A tool for publishing reproducible algorithms & A reproducible, elegant algorithm for sequential experiments

Harold Thimbleby[a,*], Dave Williams[a]

[a]*Swansea University, Wales, SA2 8PP*

Abstract

Tools to ease the burden of reproducibility are important so computer science does not fall into the trap of "cargo cult" science: particularly publishing discussions of algorithms that look like algorithms but which do not work properly when they are copied from the paper.

This paper introduces a tool, called **relit**, which makes it very easy to write about and publish correct algorithms, yet without restricting the author's style. In fact, **relit** can be used with any material: mathematics, proofs, algorithms or programs. It can be used in papers, in reports and books and, with analogous advantages, in student work — where examiners may wish to automatically check what the student claims to have written is actually correct.

To demonstrate **relit**, this paper presents a new, elegant algorithm for the design of sequential experiments to efficiently control bias, drift, random error, carry-over and other effects. The algorithm is written in C, in a clear style to simplify porting to other languages.

We developed **relit** because it was impossible to find simple reproducible code for this problem, and we wanted to do better. Thanks to **relit**, the published algorithm is reproducible and works *exactly* as published in the present paper. This paper also includes discussion of the problems and opportunities of reproducibility and the essential contributions of **relit**-style approaches to improving the reliability of computer science publications.

Keywords: Reproducibility, Publishing algorithms, Literate programming, Euler cycle algorithm, de Bruijn sequence, Combinatorics, Experimental design

*Corresponding author

Email addresses: harold@thimbleby.net (Harold Thimbleby), drdjwilliams67@gmail.com (Dave Williams)

URL: harold.thimbleby.net (Harold Thimbleby)

1. Introduction

When we needed a standard "off the shelf" algorithm for the design of sequential experiments, we found it was quicker to invent a new one from scratch than get published algorithms to work. It was hard to find a *working* solution for the problem in the literature — even when we had searched both peer reviewed publications and textbooks.

Reproducibility, or rather, lack of reproducibility is a widespread problem affecting many algorithms, especially long, complex or unusual algorithms. Often such algorithms are left as "exercises for the reader" or are merely analysed for their properties, rather than presented with correct code.

This paper introduces our elegant new algorithm for sequential experiments and its use and application for sequential experiment design (specifically, it generates $(N, 2)$ de Bruijn sequences). The algorithm is much briefer than theoretically faster algorithms that have been outlined in the literature though not so reproducibly described. The algorithm will be useful for experimental scientists (who can parameterise it for their specific research) — exactly the sort of people who need working algorithms but rarely have the time or resources to invent correct algorithms from the far-too-often inadequate descriptions in the literature.

We use our reproducible presentation of the algorithm as a full and thorough case study of a new tool-based approach for easily and rigorously explaining and documenting reproducible algorithms in published papers. A key contribution of the paper, then, is our tool-based approach to reproducibility and our discussion around reproducibility.

Finally, this paper relates our new, lightweight tool-based approach to Richard Feynman's exhortation to avoid what he calls "cargo cult science" [9]: when we have the right tools to help us, his challenge becomes easy. We show how our approach makes publishing correct working software easier and more likely. It helps improve the quality not just of publications but also of the underlying algorithms themselves: authors can now easily improve their algorithms and their papers in a tightly integrated, even enjoyable, process.

1.1. Reproducible research versus reproducible publication

Science is based on reproducibility: if an idea or theory is not reproducible, it is not science, or at least the lack of reproducibility uncovers a boundary case that refines the science. A common reason for lack of reproducibility is that a published paper does not disclose enough details or contains mistakes: what is published is inadequate for another scientist to reproduce the work. (Fraud and multiple publication are further reasons — papers may deliberately fail to disclose enough to reproduce the work.)

Algorithms and programming more generally have the interesting property that in principle everything is reproducible: programs run on computers, and programs are text

that can be fully disclosed. There has been recent increasing interest in making reproducibility a criterion for accepting papers in the field (we discuss this further in Part III).

