by the code will themselves be unreliable.

This paper justifies and proposes ways to improve research and science relying on code:

1. **Professional Software Engineers can help, particularly in critical fields such as public health, climate change, and energy.**
3. **"Software Engineering Boards," analogous to Ethics or Institutional Review Boards, should be instigated and used.**
4. **The Reproducible Analytic Pipeline (RAP) methodology can be generalized to cover code and Software Engineering methodologies, in a generalization this paper introduces called RAP+. RAP+ (or comparable interventions) could be supported and encouraged in journal, conference, and funding body policies.**

The key point is that as code is now an inextricable part of science it should be supported by competent Software Engineering input, analogously to statistics being properly supported by critical and competent statistics input. The Supplemental Material provides a summary of Software Engineering best practice relevant to scientific research, including further suggestions for RAP+ processes.

"Science is what we understand well enough to explain to a computer."
Donald E. Knuth in $A = B$ [**?**]

"I have to write to discover what I am doing."
Flannery O'Connor, quoted in *Write for your life* [**?**]

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

"From its earliest times, science has operated by being open and transparent about methods and evidence, regardless of which technology has been in vogue."
Editorial in *Nature* [**?**]

Keywords: Computational Science; Software Engineering; Reproducibility; Scientific scrutiny; Reproducible Analytic Pipeline (RAP & RAP+)

Received 30 April 2022

1. INTRODUCTION

Unreliable, often unstated and unexplored, code and computational dependencies (including using AI or ML) in science are widespread. Furthermore, code is rarely published or made accessible in a useful form, nor is it professionally scrutinized. Too often, therefore, code and results claimed do not contribute to reliable science, and do not support replication or reproduction on which future science can be firmly based.

This paper makes an explicit analogy between the use of statistics, which has clear standards for reliable use

CONTENTS

1 Introduction **1**

2 Background **2**
2.a Related code quality concerns 4
2.b Computable papers 5
2.c RAP: Reproducible Analytical Pipelines . . . 7

3 The statistics/code analogy **7**
3.a The siren call of over-fitting code 10

4 The conventional role of code **12**
4.a The deceptive simplicity of code 12
4.b The low status of coding 13
4.c The critical role of code is often ignored . . . 13
4.d Bugs, code and programming 14
4.e Long-term problems of unreliable code 14

5 State of the art **15**
5.a Case study: Pandemic modeling 15
5.b Concerns with reproducibility 16
5.c Beyond pandemic modeling 17
5.d The narrow emphasis on data 18

6 Extending RAP to RAP+ **19**

7 The paper as a scientific laboratory **20**

8 Call to action **21**
8.a Action must be interdisciplinary 22
8.b Methodological statements 23
8.c Benefits beyond science 24

9 Conclusions **24**

and presentation, and the use of code. Like statistics, code is often relied on to support key results in scientific publications, yet code is generally informal, inaccessible and incorrect. Just as sound experimental methods and sound statistics generally rely on professional specialist input, it is argued here that good use of code must rely on professional Software Engineering input [**?**, **?**], and more strategically on Computational Thinking [**?**, **?**] — an accessible form of Software Engineering that consciously applies the ideas universally, far more widely than just to software and coding.

This paper was initially motivated by concerns about the poor quality of code used in high-profile epidemiology research, because of its significance for informing and driving public health responses to the Covid pandemic. A pilot survey implies that such problems are ubiquitous and by no means limited to epidemiology: see the tables 1 & 2 for a summary of the sample, and see the Supplemental Material for further details of the survey. In the survey, *no* papers claimed or provided evidence that their code was adequately tested or rigorously developed; *none* used methodologies like RAP or RAP+ (described below). Only one paper mentioned any Software Engineering methods, albeit without technical details.

In the survey sample 81% of papers were published in journals that have code policies (which themselves are weak), but 42% of surveyed papers published in those journals breached their own policies. One paper declared it had accessible code, but the relevant repository was and still remains empty. The findings are comparable to problems increasingly recognized for data and data access (reviewed in section 2.a): code problems form part of the *reproducibility crisis* [e.g., **?**, **?**].

This paper therefore argues that code should be developed and discussed in a professional, rigorous, and supportive environment that facilitates quality science with clear presentation and appropriately rigorous scrutiny of code. Its main contribution is to suggest straightforward ways to enable this. The proposals may not be "the" right or best ways, but it is hoped the case studies and arguments presented here persuade readers that the proposals are at least a productive way to start pointing in the right direction, and inform raising the profile and constructively debating the issues more widely.

An extensive online Supplemental Material appendix to this paper provides additional resources, including brief details of many Software Engineering practices relevant to supporting quality science. The supplement will be of particular interest to research software engineers supporting non-software-specialist scientists.

2. BACKGROUND

The discoveries and inventions of scientific technologies and instruments like microscopes, telescopes, and X-rays, drive and expand the sciences. There are fascinating periods as new ideas and science unify; for example, thermometers could not measure temperature in any meaningful way until the science was mature. For over a century, there was no agreement on definitions of temperature, how to calibrate thermometers, or what units they measured in.

Paradoxically the science could not mature until there was consensus in scientific methodologies for thermometry, and having a consensus in turn depended on reproducible thermometer measurements and a thorough understanding of the science, including all the con-

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, broken down further in Table 2. Survey methodology and data is provided in the Supplemental Material. (The survey does not include the motivating papers [**?**, **?**, **?**], none of which provide code or code summaries; see section 5.a.)

Number of papers sampled relying on code	32	100%	
Access to code			
Some or all code available	12	38%	
Some or all code in principle available on request	8	25%	
Requested code actually made available (within 2 years 7 months*)	0	0%	
Evidence of any 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			
Used code repository (e.g., GitHub)	9	28%	
Used data repository (e.g., Dryad or GitHub)	9	28%	
Empty repository	1	3%	
Evidence of documented processes			
Evidence of RAP/RAP+ or any other principles in use to support scrutiny	0	0%	
Adherence to journal code policy (if any)			
Papers published in journals with code policies	26	81%	
Clear breaches of journal code policy (if any)	11	42%	($N = 26$)

*Time of 2 years 7 months is wait between code request and date of generating this table.

TABLE 2. Breakdown of pilot survey of peer-reviewed science papers relying on code.

Level 0	Level 1	Level 2	Level 3
Journal encourages code sharing – or says nothing.	Article states whether code is available and, if so, where to access them.	Code must be posted to a trusted repository. Exceptions must be identified at article submission.	Code must be posted to a trusted repository, and reported analyses will be reproduced independently before publication.

TABLE 3. The TOP committee's recommended levels for journal article code transparency. Level 0 is provided for a comparison that does not meet any TOP requirements. Concerns about the interpretation of "reproduced independently," as required at level 4, are raised in section 5.b.

founding factors that were being misunderstood [**?**].[1] Science matured from no quantitative interest in temperature, through a complex process of hand-in-hand theoretical-and-technical maturation, until today, when we have robust off-the-shelf instruments that measure temperature in reliable, repeatable, internationally standardized units, following international quality standards.

Computers are a unique, new technology, far more flexible and challenging than thermometers. Understanding computers and integrating them into science is likely to be harder than the development of modern thermometry. Computation not only expands science's paradigms and supports new discoveries (particularly with AI), but it also *does* new science — almost all modern laboratory instrumentation, including thermometers, is heavily computerized. Pushing the boundaries of science, then, involves pushing the boundaries of computer science. The synergy runs deep: for instance, while particle physics relies on powerful supercomputers, quantum physics itself is developing more powerful quantum computing.

"Computational science" has come to mean a particular style of science based on developing and using explicit computational models, but, really, *all* of science is now computational in this sense. Computational science is not just restricted to specialized fields like computational chemistry, genomics, big data … In all fields of science, computation is used at every step, from calculations of course, through note taking, sound and image processing, literature searches, analysis and statistics, correspondence with co-authors and editors, through to typesetting, distributing and archiving the final publications. All areas of science are being profoundly computerized. Furthermore, developments in computer science themselves drive science such that earlier science is even becoming obsolete as the computer technology moves on [**?**].

2.a. Related code quality concerns

Publishing high quality computer code has been strongly advocated since the earliest times, such as the *Communications of the ACM* in its first issue in its first volume published in 1958, where it outlined its algorithm publication policy. The new policy was illustrated with a square root algorithm [**?**]. However, publishing code in the computer science literature is distinct from publishing high quality general science that depends on code, which is the concern of the present paper. Of course, as a special case, Computer Science too can also benefit from improving ways to reliably use code.

In almost all published science, the code it relies on is taken for granted, just as in routine chemistry the quality of the glassware is not at issue. While chemists are trained in reliable methodologies, so taking quality glassware for granted is reasonable. Code is a newer innovation. Reliable code is a current research programme in its own right, and has resulted in calls for a Grand Challenge research effort [**?**]. Inevitably, because of these reasons taken collectively, much published science depends on unreliable code that is not explicitly discussed, was not peer-reviewed, and is not open to scrutiny, reproduction, or reuse by the scientific community.