We further make the distinction between reproducible research and reproducible papers:

Reproducible research A paper discloses enough for a reader of the paper to reproduce the research. For example, *everything* in this paper is available on GitHub, at `github.com/haroldthimbleby/relit`[1]

Some journals and conferences (such as the ACM Symposium on Principles of Programming Languages since 2015, `popl.mpi-sws.org/2015`) have started asking or requiring authors for "artifacts," documents and other evidence intended to support the scientific claims made in a paper. Papers with quality artifacts are badged, so that the additional value can be recognised by the community.

Reproducible paper A paper *as published* discloses enough for a reader to reproduce the claims of the paper, with essentially no further effort than copying the details. For example, this paper introduces a new algorithm and that algorithm is completely disclosed in this paper, in fact as *source code* in standard programming language that can be copied directly from Part I of the paper.

Semi-reproducible paper A paper *as published* discloses enough for a reader to reproduce the claims of the paper, but perhaps to reproduce it exactly will involve substantial further work. For example, this paper introduces a new approach to reproducibility that is thoroughly disclosed in this paper. However none of the code to implement the approach is disclosed in the paper itself, although the *idea* can be copied from Part II of the paper. (As it happens, the documentation and complete source code of **relit** are available on GitHub, so the research is reproducible.)

While there is an important place for non-reproducible research (for instance in inspirational writing or in papers that expound new research challenges) a serious problem is non-reproducible research masquerading as reproducible. Such research has the potential to mislead readers: it looks like it solves a problem, but it does not. Sometimes this is innocent — for example, an author publishes an algorithm because they are interested in its complexity, but a reader may want to run the algorithm to solve a problem, however the detail disclosed in the paper may be insufficient to run the algorithm. Sometimes this is an oversight — for example, some details the author knows were not disclosed, but they are nevertheless critical. Sometimes this is fraud or bad practice — for example, the author wants to achieve another publication using the additional details.

[1] Note: Elsevier may use their own GitHub URL for the "accepted for publication" version of our software.

1.2. Relit: A tool for reproducibile papers

We built a tool **relit**, so called because it implements "reverse literate programming." **Relit** helps write clearly about any algorithm or algorithms, and is intended to be particularly useful to help write reliable, reproducible papers about algorithms.

In conventional literate programming, the literate program source file drives the entire process and generates a program and structured documentation for it; in reverse literate programming the main relationship is reversed, being driven by a paper (such as the one you are currently reading) from which source code is derived. Another contrast is that conventional literate programming aims to produce a program and full internal documentation; reverse literate programming aims to produce a paper (or book or other publication) that describes algorithms or program code, at the same time providing assurance that the code is correct. Figure 1 explains the "reversed" concept further, and how it turns the conventional literate programming approach around.

Here is a small but complete, self-contained example:

The Java `for` loop `for(int i = 0; i <= 9; i++) System.out.print(i)` will print `0123456789`.

The reverse literate programming approach does not impose any formatting or style conventions on the author, so — to try and emphasise the flexibility — we typeset this example in a gray box to help make it stand out as an example. Code does not have to be syntactically complete, and can be written however best suits the author's needs; for instance, in this example, we chose to omit the final semicolon strictly required by Java. Indeed, this paper has several unrelated concrete examples in it (in several programming languages) and **relit** imposed no conventions or restrictions on the authors to achieve this.

Our tool **relit** extracted the *actual* code shown above from the LaTeX [22] source for this paper, and inserted it into a complete Java program (which also provided the hidden semicolon) that was also in the LaTeX source file. The makefile [24], also defined in this paper's LaTeX source file, compiled and ran the Java program, saving its output to a temporary file, which was input into the paragraph above. Hence the output of the program shown above really is the output of running the actual Java code shown. In fact, the complete source code for the Java program (as well as the makefile) is in the single LaTeX source file for this paper, though the rest of it is hidden from sight beyond the end of the published paper. (It appears after LaTeX's `\end{document}`, though **relit** could have extracted it from a separate file just as easily.) As an option, **relit** can summarise all the invisible (or otherwise) code to help check that nothing critical to the paper has been accidentally hidden from the reader.

As a test, the authors edited the `for` loop and the example changed as expected; this reassured us that the example above is accurate and not broken by typesetting (e.g., by some idiosyncracy in LaTeX). In other words, *both* the authors of this paper *and* therefore the readers of this paper have assurance that the code above does exactly what we say it does.

In a more critical situation than publishing code in a paper, it would be desirable to design the makefile (or other run time script) to ensure that any errors in the program result in visible warnings, perhaps no paper at all, until all errors are corrected — in our simple example, if we introduced a syntax error in the Java code, the compiler would generate no code at all, so the *last* successful executable file would be used instead. Running it would then misleadingly generate obsolete output from some earlier successful compilation. Tools like **expect** [23] can help assure that the right code is used reliability in a systematic way. However, both Java and LaTeX are powerful programming tools, so an author so intent could program any effect they want, but which might accidentally end up being misleading regardless of safeguards like **expect**; thus it is critical that tools like **relit** are very simple and elegant so that the author is not tempted into trying complex, possibly unreliable, tricks to get the effects they want.

Using **relit**, there is no notation at all to get going, and very little notation to learn to start using features as they are needed. Even when all the features are used, everything can be done with only two simple commands that are LaTeX comments, so they have no unintended interaction or side-effects on the meaning of the original document. No writing needs to be structured or re-structured to make use of it (e.g., into an outline or set of sections as in conventional literate programming [18; 30] and in systems like Mathematica [47], **org-mode** [32], etc).

Another important difference between conventional literate programming and reverse literate programming is that literate programming is motivated by documenting programs or algorithms (even if they end up being book length) whereas reverse literate programming is motivated by helping write reproducible papers (which are usually brief) about programs or algorithms. One focuses on the program, the other focuses on the publication.

We used **relit** throughout the present paper: the new algorithm we present in this paper works *exactly* as shown. The discussion about the new algorithm therefore combines three roles: its intrinsic interest as a useful algorithm, the discussion about the algorithm literature, and the algorithm as a complete worked example presented reproducibly using **relit**. Apart from our assurances that the code shown is real, there is nothing unusual in the style or format of the present paper: while giving many advantages to authors, reverse literate programming is invisible to the reader.

It is notable that with the **relit** approach we advocate, an *ordinary*, conventionally-written, paper gradually morphed into the present paper that now accurately generates the code it talks about. The original paper was written conventionally, but **relit** features were incrementally introduced to "pull out" the existing code embedded in the paper's LaTeX source document. Additionally, in the same file, but hidden from the reader as the details are not relevant to the paper, are the the full run-time resources to make the code work. In other words, the code and program results shown in this paper has been generated *from the paper itself*, and it works as described — this is reproducible research. All other related approaches to improving reproducibility of which we are aware require either some sort of special notation or structure for files, or they generate the paper from a specification. Furthermore, most impose typographic and stylistic constraints on the papers that can be published.

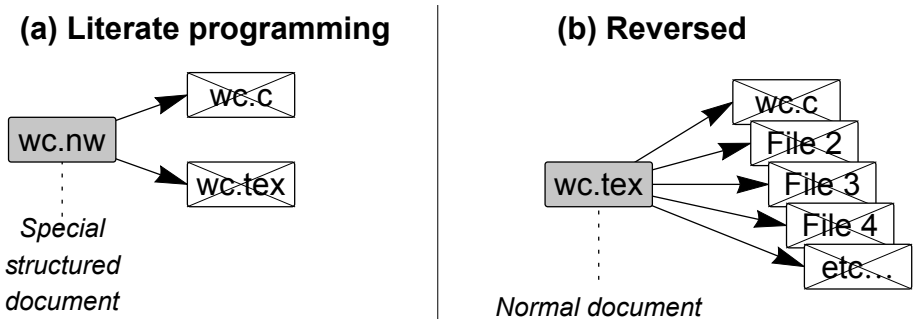


Figure 1: Conventional literate programming tools work as illustrated in Figure 1(a), which is based on a diagram in the paper on the literate programming tool **noweb** [30]: as shown, the author edits a source file (e.g., `wc.nw`) and the tool generates two files: a compilable program (`wc.c`) and a stylised LaTeX documentation file (`wc.tex`). Contrast this with Figure 1(b), which shows the comparable diagram for **relit**. With **relit**, the author edits a file (now `wc.tex`) from which **relit** can generate many files, including `wc.c`, which is compilable. Additional files generated by **relit** can be used for any purpose; for instance, in the present paper one of the generated files is a makefile so generated files were compiled and executed with their outputs inserted back into this paper.