A study of 863,878 Python-coded Jupyter notebooks [**?**] found a 76% failure rate for code to complete execution successfully. Trisovic *et al.* [**?**] performed a study of 9,000 research codes written in the language R on Dataverse, an open-source repository maintained by Harvard University's Institute for Quantitative Social Sciences. They found a comparable result that 74% of the code files analyzed failed. These results are consistent with this paper's findings, as summarized in tables 1 & 2.

The authors of [**?**] make technical recommendations to improve reproducibility, such as "Abstract code into functions, classes, and modules and test them." More generally, the authors of [**?**] recommend establishing Working Groups to support reproducible research — and in fact Dataverse has already done so. These ideas may be compared to the present paper's proposal of Software Engineering Boards (SEBs), as discussed in detail in section 8, supported by more specific suggestions in the Supplemental Material.

These concerns about code are part of the reproducibility crisis for science generally [**?**,**?**,**?**,**?**,**?**,**?**]. Concern has led to a new journal *ReScience C* to explore and encourage the explicit replication of previously published research [**?**]; furthermore, *ReScience C* promotes new and open-source implementations in order to ensure that the original research is reproducible. However, most of its published concern focuses on data rather than code.

To start to address code quality issues, the Transparency and Openness Promotion Committee first met in 2014, and has since been promoting Transparency and Openness Promotion, TOP, starting with journal publication policies [**?**]. TOP covers

[1] One example: if the volume of mercury is chosen to measure temperature, a confounding factor is that the volume of the container measuring the mercury volume also increases with temperature (but at a different rate), so the volume measurement is inaccurate.

citation standards, replication standards, and code standards amongst others. TOP recommends levels of compliance to their recommendations, where level 0 does not meet the standard, and levels 1 to 3 are increasingly stringent. The TOP levels for code are shown in table 3. The TOP standards continue to develop, and are now maintained on a wiki at URL `osf.io/9f6gx/wiki/Guidelines` [**?**]. Note that TOP aligns with but is a stricter approach than the Findable, Accessible, Interoperable and Reusable (FAIR) initiative, which is critiqued in section 5.d.

As the present paper argues, developing quality code is widely under-appreciated, which leads to a vicious cycle of lack of acknowledgement, invisibility, and being unable to recruit adequately competent coders (section 3). The term *research software engineer* was coined in 2012 to help address this problem, and to stimulate thinking about researchers' career paths. The Society of Research Software Engineering (URL `https://society-rse.org`) has been established to further promote research software engineer interests.

There is no shortage of computational tools available to help address the problems. However, it should be noted that such tools are not a panacea, as the study [**?**] cited above makes clear. This suggests that human support for improving coding and reproduction quality, perhaps in the form of Working Groups or Boards, as this paper suggests, will be important.

In addition to unintentional problems with code quality and reproducibility, scientific misconduct occurs when the outcome is intentional. While pure plagiarism, which is misconduct, generally does not affect the quality of reported science, when data or code is fraudulently manipulated to have realistic properties, the results are likely to be destructive. The Wakefield MMR fraud linking vaccines and autism published in *The Lancet* took 12 years before it was retracted, and has been extraordinarily destructive [**?**].

A recent meta-analysis of surveys of scientific misconduct estimates that nearly 2% of scientists at least once fabricated, falsified, or modified results, and over a third undertook other questionable research practices [**?**]. The meta-analysis qualifies the figures carefully, but these are alarming rates with any qualifications; indeed, the authors suggest that as misconduct is a sensitive issue, the rates are likely to be under-estimates.[2]

While technical solutions like using AI may help, it is notable that many misconduct issues can be detected and constructively managed prior to publication using the same methods as will improve research reproducibility, as discussed throughout the present paper.

2.b. Computable papers

"Electronic lab notebooks" (ELNs) [**?**], emphasize computer tools that specifically support laboratory notebook authoring and editing. In contrast, "computable papers" aim to support the scientific paper authoring process: the emphasis is that papers should produce faithful results from code embedded in (or easily accessible from) the text of the paper.

If, for example, HTML is being used to prepare a computational paper, it could include Javascript code like `<script> document.write(responses.total) </script>`, which would insert the result of running that Javascript code into the paper, such as the number into a sentence in an empirical paper, like "We collected data and obtained 754 responses."

Systems like LaTeX (which was used for the current paper) can combine advanced typesetting with basic calculations (e.g., here we calculate 10! = 3628800 by writing LaTeX code *in* this paper). More practically, a separate program can generate a file of LaTeX definitions for data, tables, and cross-references, etc, as needed for a paper.

> For the present paper, as a concrete example of this approach, the pilot survey analyzed **32** papers with **264** authors. The paper **[?]**, discussed below in section **5.a**, relied on code composed of **229** files, with over **25 thousand** lines of code (how this compares with files in the survey is summarized in table **4** in the Supplemental Material).

In the "old days" such numbers (32, 264, etc) quoted above would have been manually worked out, then eyeballed and typed up by the authors. Instead, in this paper, those numbers (and many more) were computed automatically. Furthermore, they update automatically when the data or calculations change. They were inserted into the text of this paper automatically, and are auditable back to their sources.

The idea is easy to implement. Code can generate text like `\newcommand{\numberOfAuthors}{264}` (where the 264 is the calculated number) save that line of text to a file, which is then input into the paper where it is typeset in the normal way. Then, when and wherever the authors write the name `\numberOfAuthors` in their paper's text, the typeset paper says 264, or whatever the correct value is at the time. Some easy LaTeX coding can then present the numbers in the publisher's

[2] The author's survey of scientists publishing in the *Journal of Machine Learning* had comparable results [**?**].

preferred style, such as zero, one, … nine, 10, 11 … 1,000 etc, as done (for example) in this paper's footnote 4.

Rather than use quirky languages like LaTeX computable papers more often build on general-purpose programming languages. Python and R, for instance, have powerful features for processing data, plotting graphs, doing statistics, and more.

The key concept in computable papers is that as the authors of a paper collect or revise data or calculations or otherwise update it, the results in the paper update automatically, and all the statistics, graphs, and analysis reported in the paper remain correct — *with no further work from the authors*. Indeed, if an author corrects a mistake, the correction will apply automatically to all future edits.

However, there is no structure to using general-purpose systems like HTML, LaTeX, or Rmarkdown. Many authors therefore prefer a more structured approach imposed by tools designed for the purpose. A taut review is Perkel [**?**], but here four tools, WEB, *Mathematica*, Jupyter, and knitr, will serve to sample the variety of approaches that are available:

- WEB is the earliest tool reviewed here. WEB was developed by Donald Knuth in 1984 [**?**, **?**] as a batch (non-interactive) tool to support his then radical new concept of *literate programming*. The idea was to facilitate programmers write literate documentation for their code. WEB combines a sequential documentation file with code that can be presented in any order, thus overcoming the problem that the best explanation of a program is not necessarily written in the same order as the code it explains. The original WEB allowed Pascal programs to be documented in TeX, but many variants of WEB have since been developed that are more flexible in the systems they support.

  WEB documents an entire program, but there are variants such as relit [**?**] that allow arbitrary parts of programs to be documented, and hence are useful for normal scientific papers that need to explain algorithms, but do not need to show or explain the entire code.

  In contrast to the other tools reviewed here, literate programming is intended to produce high quality publications *about* code, rather than publications *using* the code, inserting output, such as graphs, generated by running it.
- *Mathematica* is one of the earliest fully interactive notebook tools for computational papers. Notebooks were developed by Theodore Gray in 1988 [**?**]. Notebooks consist of a collection of "cells." Some notebook cells are labelled as text or as section headings, but if a cell is labelled as code, it can be run and it will normally generate a new cell following it as its output. The new cell can be numbers, tables, mathematics, a plot, an image, or even more text — the arbitrary output of running code, in fact. Moreover, a notebook itself is a *Mathematica* expression, so it — the paper itself — can be analyzed or manipulated by code in any way. In particular, the entire notebook structure and contents can be checked for consistency and correctness in any way the author chooses.

  A *Mathematica* notebook can be published directly as a paper, but some code might be distracting. Typically the author therefore hides some or all code cells that are irrelevant to the narrative of the paper, but which nonetheless were required to generate the results presented.

  The user manual for *Mathematica* itself was written as a *Mathematica* notebook, which ensured all its examples actually worked — and probably helped ensure the correctness of the *Mathematica* code used behind the scenes (see section 7). The book is now the largest example of software documentation in existence: in its latest edition it runs to over 10,000 pages.
- Jupyter was developed by Fernando Pérez and Brian Granger [**?**], and takes a similar approach to *Mathematica*, but is open-source and not closely integrated with any particular programming language, as *Mathematica* is. Jupyter can be installed on a PC or run over the web.

  Jupyter is both an authoring tool and a framework on which to build other tools: thus Google's colaboratory is built on top of Jupyter, using it as a foundation but making stylistic changes, including providing free computational resources.