1.2.1. Relit: Basic use

Here is an illustrative and basic development cycle using **relit**:

1. Write the initial paper *as usual*. (Of course, anticipating using **relit** one can do better than just writing an ordinary paper.)
   - In conventional literate programming, one has to start with a special "web" file. No web files are in formats suitable for conventional publication.
2. Edit the paper so program code snippets are preceded by simple **relit** commands, written as comments so they do not affect the published paper at all.
   - In conventional literate programming, the code has to be structured to make it possible to document. The structuring decisions are hard to change later.
3. Optionally, somewhere that is ignored — in another file, or beyond the LaTeX `\end{document}` — add **relit** commands to add any support code or housekeeping that is needed.
   - Conventional literate programming has no way to hide code that is distracting or needs no explanation.
4. Running **relit** now generates the specified files from the contents in the paper. Compile and run or otherwise test the generated files.
5. Concurrently edit the original paper to improve the code so it works better and edit the text of the paper so it more accurately describes the code.

    - You cannot edit the original paper in literate programming; you have to edit the structured web file that generates the paper. This is made clear in Figure 1.
6. Repeat until everything works as intended. At all times, the submitted versions can be used to automatically reconstruct the algorithms to ensure they work correctly.
7. Submit to a journal or conference.
    - If the publishers have stylistic or formatting requirements, conventional literate programming hits a wall — the paper will require substantial editing, which undermines one of the key advantages that the code should be correct and coherent with the description. Editing the submitted paper will introduce inconsistencies with the original documents, and hence will compromise reproducibility.

1.2.2. Relit: An example of its real use

At the last moment while writing this paper, our colleague Paul Cairns pointed out a worked example we had used explaining our algorithm erroneously confused wine regions (e.g., Chianti) for grape varieties (e.g., Pinotage and Merlot).

In the conventional approach to writing about algorithms, we would probably have edited the paper alone to fix the error ... and the paper and the programs we had written about would have diverged, or at least we would have needed to edit several files and do a lot of careful cross-checking — if we could be bothered to do it. We might have decided it was not worth the trouble. We might have fooled ourselves: the original program works even if it has the names mixed up; we would probably have persuaded ourselves that, surely, the program *used to* work, and then we would edit the paper alone to fix the errors. It is then but a short step to justifying to ourselves not doing the cross-checking properly because it more easily imagined than done. If we had gone down this route, the paper and the program would have been different. If we had corrected any other errors or polished — which is always very tempting — any aspect of the paper, potentially the published paper and the algorithm would have become very different. If we made any clerical slips editing the paper, the paper and the program could be critically different. Worse, neither we the authors nor the readers would be aware of the bugs and inconsistencies introduced by these well-intentioned "corrections."

Instead, because we were using **relit**, to fix the wine naming issues we did *a very trivial* edit in just *one* place in a *single* file, that is, in one place in the paper you are reading right now. With no further editing, thanks to using **relit**, we then automatically had a new C program that correctly used Shiraz, *and* the output from running the actual program was then inserted back into this paper. The example output (used in Section 2.5 below) also, of course, automatically updated to say varieties instead of regions.

Everything remained consistent with negligible effort. There was no need for any tedious or error-prone double-checking, because all the necessary edits needed were in

exactly one place — the program code that is explicit in the source text of this paper. If there is only one object, it is necessarily consistent! And, finally, everything (all the code examples and the outputs from running the code) is obviously consistent with the code published in the paper because it was generated by running exactly the code in the paper you are now reading.

In fact, doing the change was much easier than explaining how easy it was to do!

Part I

Case study & The problems of finding reproducible algorithms

2. Algorithms for sequential experiments

In a sequential experiment, such as generation of sensor calibration curves, a number of random and systematic errors may occur. Errors can include: bias, drift of sensor readings with time, and carry-over effect where the previous reading influences the next reading. Good experimental design therefore uses a sequence of calibration values arranged in such a way the sequence will normalise and cancel out random and systematic errors. Ideally the sequence will be as short as possible in order to minimise unnecessary work and expense in the collection of data.

The problem may be illustrated by a familiar example. We might want to know which of N types of wine tastes best, but, as is well known, the flavour of a wine is affected by the last wine just tasted. We therefore want to design a systematic experiment for wine tasting that tries every sequential combination of pairs of the set of N wine types available in our cellar, and of course we want the shortest such cycle, because experiments with wine are expensive.

For the sake of concreteness, suppose we have $N = 3$ types of wine in our cellar, specifically, say, Merlot, Pinotage and Shiraz. If we have just drunk Pinotage, then there are three possible experiments to do next: to drink Merlot next, Shiraz next, or of course to drink Pinotage again. If we drink Pinotage, then the next experiment should probably be to assess Merlot or Shiraz, since we already know what Pinotage after Pinotage tastes like.

With only three wines, the best sequence of experiments is not too hard to work out by hand, but in general with lots of wine it becomes much harder, especially if we start the wine tasting before we have finished working out the right sequence.

For simple cases the problem can be solved by hand by drawing a graph with one vertex for each of the N types of experiment (e.g., testing a type of wine) with N^2 arrows to represent each possible sequence of two experiments. This creates a *complete graph*, usually denoted K_N. Figure 2 (on page 14) shows the complete graph for $N = 3$. (The background in graph theory is explained in Section 2.1 below.)

A sequence of several experiments forms a path following arrows in this graph. In our problem we want to find a shortest path that follows every arrow at least once. It is a standard graph theory result for a complete graph that a shortest such path exists and only needs to follow each arrow exactly once, and it will also end up where it started — obviously, if it followed any arrow more than once, it would be repeating an experiment and take longer. An optimal sequence is called an *Euler cycle*, after Leonard Euler who first studied paths in graphs (but not sequential experiments). Our problem therefore reduces to finding an Euler cycle algorithm.

Our problem, then, is to write a computer program to generate an efficient sequence of experiments for the general case. Fortunately, finding Euler cycles is a well-known programming problem, which has been solved many times. Unfortunately, it turns out that programming textbooks typically leave this problem as an exercise for the reader, and that does not help if your aim is to do a sequential experiment, rather than learn the hard way how to program algorithms!

The definitive algorithms textbook by Cormen, Leiserson, Rivest and Stein just says:

> **Exercise 22-3** [...] Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (*Hint:* Merge edge-disjoint cycles.) [4]

Elsewhere Cormen estimates that writing up the solutions to the book's exercises would run to between 2,000 and 3,000 pages [3]: it is just not going to happen. Other books do not show answers to exercises in case students might cheat.

But worse than not providing an algorithm, in many textbooks and published papers, the Euler cycle algorithm — if presented at all — is described in high-level English, as a sketch, or in a simplifying pseudo-language. It is also easy to find numerous lecture notes online that describe the Euler cycle algorithm in pictures with English annotations. Aguably, this is all cargo cult science — they look deceptively like programs, but are not.

We will return to the general problems of reproducibility in Part III, but here we now focus on the computer science problems — and solutions.

While English may be sufficient to explain some principles or to estimate the time complexity of the algorithm or other properties, and to do other things programmers are interested in, it is completely inadequate to convert into a working program in a real, executable, language such as C or Java. To get a working program, a lot of detail needs careful consideration: how should a graph be represented in a language with type checking, for instance? If vertices are numbered, are they numbered starting from 0 or 1? Or should you use a standard package, and then have to convert the pseudo-program into the programming conventions of that package? Unfortunately, the necessary detail to get anything to work makes explanations unwieldy. In the worst case, the detail never existed, and the presented algorithm is not even an abstraction of anything that was ever executable.