  Many extensive examples of using Jupyter notebooks (and other good practice, such as using repositories) to support large scale science projects can be found at the Gravitational Wave Open Science Center at
  URL `www.gw-openscience.org`.
- knitr [**?**] is a powerful culmination of a variety of tools, Pweave, Sweave, and ideas from literate programming. Knitr combines a markdown document with R code, and is a more powerful approach than the analogous, and perhaps more familiar, HTML+Javascript example shown above to motivate this section.

Thimbleby [**?**] is a 1999 example of a peer-reviewed paper (about user interface design) written as a *Mathematica* notebook, which makes the point that a distinctive feature of its methodology is that the *Mathematica* notebook creates a fully inspectable and replicable process. The notebook is available on the author's web site; it can be checked by others, or easily extended or repurposed to support new research — and it still works 24 years later.

There are tools to make using code more convenient and more reproducible, as this section briefly reviewed, but they are rarely used or used haphazardly, as the next section shows.

2.c. RAP: Reproducible Analytical Pipelines

Writing a paper typically starts in a word processor (such as Microsoft Word), sketching an outline, writing boiler-plate text (such as the authors' names and standard section headings), and then gradually building up the evidence base (including citing the literature) that the paper relies on. The process will be concurrent with many other activities — grant writing, writing up lab books, negotiating authorship, protecting IP, workshops, finding publication outlets, and so on.

Table 4 illustrates the core pipeline of how experiments and data are used to provide information on which analysis and calculations are based, the results of which are then edited into a paper.

For clarity, the schematic pipeline in table 4 omits many steps in the creative scientific process. Furthermore, each step is iterated and modified as the research progresses, and, indeed, as referees require revision. The point is that in typical scientific practice each step in the table is largely or entirely manual, typically selecting and copying output from the previous phase, and then pasting and editing the results into the next. The pipeline of data $\rightarrow \cdots \rightarrow$ paper is then iterated by hand as the various components are refined and improved until the authors are happy with the final paper.

As problems are found in a paper, the data, calculations and code are debugged, refactored, and refined. The process is rarely systematic, and even less likely to be documented — after all, the atomic steps seem to be innocuous copy and paste actions. The final paper and the ideas it embodies are what matters.

The insight of the reproducible analytic pipelines (RAP) proponents is that every time any step in the pipeline is performed it could have been automated [**?**, **?**, **?**]. If automated, it could then be repeated reliably — unlike a manual cut and paste which is error-prone every time it is done. It can be repeated if any experimental data, literature, or other knowledge changes, and the paper's analysis brought up to date with ease. In particular, any other researcher, whether part of the authorship team or a later reader of the paper, can reproduce it reliably if the RAP process is made available. Table 5 provides a brief summary of RAP principles.

For example, if the paper in question is a systematic review, it could be kept current by automatically re-running the programmed atomic actions that it was built with. Indeed, this ability is one of the original motivations of RAP, where Government agencies could easily generate up to date reports on request without having to repeat all the manual work and risk making procedural errors doing so. Furthermore, every time they update a publication, the RAP pipeline itself is reviewed and improved, so the quality of the reproduced work improves — unlike in a non-RAP process where new errors are generally introduced every time the work is revisited.

RAP not only helps develop reproducible science, and improve the quality of the science as the authors debug and refine their methodology, it also provides a precise audit trail that can be used to protect against fraud, as discussed in [**?**], and can in principle be largely automated and perform checks much faster and more efficiently than conventional *post hoc* investigations.

RAP embodies Donald Knuth's comment, also quoted after this paper's abstract,

> "Science is what we understand well enough to explain to a computer"
>
> from the foreword to $A = B$ [**?**]

The corollary is that if we are doing arbitrary cut and paste that has not been programmed into a computer, then we are not doing good science. Science is in principle an algorithmic process, and therefore, as Knuth says, if we understand well enough what we are doing in science, we can explain it as code, as RAP, for a computer to run and rerun.

3. THE STATISTICS/CODE ANALOGY

The central role of computational methods in science may be fruitfully compared to statistics, an established 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

Data sources →	Analysis →	Select results for write up →	Submit for publication
Experiments Standard data Search engines Literature Sensors ⋮	Hand calculations Packages SPSS etc Graphics packages Specially-written code ⋮	Copy & paste and edit data (text, images, graphs, etc) into paper	Final paper

TABLE 4. A simplified schematic of the publication pipeline. For clarity, the pipeline has been linearized; in general, there will be repetitive cyclic iteration and refinement. The RAP and RAP+ approaches encode the normally manual steps in the pipeline processes so that they can be run automatically, and hence reproduce the results that underpin the final paper. The encoded RAP+ algorithms can be shared with other scientists, scrutinized, simplified and optimized, and themselves turned into publishable objects — they are scientific instruments, just like thermometers or DNA sequencers. Additional schematics are provided in section 12 in the Supplemental Material.

1. Peer-review used to ensure the process is reproducible and to identify improvements
2. No or minimal manual interference; for example copy-paste, point-click and drag-drop steps replaced using computer code that can be inspected by others
3. Open-source programming languages so that processes do not rely on proprietary software and can be reproduced by others
4. Version control software, such as Git, to guarantee an audit trail of changes made to code
5. Publication of code, whenever possible, on code hosting platforms such as GitHub to improve transparency
6. Well-commented code and embedded documentation to ensure the process can be understood and used by others
7. Embedding of existing quality assurance practices in code, following guidance set by recognized organizations

Adopting RAP principles is not necessarily about incorporating all of the above: implementing just some of these principles will generate valuable improvements.

TABLE 5. A minimum standard of RAP, based on the UK Statistics Authority summary [**?**].

of their own statistical skills, so careful scientists generally work closely with professional statisticians.

In good science, all statistics, methods and results are reported very carefully and in precise detail [**?**, **?**, **?**], generally following strict journal or disciplinary guidelines. A statistical claim in a paper might be summarized as follows:

> "Random intercept linear mixed model suggesting significant time by intervention-arm interaction effect. […] Bonferroni adjusted estimated mean difference between intervention-arms at 8-weeks 2.52 (95% CI $0.78, 4.27, p = 0.0009$). Between group effect size $d = 0.55$ (95% CI 0.32, 0.77)." [**?**]

This wording formally 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 — it is a clear, rigorous, and readily interpreted way to express uncertainty in results. Moreover, behind any such brief paragraph is a substantial, rigorous, and appropriate statistical analysis.

Scientists write like this and conferences and journals require it because statistical claims need to be properly accountable and documented in a clear way. The journal *Science*, for example, in its many explicit and quite technical statistics requirements requires

> "Adjustments made to alpha levels (e.g., Bonferroni correction) or other procedure used to account for multiple testing (e.g., false discovery rate control) should be reported." [**?**]

Spiegelhalter [**?**] 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 13). 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; in fact, there is widespread concern about the poor quality of statistics in science [**?**, **?**]. While it is assumed that statistics should be peer-reviewed, and that review will often lead to improvement, these critical papers show that reviewers and editors are often failing to pick up on poor statistics.

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, do the data sampling and processing, to operate the sensors or surveys that generate the data, and to process it. Many papers now explore the contribution of AI and ML to their fields. 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 the research.

In short, computers, data, and computer code are central to modern science, not just to the explicitly computational sciences. Some AI work is uncovering biases and ethical issues that were previously unrecognized, so computational sciences are not just routinely contributing to existing science but extending its reach and improving its quality.

However, using any 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 subject to unintentional bias [**?**, **?**]. All these issues are non-trivial concerns requiring technical expertise to manage well. As with statistics, good answers to such "technical" issues makes the science that relies on them better.

A common oversight in scientific papers is to present a model, such as a set of differential equations, but omit how that model was reliably 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, and possibly even no link to the code at all.

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 planned and 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 concerns the many papers that disclose as part of their methodology that they used a particular software package, for example

> "Data analyses were performed using SAS 9.2 (SAS Institute, Cary, North Carolina, USA)." [**?**]

but without giving any further details. The paper cited above describes one of its multivariate analyses as "multiple correspondence analysis (MCA) and ascendant hierarchical clustering" with no specific details — reproducing the work would be non-trivial since the methodology would have to be reconstructed from scratch, even assuming the data was made available by the authors as the paper gives no details of the data used or how to obtain it.

The problem is that the common practice of declaring using a named computational system (such as SAS in this case) does not help scrutiny in the least, as such systems can do almost anything. *How* those analyses might have been performed is not discussed, and one assumes it follows that the analyses could therefore not have been properly reviewed for basic scientific competence during the publication process.

A reviewer, if nobody else, needs to actually examine the code used and its documentation to assess whether the analysis presented in the paper is appropriate and sufficiently reliable. Furthermore, if the analysis in this case actually depended on using SAS version 9.2, and not *any* general purpose statistical system, then it is problematic because it is not reproducible as it relies on idiosyncrasies in SAS version 9.2. Of course, an author can disclose the idiosyncratic dependencies; while this seems to be an onerous obligation, conversely it is

arguable that if an author is unaware of dependencies, then their science relying on them is equally unreliable.

It is recognized that to make critical claims, models need to be run under varying assumptions [**?**], yet somehow it is easy to overlook that the code that implements those models also needs to be carefully tested under varying assumptions to uncover and fix bugs and biases, as well as to uncover unknown dependencies. Indeed, the code may be poorly written (as this paper shows it is), so the results derived from the code simply may not be reliable.

In normal scientific reporting (outside of teaching and assessing science) details of methodology are routinely glossed. A chemist does not say they cleaned their glassware. One might argue, then, that scientists need not discuss their code in detail because they know how to program and their code is correct. This argument is mistaken. Code is rarely considered a valuable part of the science to which it contributes, which creates a vicious cycle of ignoring code, leading to ignoring the critical — and non-trivial — role of correct code in science.

3.a. The siren call of over-fitting code

Poor code undermines scientific quality: poor code can generate plausible results from *any* data, including even fictional science. A temptation is that adjusting the code's results gets more attention than the theoretical basis of the code's overall faithfulness to the experimental phenomena.

In modeling terms, superficially successful computer programs tend to *over-fit* phenomena [**?**]. Instead of using code to test the models, we tinker with the code adapting it closer and closer to our prejudices. The code then apparently confirms our science, since we fitted it to our understanding.

In general, an *over-fitted model* fits a set of data closely, but contains more parameters than can be justified by the science. An over-fitted model fails to reliably predict results beyond the scope of the data it has been fitted to. Over-fitting is a well-known problem, but the point is that when code is used, over-fitting is done unconsciously by programmers adjusting the code — its parameters, its structure, its embedded data, the calculations it performs — that specifies the model. "A model over-fits if it is more complex than another model that fits equally well" [**?**], is a criterion that describes almost every program! Programmers without the discipline and experience to manage the unlimited adaptability of code debug, alter, and extend their code to make it do what they think it should do. This becomes a vicious circle as the idea of what the science is becomes driven by the code. Rather than debugging code by improving its fit to the actual science, it gets debugged and extended to fit the expectations.

The problems of over-fitting data may be visualized using a Real→Real function of one variable (figure 1). The code that generates an over-fitted curve seems to work very well: in the example shown, the over-fitted curve fits the sampled data exactly; indeed, the code used here will fit any new data exactly as well. But the code has a negligible ability to predict new data or to describe theories of the data, which is the point of modeling. The fact that over-fitted code seems to work well is deceptive.

Looking specifically at the data plotted in figure 1 if, for the sake of argument, we assume the error in the data is normally distributed then the values the over-fitted code generates outside the range of the sample are improbable. For example, the basic linear model predicts $\hat{y}(0) = 0.5$, versus the extreme value predicted by the over-fitting code, $\hat{y}(0) = -41.8$, even lies far outside the plot region shown in the figure.[3] Similar problems happen with interpolation rather than extrapolation, for instance around $x = 4$.

While over-fitting data is a well-known problem, the point for this paper is that *code* itself can easily be over-fitted. Code can of course be over-fitted in more complex ways than can be illustrated with elementary polynomials, as here. Code over-fitting is much harder to recognize because there may be no simple graph plot, like figure 1, to highlight the problems. Furthermore, almost all code is far more intricate than the two trivial polynomials used to illustrated figure 1.

Unfortunately, code in published science *is* often over-fitted, and over-fitted in a way that is very hard to scrutinize. For example, in epidemiology (which is considered in section 5.a) it is routine to use very complicated, large dynamical models parameterized with numerous social, cultural, health, demographic and geographical data. The parameterization is mixed between data files, data written explicitly into the code, and with conditionals and other structuring in the code to cover special cases. Indeed, many of the programs in the survey used comment to inactivate code, presumably indicating an unfinished tinkering approach to code development.[4]

[3] The bounds of the confidence interval illustrated in figure 1 depend on assumptions about the distribution of the data; in this example, we are *assuming*, perhaps because we know something more about the experiment, or thanks to Occam's Razor that a linear function is more likely than a high order polynomial.

[4] Of the 10 papers in the pilot survey that reported use of code

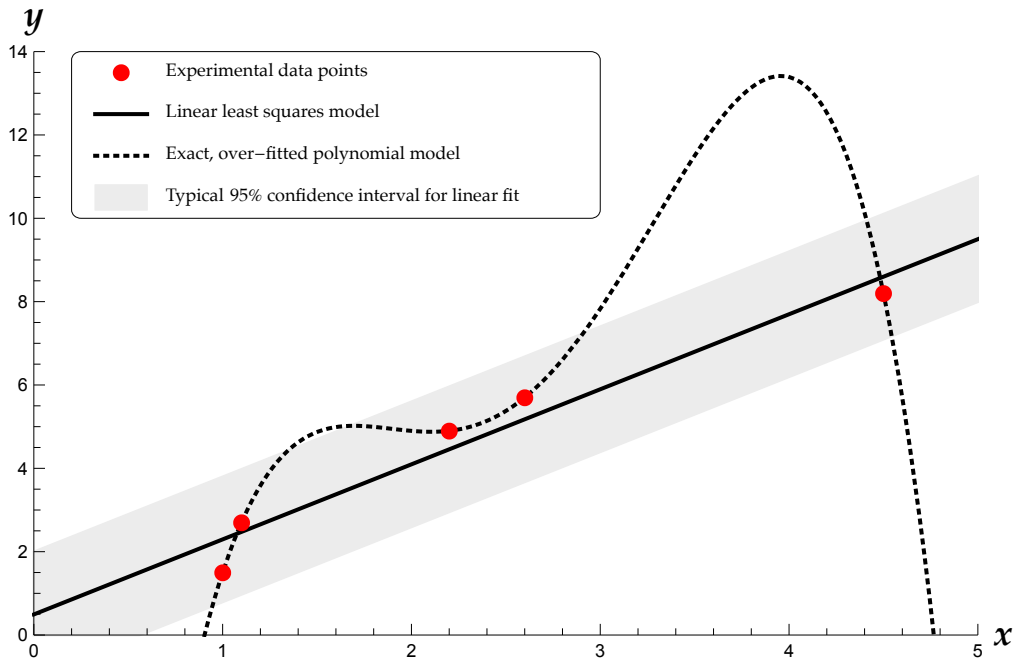

FIGURE 1. Much computational science is concerned with finding plausible multi-dimensional models that fit models to data with the aim of extrapolating or predicting new results from them. Shown here is notional sample of experimental 2D data (the dots), a linear least squares regression, and an exact polynomial model. The over-fitted polynomial model fits the sample *exactly*, but since the experimental data is presumably subject to random error (indicated by the confidence interval, itself estimated) the linear model would generally be considered a better description of the experimental data.

Furthermore, scientific support code is rarely documented well enough to know what it should have been doing, which should be answered by a specification. With no clear specification and documentation, the code can be arbitrarily hacked to get any convenient results, since no particular specification for it has been defined that it should adhere to. Thus we risk doing and promoting substandard science because we — the scientists and the publication process — are not managing the unlimited adaptability and complexity of code that science has come to rely on. This is over-fitting of the worst kind — in conventional over-fitting one can at least hope to see that the fitting is over-parameterized for the data, but in code over-fitting the code and specification are not visible, therefore not adequately scrutinized, and — worse — the "data" the code over-fits includes the entire conceptual contribution of the paper.

Reference [**?**] shows that even trivial code (in the case cited, implementing simple difference equations) with very few parameters can have very complex results, and reference [**?**] is a historically significant example how pointing out the problem of over-fitting has improved science.

repositories (covering 182 thousand lines of code — so this is not a trivial amount of programming effort), one provided an empty repository with no code at all (effectively commenting out all their code!), and seven repositories explicitly commented out chunks of workable code. The two remaining non-trivial repositories with no commented-out code consisted of straightforward, short code files with few comments of any sort.

This section (3.a) of this paper, including figure 1, was generated as a computational paper in a *Mathematica* notebook, then exported to LaTeX to include here. The Supplemental Material provides a more complete explanation, as well as the data and details of the fitted polynomials (also generated automatically). This paper's repository includes the notebook.

4. THE CONVENTIONAL ROLE OF CODE

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

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 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 is the concern of Software Engineering.

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 they are long, complex, inscrutable and (as 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 their historical thinking well enough (say, just a few weeks or months later) to maintain it correctly if it has not been clearly documented? As a scientist pursues a research career building on their previous work, how can they be certain their work is reliable, and not merely converging to their prejudices? Without proper documentation, including a reasoned case to assure that the approach taken is appropriate [**?**], 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 `k`, `x`, and `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 under- or overflow).

Without explicit links to the relevant models (typically mathematics), 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.

Some authors assert that the purpose of code is to provide insight into models, rather than precise (generally numerical) analyses summarizing data or properties of the data [**?**]. In reality, if code is inadequate, any so-called "insights" will be flawed, and flawed in unknown ways. Indeed, none of the papers sampled (described in Supplemental Material section 14) claimed their papers were using code for insight; all papers claimed, explicitly or implicitly, that their code outputs were integral to their peer-reviewed results.

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

4.a. The deceptive simplicity of code

It is a misconception that programming is easy and even children can do it [**?**]. More correctly, toy programming is easy, but mature programming is very difficult.