Furthermore, Euler cycle programs assume the graph is arbitrary, and there are different algorithms for directed, undirected and mixed graphs. In our case, the graph happens to be both directed and complete, and we discovered that these facts can be used to

greatly simplify the problem. In fact, inventing and implementing a new algorithm for our special case took less time than correctly implementing a standard algorithm.

Our complete working program *exactly as described reproducibly in this paper* is 21 lines of code; moreover, the core algorithm is only 11 lines, including comments. (The code is given in Section 3.3.) This compares favourably to a quality reference algorithm [35] that is 22 lines of code (plus 49 lines of test code), though it will not work "out of the box" without also finding and compiling it with the various files it depends on — the total code needed runs to 493 lines, about 23 times longer.[2]

This reference code also requires further work to edit the Java and local Java environment to compile it, as well as code to define a complete graph and print the desired solution (details that are already included in our algorithm). Furthermore, while Java may be a fine language, it is hard to translate algorithms written in it into other languages; whereas our code, written in basic C, is easy to translate into any language that has arrays (Fortran, PHP, JavaScript, Mathematica, Matlab, Java itself ...), which will be a considerable advantage for experimenters who are not familiar with Java.

The Java reference algorithm is sophisticated and no doubt ideal for teaching purposes, but there is an evident separation of the published book text [34] from the published program [35], a separation that allowed the code to develop into a sophisticated program independently of the published book. Unsurprisingly, the book omits the code altogether: it has to be downloaded from the web instead. Of course, this is entirely defensible as the code available on the web has many details that go far beyond what is needed in a book — the code includes unit tests and examples, which would be tedious and distracting to explain in a book. Even when code from a working program is published, unknown details in the support code that is not published may be critical to getting it to work properly.

It is interesting to note the polarising force of positive feedback when writing about algorithms:

- When code cannot be seen by the reader, the author (as a good programmer) is under pressure to add features so it can do anything that a reader might be anticipated to want. There is thus a natural tendency to add features, which tends to make the code more complex, which further justifies keeping it out of sight ...
- When code is brief and visible, the reader can adapt what they can see themselves: the author has no pressure to add features. When code is visible, there is a natural tendency to improve and clarify it, which further justifies keeping it visible ...
- The third possibility is that the author manages the complexity by abstracting the algorithm: this makes the book or paper concise, which conceals details the

[2]Line counts are based on the actual code extracted from this paper by our tool **relit** (they were calculated by the Unix utility **wc** on the program source files and then input in the LaTeX for this paper). Line counts ignore comments and blank lines. No attempt has been made to "squash" code to save lines.

> author finds irrelevant to their immediate goals.[3] Unfortunately, as "irrelevance" is abstracted away, the author thinks less and less about the reader's possible wider needs.

It is clear that normal writing and publishing practices combined with the intricacies of managing working programs do not align well with the goal of readers finding reproducible algorithms. The goals and priorities of publishing are not the same as the goals of science; the pressures of publishing, whether papers or books, conspire to compromise reproducibility.

2.1. Graph theory background and applications

Leonard Euler effectively invented graph theory in 1736 with his solution to the famous Königsberg[4] bridge problem [8; 16]: is it possible to walk a cycle across each of the seven bridges in Königsberg exactly once? In modern graph terminology, a bridge is an *edge* and the land a bridge ends on is a *vertex*; if bridges are one-way to traffic, then the graph is a *directed graph* as opposed to an *undirected graph*. An *Euler* (or *Eulerian*) *cycle* is a walk that traverses each edge of a graph exactly once, starting and ending at the same vertex.

The example algorithm in this paper is concerned specifically with directed graphs. A *complete* directed graph is a graph in which each ordered pair of vertices (including repetitions) is connected by a directed edge. In other words, every pair of distinct vertices is connected by two edges, one in each direction, and each vertex has a single self-edge from itself back to itself. A complete graph of 3 vertices is shown in Figure 2.