An analogy helps justify this key point. 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 point is obvious because we can see Lego houses, and everyone is familiar with building-block play. Its real-world engineering

limitations are too obvious to need stating.

In contrast to Lego, computer programs are generally invisible, and therefore the engineering problems within them are also made invisible. 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, and because — like Lego — toy programs can look inspiring but be unreliable, difficult to use, even dangerous.

Saying programming is easy is like 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. These are professional engineering issues that Lego builders ignore. Certainly, even real building is much easier and faster when the technical details are ignored, as anyone who has experienced a cowboy builder can attest.

Unlike building houses (the Code of Hammurabi dates to around 1755BC), programming is a new discipline, and the problems of poor programming are not widely appreciated or embedded in our culture. Professional standards, even when they exist, are not enforced.

Problems for the reliability of science arise when doodling and tweaking software drifts into claiming scientific results that do not have reliable engineering processes or structures underpinning them (let alone the properly developed and documented accessible code) to justify them.

In many countries, there are laws that require all but the very simplest building structures to be formally approved from plans and inspected as they are built, but who writes plans for software, who inspects scientific models while they are being coded? Yet the consequences of building a shoddy garage have negligible impact compared to the consequences of writing poor code that informs national public health policies or climate change interventions.

4.b. The low status of coding

Since programming appears to be so easy, developing code has a correspondingly low status in scientific practice (and more widely). 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 contributors to it. Because it is apparently easy, there is no need to work hard to make it correct. Because of the ease of over-fitting (section 3.a), code "works well" with little skill or effort. While such mistaken views prevail, the vicious cycle is that the low status means software development is casualized, which reinforces the low status.

Almost all scientific papers *routinely* describe their experimental method, their data handling, and provide an overview of their analytic (usually statistical) methods. If they are theoretical papers, they will describe their mathematical models and data that is used to run or test their models. Outside of pure computer science, scientific papers are almost entirely silent on the code they rely on and how it was developed — in particular, how the code might have been protected from bugs, analogously to how appropriate experimental methods were used avoid or control for experimental error.

In reaction to this vicious cycle, there is a growing movement to cite code correctly [**?**, **?**], because code *is* important, particularly for reproduction, testing and extension of any scientific work. Curiously, the importance of *correct* code is rarely emphasized.

Few journals editorial policies recognize 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 should also apply at least as strictly to code.

4.c. The critical role of code is often ignored

Because statistics, like code, is so readily susceptible to uncontrolled bias and error, there are many protocols and journal policies that enforce best practice, for example journals often require adherence to PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) [**?**] for any paper performing a systematic review of the literature. Yet PRISMA, like many such policies, ignores the critical role of code, and ignores the Software Engineering principles that assure code that research relies on is reliable and reliably reported.

PRISMA "was designed to help systematic reviewers transparently report why the review was done, what the authors did, and what they found," which is reasonable enough. PRISMA covers the review process carefully. For example, the authors should report the number of papers they included in their review. Perhaps $N = 2000$. This number is then written into their draft paper, perhaps in several places. As the authors read and revise their paper, submit it, and respond to peer-reviewers, it is very likely that the number of papers in the survey changes, or other numbers and details may

change.

The authors now have a maintenance problem: where are the numbers that have changed, and what should they be changed to? Doing a search-and-replace, whether automated or by hand, is fraught with difficulties. What happens if 2000 is used for some other purposes as well? What happens if some of the 2000 values are written as 2,000 or as 2000.0? What happens if some 2000 are year dates and are changed incorrectly? Then there are the Human Factors: slips and errors will happen in this process anyway [**?**]. Typos, slips during cut-and-paste, and other errors are common. Similar iterative revision cycles happen with any paper, not just with systematic reviews. PRISMA rightly deals with a wide range of methodological problems, but ignores the methodological problems (such as the example earlier in this paragraph) that using code naïvely raises.

The irony is that PRISMA says nothing about how to ensure the results of a survey are correctly and reliably presented in a paper, despite this being one of PRISMA's explicit motivations. Such rule schemes reinforce the fallacy that code is trivial and unimportant. PRISMA consciously warns about methodological issues, but ignores that poorly managed code raises comparable methodological issues that undermine the validity of the research it supports.

4.d. Bugs, code and programming

Critiques of data and model assumptions are increasingly common [**?**, **?**] but program code is rarely mentioned. Yet data and program are formally equivalent (see Supplemental Material, section 12). Program code has as great an effect on results as the data; in fact, without code, the data would be uninterpreted and probably useless. Code, however, is harder to scrutinize, which means that errors in code have subtle, often unnoticed effects on results.

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 embedded in the code. Such issues emphasize 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 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. Contracts and assertions are essential defensive programming technique that block compilation or block execution with incorrect results; they turn bugs into safe termination, or, better, failure to compile. None of the science surveyed in this paper includes any such basic techniques.

Random numbers are widely used in computational science (and in many of the papers surveyed), for simulation or for randomizing experiments. Misuse of random numbers (e.g., using standard libraries without testing them) is a very common cause of bugs naïve bugs [**?**].

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 [**?**, **?**] — they are bugs that were intended but were ideas based on mistaken 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, but are bugs in the wider sense of producing incorrect results (e.g., ill-conditioning); that is, what was intended was wrong.

4.e. Long-term problems of unreliable code

Scientists explore and extend the boundaries of rigorous knowledge. Put briefly, the purpose of scientific experiments is to vary details to either test and specify the boundaries of theories, or to discover new phenomena that then lead to theory revision.

If poor code, or poorly documented code, is made available with scientific papers, the code is a natural place to start replicating and varying experimental conditions, including both data or code. However, if the starting point is not accurately known, whether due to bugs, obscure code, or because of poor documentation, then experimental variations will have an unknown effect. Theory will then be driven by artifacts of the code, not genuine phenomena.

In section 5.a, below, an example is documented of a research code development process of at least 15 years' duration where the code was admitted to be completely undocumented, leaving details in just one author's head. None of the various related papers describe any controls over the drift of the science, or how independent researchers building on it might have been able to build with confidence rather than merely reproducing the same errors.

Since the code in question was substantial and non-trivial, it is very unlikely that any constructive reproduction occurred outside the original laboratory and

mindset; indeed, section 5.b describes how "reproduction" became trivialized because of community pressure to confirm the insights of this particular research.

Trying to constructively refute aspects of this research in the Popperian sense [**?**] would have been impossible. For example, had the relevant papers published critical code invariants then scientists building on the research could have explored whether those invariants remained valid and, if so, under what assumptions. In fact, invariants are the theories of code, and deserve as high a prominence in published science as the domain theories the code itself is supporting investigating.

5. STATE OF THE ART

5.a. Case study: Pandemic modeling

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

A systematic review [**?**] 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 [**?**]. The review [**?**] itself ignored the mapping from models to their implementation, yet if code is unreliable, the model *cannot* be reliably used, and cannot be reliably interpreted regardless of whether TRIPOD guidelines are followed. Indeed, TRIPOD guidelines ignore code completely.

It should be noted that flowcharts, which the review [**?**] did consider, are programs intended for human use. Flowcharts, too, should be designed as carefully as code, for exactly the same reasons.

A high-profile 2020 COVID-19 model [**?**, **?**] uses a modified 2005 computer program [**?**, **?**] originally developed for modeling H5N1 in Thailand, when 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 [**?**, **?**, **?**] none of which provide details of their code.

A co-author disclosed [**?**] 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 …" [**?**]

This comment was made by a respected, influential world-leading scientist, with many peer-reviewed publications, and a respectable h-index[5] of 93. Ferguson should be well aware of the standards of coding used in his own field. Arguably, then, this comment quoted above is representative of the culture of the field.

Ferguson's admission is tantamount to saying that the published scientific findings are and need not be reproducible.[6]

Lack of reproducibility is problematic, especially as the code would have required many non-trivial modifications to update it for COVID-19 with its different assumptions; moreover, the code would have had to have been updated very rapidly in response to the urgent COVID-19 crisis.

If Ferguson's 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 and repurposed 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, like many popular programming languages, not a dependable language; to develop reliable code in C requires professional tools and skills. Some of the code was written in a naïve style (e.g., writing `*(a + i)` instead of `a[i]`, and with obscure numerical goto statements like `if(l == 0) goto S150`), and with C code that was translated simplistically from FORTRAN and Pascal code, from references dating from the 1970s and 1980s [e.g., **?**, **?**].

[5] 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.)

[6] A constructive discussion of Software Engineering approaches to reproducibility can be found in [**?**].

Moreover, C code is not portable, which limits making it available for other scientists to use reliably: C notoriously gets different results with different compliers, libraries, and hardware. In fact, in any area where reliable programming is required in a C-like language, a special dialect such as MISRA C should be used: MISRA C manages the serious design flaws of C that otherwise make it too unreliable [**?**]. Alternatively, a high integrity programming language, unrelated to C, such as SPARK Ada [**?**], or modern languages (many related to the "ML family") like OCaml, F*, Haskell [**?**]. These languages have learning curves; however their key benefit is that correct programs are far more likely and are *much* faster to write. (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" [**?**]

This claim is implausible.

The original work on theoretical epidemiology may be fine if it does not use any of his code, but if the science is not supported by code that correctly implements the models, then the program's output cannot be relied on without independent evidence. Over the fifteen plus years the code was in development the science it informs will have developed too, as will the relevant data; it is not clear how they will have remained in alignment.

Typically, models will be developed iteratively as their results are improved to better fit a scientist's goals — but this, especially when it is done by tinkering, as here — risks making the code arbitrarily fit the goals (that is, over-fitting; see section 3.a), rather than to objectively elucidate the science.

In fact, the Ferguson code, `covid-sim`, is very large at 25 kLOC (thousands of lines of code),[7] so it is implausible that the "mathematics or science" has been correctly implemented in it. Ferguson's *reported* science unlikely to be reliable.

5.b. Concerns with reproducibility

Getting science right, which now, in turn, depends on correct code, is a normal requirement of *reproducibility*.

The code in [**?**, **?**] has been "reproduced," as reported in *Nature* [**?**, **?**], but this so-called reproduction merely confirmed that the code could be run again and produced comparable results (compared using an Excel spreadsheet). This weak test would pass provided the runs gave the same invalid answers — it is not usefully a stronger test than just checking that the code compiles.

Running code to obtain results claimed in a paper is a very weak test, and anyway one that should have been checked automatically during paper preparation and submission. If you do not know what you are reproducing, as is the case in this *Nature* paper, there is little scientific value in doing so.

Unfortunately, the terms reproducibility, replicability, and repeatability, have similar meanings in English but have been used in different specific technical ways by different authors. In [**?**, **?**] the reproduction amounted to just re-running the original code. It is certainly essential to establish that a paper's code can be run, as non-working code cannot support any claims in a paper; if the original code runs this confirms a basic level of access for the wider scientific community. But a more realistic criterion than basic reproduction in this sense is whether an *independently* developed model developed from the same paper(s) specifications produces equivalent results (called N-version programming, a standard Software Engineering practice [**?**]) like public health surely requires as, indeed, Ferguson's own influenza paper [**?**] argues. However, much stronger scrutiny of code than "reproduction" is required to answer essential questions including:

1. Is the code valid: does it do what the paper claims?[8]
2. Do other scientists, including reviewers and the authors, understand the code?
3. Does the code implement the methods described in the paper?
4. Has the code been over-fitted or tweaked to support specific claims in the paper?'
5. Is there a definitive version of code?
6. Is the code controlled and signed?
7. What limitations does the code have?
8. Was the code developed to any standard, and does it comply to that standard?
9. How does the code protect against data, coding, and human error?
10. Was the code tested adequately?
11. Does the code depend on arbitrary parameters, data, or code to over-fit to obtain the published

[7] Ferguson's `covid-sim` system is composed of 229 files, and uses 734 Mb of data. It is now rewritten from C into C++ with Python, R, sh, YAML/JSON, etc. For more details, see Supplemental Material.

[8] Of course, the underlying science may be wrong to, so it is useful to distinguish *internal validity* and *external validity*. Internal validity occurs if the code does what the paper claims; external validity occurs if the code represents correct science (which the paper may have interpreted incorrectly.

results?
12. Is the code documented adequately, so we know what it is trying to do, and how?
13. … and so forth — questions are numbered for reference.

All such questions also apply to specifications, documentation, assurance cases, test procedures, and other essential documents, not just to code. In turn, the levels of scrutiny demanded should be guided by explicit claims in the paper [**?**] — for example, a pilot study requires weaker assurance than code that is developed concerning nuclear power, driverless vehicles, public health, etc.

The questions in the list above are certainly hard to answer for all but the briefest code, but corresponding levels of quality assurance are demanded for other methodologies [**?**, **?**, **?**, **?**, **?**, **?**], such as data preparation and statistics to support claims in peer-reviewed science.

Because of the recognized importance of the Ferguson paper, a project started to document its code [**?**].[9] Documenting code in hindsight, even if done rigorously, may describe what it does, *including* its bugs, but it is unlikely to explain what it was originally intended to have done. 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 will be different.

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, since so many years' opportunity to improve the code have been lost. The revised code has now been published, and it too has been heavily criticized [e.g., **?**], supporting the concerns expressed in the present paper.

Some papers [e.g., **?**] 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 instead of making actual code available is worse than not publishing code at all (see [**?**]). Pseudo-code is too vague to help anyone scrutinize a model; moreover, pseudo-code may mask over-fitting in code used that is not explicit in the pseudo-code.

An extensive criticism of pseudo-code, and discussion of tools for reliable publication of code can be found elsewhere [**?**].

The Supplemental Material provides further discussion of reproducibility.

5.c. Beyond pandemic modeling

Epidemiology has a high profile because of the COVID pandemic, but the problems of unreliable code are not limited to COVID-19 modeling papers, which, understandably, were perhaps rushed into publication. But other examples that were not rushed include a 2009 paper reporting a model of H5N1 pandemic mitigation strategies [**?**], which provides no details of its code. Its supplementary material, which might have provided code, no longer exists.

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 that will have an impact long beyond the COVID pandemic.

A short 2022 summary of typical problems of Software Engineering impacting science appears in *Nature* [**?**], 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 [**?**] 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 promote the use of computational modeling [**?**], arguing that through writing code, one debugs scientific thinking. Psychology, their 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 [**?**]. 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 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

[9] The system is now open-source, available at URL `github.com/mrc-ide/covid-sim` version (19 July 2021).

precisely!

For instance, 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 [**?**]. Furthermore, any paper using ACT-R would require additional code on top of the basic ACT-R framework.

The psychology paper [**?**] presents an example computational model from scratch to illustrate a 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 modeled. 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; Software Engineering is ignored. The program listed in the paper certainly runs without obvious problems (ignoring some typographical errors due to the journal's publishers), but ironically the Python does *not* implement the mathematical specification explicitly provided for it, thus 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. Otherwise, code based on inadequate Software Engineering will 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 [**?**, **?**, **?**] 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 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 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 [**?**]. 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 ideas, unable to distinguish genuine insights from artifacts of bugs. In addition, and perhaps more widely recognized, code — especially but not exclusively mathematical code — is subject to numerical errors [**?**]. It is therefore inevitable that typical modeling code has many bugs (reference [**?**] 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 the researchers have done and whether that corresponds closely to what they are reporting.

Some COVID-19 papers [e.g., **?**] make unfinished, incomplete code available. While some [e.g., **?**, **?**] 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. Contracts, for instance, originated in work in the 1960s [**?**], and are now well-established practice in reliable programming.

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, e.g., [**?**]) 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 by the paper's authors and reviewers, will have affected the results they report [**?**].

5.d. The narrow 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 $28 billion a year is spent on preclinical research that is not reproducible [**?**].

Curating data is taken seriously as a part of normal science and peer-reviewed publication. Journal policies widely require appropriate discussion of data, much

as they require appropriate discussion of statistics. Journals often require archiving data in standard formats so it can be accessed for reproduction in further scientific work.

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) for scientific data management and stewardship [**?**, **?**], and in the development of journal and national funder policies. For example, the 2022 update to the US National Institutes of Health data policies [**?**] is described as a "seismic mandate" by *Nature* [**?**] in its attempt to improve reproducibility and open science yet they ignored code.

These cost estimates and initiatives under-play the role of code as a critical component despite its becoming 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 intended to be abstract) cannot make a quantifiable contribution to the literature. 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 programming systems may produce different results when used on different computers.

In general, without proper management of code — for example to record, detect and report version control differences — sharing code may even be counterproductive.[10]

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 (see Supplemental Material) suggests that, generally, some but rarely all develop code that is uploaded to a repository just before submitting the paper in order to "go through the motions." In the surveyed papers there is no evidence (before, during, or after the date of the survey sample) that any published code was prepared or maintained *using* repositories. This is consistent with the finished code being uploaded to a repository just for the purposes of satisfying publishing requirements, but not using one earlier probably because they did not understand the benefits of doing so.

[10]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 at least when they start their work, then there are problems that need investigation before relying on the reproducibility of the data.

Using a standard repository for lodging a paper's supporting code would help other scientists access the code easily, but not using the repository for developing and maintaining the code means the author of the paper misses out on many helpful features of repositories, such as version control, open-source development and review, actions and other approaches for automating development, sharing workload, and so on.

In summary, there is a lop-sided emphasis on data in science. In fact, data is useless without code, and code must be used to analyze data and can manipulate it in arbitrary ways — some ways, like renormalization and other transformations, deliberately, and some accidentally caused by bugs and unwittingly misunderstanding programming techniques. Often code is used to extrapolate data, so the code itself effectively generates more data, or the code eliminates outliers so it effectively deletes data. Data is routinely formatted in standard ways (else code could not process it), but code is generally very architecture- and version-specific, so — unless properly managed — code goes obsolete faster than data. And so on. In short: the integrity of code and its availability to scrutiny is in fact both harder and more important than the usual requirements put on data.

6. EXTENDING RAP TO RAP+

Code is usually seen as an independent set of files that are run to generate results, typically to be copied into a paper. But we can go further: a paper can itself embed code or become code [e.g., **?**, **?**]. This supports the generalization of RAP to form RAP+. Essentially, RAP+ is the recognition that computing is not just about programming computers (which results in RAP) but is about applying computational thinking [**?**, **?**] that supports and constructively analyzes any process, and in particular human and scientific processes that may not normally be computer-based at all.

Once processes in the pipeline are automated, this means that there is code to run the steps again. Once there is code, it can be managed in a version control system. A version control system then provides an audit trail for free, as well as many advantages such as being able to backtrack to an earlier version to review earlier edits. Importantly, code can also perform sanity checks on the process — a very simple example is automatic bibliography systems that check that journal names, DOIs are correct, and so forth. (They also allow the bibliographic data to be pooled and curated with other scientists, which improves its scope and quality.)

GitHub is a tool that provides *actions*, which are named specifications to run RAP's analytic pipelines,

or workflows in GitHub's terms. GitHub happens to specify actions in the language YAML, which, being a textual notation, in turn means that all the helpful features of GitHub — open-source, version control, and so on — can be applied to the pipelined processes as well. Research pipelines can thus be made explicit, documented, shared and improved with open collaboration.

Almost the entire scientific process can be automated (even with its interaction with the world automated by sensors and robots). There are many ways to do this; for example, *Mathematica* makes the analysis of the data and the calculations and the paper "the same thing" in its integrated notebook user interface. The many alternatives include R Markdown, an approach based in the open mathematical system R [**?**]; a system, Lepton [**?**], which allows a LaTeX document to execute and include arbitrary code, and language-independent notebook systems like Jupyter; and so on. There are also variations of literate programming, notably relit [**?**], that enable papers themselves to generate the code they rely on and may describe, which therefore ensures the code run to support papers is exactly the code the author ran to support the claims.

In all such systems, running the computational paper creates the publication. Indeed, every time the paper is run, the authors are likely to check the results after they have been re-computed, so the RAP process actively helps reduce errors.

Here is the insight: the code, just like the paper is text, so the code *itself* can fully form part of the pipeline, made reproducible and benefit from all the RAP benefits. To date, this critical point has been overlooked. Since using RAP to improve code, rather than just the pure scientific pipeline, has not been mentioned previously, we call it RAP+ to make it clear this is a new and important generalization. RAP+ helps improve code quality, for the same reasons RAP improves the quality of the paper and the science.

Software engineers have many tools for automatic code development (such as Unix's `make`) but the idea that these tools can be used to integrate and help automate code authoring as well as its documentation *and* paper authoring is radical. Amongst other things, it means that the *entire* research and development process of the paper *including* all its underlying code can be reproduced and reused by others. The present paper is a modest example of RAP+; more details are given in the Supplemental Material.

Note that as RAP+ objectifies how science is done to a standard sufficient to enable a computer to run it. RAP+ enables all of the methodologies of Software Engineering to be brought to bear *on the science itself*. RAP+ means the normally tacit, manual, and undocumented processes of science become explicit code. Code can then be scrutinized, optimized, and ensured correct by standard Software Engineering practice — thus RAP+ does not just automate science for reproduction, it makes the automation explicit so the doing of science itself can be reasoned about — not just mentally but supported by sophisticated tools, such as theorem proving and AI. Science will be improved.

7. THE PAPER AS A SCIENTIFIC LABORATORY

The conventional view of science is that experiments are done *then* written up. However, it is more productive to think of the paper itself as an active laboratory, not just as a record of finished work. Viewing the paper itself as a laboratory encourages authors to copy and adopt laboratory best practice (such as keeping records, as RAP and RAP+ suggest) into the preparation of the paper itself; it also encourages authors to see writing as a scientifically — not just expressive — creative act, and not just as the final summary of a period of scientific creativity that is not written down. In short, seen as a laboratory, writing the paper is no longer a reactive write-up of finished work, but it is an active part of doing the science itself. Writing a paper thus explores the space of scientific possibility as constructively as working in the field or on a bench: the paper is a laboratory.

With a computational paper, authors can literally experiment *in* the paper, exploring the effectiveness of ideas and explanations. Furthermore, they can experiment with hypotheses: for example, they can make a clear claim that at that point in the lifecycle of the paper is but a wish rather than an established fact — sketching a direction they plan to go in and develop supporting arguments and evidence for. So they then do the experiments or calculations (including consulting other scientists and peer reviewers) to establish a justification and other details. Typically, the evidence they generate or the criticism they receive may not be quite what they expected, so they then revise the claim to be correct, or change it to be a more realistic claim still as yet to be fully substantiated, or they could delete it if it just turns out to be a mess, or they could develop some altogether much better approach to the work as a result of the exploration.

Typically, many ideas in the paper will be linked to data from conventional experiments. For example a computational paper might calculate that the statistical

power of an experiment is too low (there is an unacceptable risk of committing Type II errors), so the authors will decide to improve the experiments.

If there is code in the paper, every time it is typeset, the code will be run. Therefore the authors of the paper proof-reading the results of the code will have opportunities to try to debug and improve the code. Again, the paper itself is acts like a laboratory, helping the authors refine the science.

Note that systems capable of handling computational papers (including LaTeX) can create conditional documents: for example, there could be a flag `publish`. If `publish` is false, the author can see all their private work and thinking, including all their experimental thinking and workings; but if `publish` is true, the paper would be typeset as for a submitted paper with the detailed working concealed to ensure a clean and concise presentation. In principle, also submitting to the journal with `publish` false (hence with data and workings visible) could satisfy journals' code and data requirements. Of course, when there is a large body of data or code, they need not all be an explicit in the computational paper: they would be made available separately in the usual way.

The more we view the paper as a proactive scientific laboratory, the more we gain from the RAP+ perspective, and the more science gains from improved, conceptually broadened, reliable, and reproducible science. The more we will want to engage mature Software Engineering standards too, because the quality and creativity of future science relies on them.

8. 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 [**?**] — and for the same reasons: computer bugs accidentally cultured in one laboratory can infect research and policy worldwide.

Inadequate code can be extremely problematic. Incorrect results might be used for supporting science, modeling COVID-19, informing public health policy, medical research, or used in other *critical software*. Professional critical software development, as used in critical industries such as aviation and the nuclear power, is (put briefly) based on *correct by construction* [**?**], effectively: design it right first time, supported by numerous rigorous techniques, such as Formal Methods, to manage error. Not coincidentally, such are *exactly* the right methods to ensure code is both dependable and scrutable, as required for supporting quality science. Conversely, not following these practices undermines the rigor of science.

An analogous situation arises in ethics.

Misuse of data, exploiting vulnerable people, and not obtaining informed consent are typical ethical problems. Planned research may be 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. National funders, and others, therefore require Ethics Boards to formally review ethical quality. Medical journals will not publish research that has not undergone appropriate formal ethical review.

Analogously, and supplementing Ethics Boards, it is proposed that Software Engineering Boards would authorize as well as provide advice to guide the implementation of high-quality Software Engineering to support research and publication processes. Just as journals require conflicts of interest statements, data availability statements, and ethics board clearance, we should move to all scientific papers and funded research being required to include Software Engineering Board support and clearance statements as appropriate. Note that Software Engineers themselves have a code of ethics that applies to *their* work [**?**].

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 12 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 earlier. And the solution is the same: relevant experts — whether statisticians or Software Engineers — need to be routinely available and engaged with the science. Software Engineering Boards 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 [**?**].

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; indeed, few people consider Ethics Boards to be disproportionately onerous. 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 [**?**, **?**] 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. They meet Tony Hoare's comments about the value of rigorous management of procedures [**?**], and widen them to non-programmer scientists. Methodological suggestions already made in the literature include open-source and specific Software Engineering methodologies to improve reproducibility [**?**, **?**]. Reference [**?**] provides an conceptual framework. However, there is scope for further research to provide an evidence base to motivate and assess appropriate interventions (such as those proposed in this paper) to help scientists do more rigorous and effective Software Engineering to support their research and publishing.

An analogous proposal to SEBs has been made for Methods Review Boards [**?**], to help scientists ensure the methods they use are appropriate for addressing their research questions. The Methods Review Boards was motivated by an Ethics Board member noticing that often experimental methodologies are inadequate, and will waste time that cannot be corrected until the flaws are spotted too late during peer-review. The paper [**?**] raises many of the same trade-offs that SEBs also face; indeed one would hope that Methods Review Boards would include Software Engineers or have SEBs as sub-boards — as this paper has argued, Software Engineering is now a key methodology of science. As with SEBs, the goal is not to gatekeep, but to improve.

8.a. 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, such as TOP and Methods Review Boards.

Relying on 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 (sections 4.a & 4.c) 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* [**?**, **?**], 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 how that needs refining or optimizing. In other words, the Software Engineers can bring important insights 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 it's ill-conditioned, there is no good answer to that question." Ideally, then, it is not a simple sequential process

but an iterative cycle of collaboration and growing understanding, informed by Software Engineering best practice (via SEBs), and implemented using papers as laboratories.

In short, the way SEBs work and are used will be crucial to the success of the science they support. 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.

8.b. Methodological statements

Many science journals, conferences, workshops, videos, books, etc, and funders require explicit statements how the authors have conformed to appropriate methodological standards covering issues such as conflicts of interest, ethics, data access, consent, authoring, funders and other acknowledgements of support, and so forth. Conformance to PRISMA, discussed above in section 4.c, is one such methodological statement; like many journals, the high-profile journal *PLOS ONE* has a list of similar methodologies it expects authors to comply with as appropriate.

It would be easy for journals and funders to require equivalent types of statements on the quality of code, that is on the quality of its Software Engineering.

Studies of data access statements show that they are unreliable: some authors withdraw papers when journals request access statements [**?**] (indicating that journals that do not make an explicit request are likely publishing papers that will not provide access), and some authors do not respond to access requests [**?**]. How we help scientists who do not want to provide data or code access is one problem, but the serious issue for science is how to ensure any statements made are accurate, and any access provided is actually helpful (e.g., well-defined, versioned, etc) to support useful reproduction.

Journals and funders should provide support for hosting data and code (and any other relevant data, such as qualitative data, video, etc), and the review process must check that authors actually provide the material as they claim in the methodological and access statements. Conversely, scientists should be able to access funding to ensure data and code access, and, as appropriate, funding for on-going maintenance of the databases, which will typically require funding beyond the end of the normal funding period.

Intellectual Property (IP) is an increasing concern, both for author scientists and their sponsors who want royalties, and for other scientists wishing to freely build on the published science. Particularly concerning access to code, IP is potentially and often is in conflict with scientific openness. Methodological statements should be made concerning any IP associated with code, and to what extent this interferes with open access to code. More routine discussion should include raising known system dependencies, such as operating system, compiler, or special hardware dependencies; it is also appropriate to mention standards conformance, such as to IEEE Floating-Point Arithmetic (IEEE 754).

Journal policies could start to explicitly encourage computationally reproducible science using RAP and RAP+ techniques. That is, the research's methodology itself may be a mixture of data and code. As this paper's Supplemental Material shows in section 14, many journals (e.g., *PLOS ONE* and, ironically, *IEEE Transactions on Software Engineering*) and repositories have policies that make RAP much harder or just counter-productive at the last step.

Methodological statements should be required that make clear what access rights are available for RAP or RAP+ material, as it is much more likely to raise IP issues that normal disclosures. In particular, if the authors plan on publishing a series of papers based on the same methodologies, the RAP/RAP+ access might be provided in a later paper or held under escrow by the journal or funding body.

Journals and funders often require data and code access statements, but as this paper has made clear, code is complex and it is rarely easy to understand and scrutinize even with access to substantial documentation (which is unusual). It is therefore recommended that journals and funders require *assurance arguments* [**?**], a familiar technique from the safety assurance domain. Assurance arguments provide a concise, high-level argument that the system does what is claimed. Assurance arguments can be more or less detailed, and more or less formal in their approach; we envisage referees would have views on the level of detail and formality required for any specific contribution.

Finally, as there is no practical distinction between data and code (see Supplemental Material) and methodology (thanks to RAP), and certainly no distinctions that cannot be circumvented, journal and funder policies of code and data access should be reviewed *and unified* so that the access statements apply to all information, regardless of any arbitrary classification of it as code or data (or documentation,

assurance case, etc).

8.c. Benefits beyond science

Science increasingly recognizes the key supporting roles of code and computation, but many fields do not recognize computation as such, as a skilled discipline, and therefore they are missing out on the leverage that comes with recognizing computation as a first class player in their activities.

Healthcare is supported by computers, yet medical research papers remain traditional and do not refer to code. Yet clinical practice relies on computer analysis, so inevitably must use code unrelated to the code developed in research, which is unlikely to be developed to Software Engineering standards. Conversely, the critical issues (including patient safety) that assume code is more reliable than that needed for scientific research do not get evaluated by researchers. The gap is wide. The problems and missed opportunities of under-valued and poorly-managed code are ubiquitous in healthcare [**?**]. Lasting solutions might well be initiated through SEBs or equivalent.

Numerous problems in finance have been facilitated if not precipitated by computational naïvety. JP Morgan Chase (JPM) lost over $6 billion in a credit derivatives trade [**?**], in a costly parody of bad science — compare the discussion here with figure 2, in a series of figures in the Supplemental Material, which illustrates the same problem as it presents in many scientific papers.

As reported in [**?**], the traders did not understand the trades, did not monitor them, doubled down when results were poor, and did not communicate the extent of their losses. They were using manual coding methods; as the report says:

> "[…] the model operated through a series of Excel spreadsheets, which had to be completed manually, **by a process of copying and pasting data from one spreadsheet to another**. [Our emphasis.]
>
> […] this individual immediately made certain adjustments to formulas in the spreadsheets he used. These changes, which were not subject to an appropriate vetting process, inadvertently introduced two calculation errors
>
> […] after subtracting the old rate from the new rate, the spreadsheet divided by their sum instead of their average, as the modeler had intended."
>
> etc [**?**]

The report does not detail how the Excel spreadsheets were specified or coded, seemingly as unawqre of Software Engineering as the traders. However, one infers from the brief discussion of data handling (which is easier to automate) that it was all an unconsciously incompetent process.

Reviewers in JPM failed to scrutinize not just the coding, but the trades informed by the code. They passed on optimistic reports. Then there was a merry-go-round of blame: "the information communicated to the Risk Policy Committee … did not suggest any significant problems … there was no robust debate with the right facts at the right level about the portfolio risk." UK and US governments are now investigating fraud. Again, lasting solutions might well be initiated through SEBs or equivalent.

Without taking the lessons of improving Software Engineering to other fields, including improving and broadening the recognition and career paths for developers, there will continue to be an unfortunate and unnecessary disconnect between competent coding and actual practice. Everything, from healthcare to finance — not just science — will suffer because the critical contributions of dependable code, quality Software Engineering, and competent Computational Thinking are not recognized, understood, valued, or required.

9. CONCLUSIONS

In a pandemic, epidemiological modeling (discussed in this paper), track and trace [**?**], modeling mutation pressures against vaccine shortages and delays between vaccinations [**?**], and so on, drive public policy and have a direct impact, whether positively or negatively, on the quality of life.

While this paper was originally motivated by Ferguson's public statements [e.g., **?**, **?**] about his high-profile COVID modeling, the wider evidence reviewed here suggests that scientific coding practice is inadequate in every field. As science becomes more and more reliant on computers, we need to correspondingly improve the quality of code, the quality of code policies, the quality of Software Engineering, and the quality of all scientists' understanding of computation and how to manage its unlimited complexity.

The main challenges to mature computationally-realistic science are:

1. To manage software development to reduce the unnoticed and unknown impacts of bugs and poor programming practices that research and publications rely on. Computer code should be explicit, accessible (well-structured, etc), and

adequately documented. Papers should be explicit on their software methodologies, limitations and weaknesses, just as Whitty expressed more generally about the standards of science [**?**]. Professional software methodologies should not be ignored.

2. To use computation to help make scientific processes explicit, so that they can be reproduced, scrutinized and improved. RAP is an increasingly popular way to help do some of this, but as this paper points out, RAP should be generalized to RAP+ to help the computational parts of science as well, leading to a virtuous circle.
3. To support and develop the scientific community in the professional use of computation.
4. To find effective ways to promote professional software engineers being recognized and participating fully in scientific research, like professional statisticians routinely support quality research.

While programming seems easy, and is often taken for granted and done casually, programming *well* is very difficult [**?**]. We know from software research that 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 magically make it more reliable [**?**, **?**, **?**].

Software Engineering Boards, as proposed in this paper, are an initial, straightforward, constructive, and practical way to support and improve computer-based science.

This paper's Supplemental Material summarizes relevant Software Engineering good practice that Software Engineering Boards would draw on, 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, Konrad Hinsen, 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, Isaac Thimbleby, Prue Thimbleby, Will Thimbleby, Martyn Thomas, and Ben Wilson. The author also thanks the anonymous *Computer Journal* referees who also contributed to the quality of this paper.

Data and code access There is an extended discussion of the methodology of this paper in the Supplemental Material, section 12. The Supplemental Material also presents all raw data in tabular form. All material is also available for download at URL `github.com/haroldthimbleby/Software-Enginering-Boards`,[2] 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 classes 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.

In addition, a standard CSV file is generated from the JSON in case this is more convenient, for instance to browse directly as a spreadsheet or to import easily into other programs.

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 [**?**, **?**].

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], by the Royal Academy of Engineering through the Engineering X Pandemic Preparedness Programme [grant EXPP2021\1\186],

[2] This is a temporary URL before meeting publication repository requirements for accepted papers.

and by Assuring Autonomy International Programme, Assuring Safe AI in Ambulance Service Triage. The funders had no involvement in the research or in this paper.