> *Notation*. We use $u \to v$ for the directed edge (arrow) connecting vertex u to vertex v, and $u \rightsquigarrow v$ for a directed path, consisting of one or more edges connecting u to v. For example, if $u \to v, v \to w$ and $w \to x$ (also written $u \to v \to w \to x$) then $u \rightsquigarrow v, u \rightsquigarrow w, u \rightsquigarrow x$ and $v \rightsquigarrow x$, etc.

A *cycle* is a path that starts and ends at the same vertex.

A directed graph is *(strongly) connected* if it contains a directed path $u \rightsquigarrow v$ and a directed path $v \rightsquigarrow u$ for every pair of vertices u, v. Since edges are paths of length 1, complete graphs are strongly connected.

A *bridge* in graph theory is an edge that if it was deleted then the graph would no longer be connected.

A (k, n) *de Bruijn sequence* is a cyclical list of length k^n which contains k unique symbols, arranged so that every permutation of overlapping sublists of length n occurs exactly once. For example, 0011 is a $(2, 2)$ de Bruijn sequence using 0 and 1 as the 2 symbols, and where all the length 2 sublists, namely 00, 01, 11, and 10 (wrapping around), each occur exactly once, and in this order. Although the sequences were

[3] Few papers talk about more than one of: complexity, correctness, implementation, security, usability ...
[4] Königsberg was in Prussia and is now Kaliningrad in Russia.

named after Dutch mathematician Nicolaas Govert de Bruijn [5], they had been previously described in the 19th century [11; 31]. In fact, the Sanskrit poet Pingala employed a $(2, 3)$ de Bruijn sequence over $1,000$ years ago to permute poetic meters and drum rhythms [13; 20].

Modern applications of de Bruijn sequences include: magic tricks, optimal strategies for opening combination locks, and cryptography (including generation of one-time pads, and the Data Encryption Standard algorithm). They have been used in gene-sequencing to construct genomes from billions of short sequences [2]. Two-dimensional de Bruijn arrays may be used to identify the position of industrial robots on a warehouse floor, or the position of an infrared-sensing pen on specially marked paper [6]. Randomised de Bruijn sequences have applications in experimental design in many areas of scientific and statistical research, and have been used in signal processing to improve the signal to noise ratio, for instance in Magnetic Resonance Imaging (MRI) scanning [1].

Fisher proposed the application of randomised Latin Squares to balance and equalise sampling error and bias in the design of experiments into soil fertility [10]. In a similar way, if a sequence of experimental tests is prescribed by a randomised $(N, 2)$ de Bruijn sequence, every element in the series will be tested N times in random order, and will be immediately preceded by every other element in the series in the set exactly once. This approach allows the most efficient means of testing multiple items, balancing and equalising the effects of systematic bias, drift, serial carry-over effects, hysteresis and random error. We ourselves used de Bruijn sequences for experimentally evaluating automatic drug administration data from syringes [7].

de Bruijn sequences can be generated from Eulerian cycles [11; 15]. An Eulerian cycle of a complete graph with N vertices follows in order every edge $u \to v \to \ldots$ exactly once for all u and v, and this is equivalent to the definition of an $(N, 2)$ de Bruijn sequence. More generally, a *de Bruijn graph* is a graph whose Euler cycle generates the corresponding de Bruijn sequence. As a special case, the complete graph is a de Bruijn graph; Good [12] gives further examples. Other approaches for generating de Bruijn sequences include feedback shift registers and genetic algorithms [20; 45].

2.2. Classic Euler cycle algorithms

Many deceptively simple algorithms to generate Eulerian cycles have been described. These two classic algorithms were invented well before modern computers:

- **Hierholzer's algorithm** [15] first finds all cycles in the graph then merges them together.
- **Fluery's algorithm** [11] follows a path successively choosing edges to delete at each vertex by first choosing any edge that is not a bridge, and finally choosing the bridge when there is no other choice.

However their simple descriptions in English belie their relatively complex implementations in software — just saying "find cycles" and "merge" assumes all sorts of implementation details. Should you use depth first search to find cycles, and what are

the exact details for identifying bridges (especially when deleting edges changes the bridges)?

This common but deceptive simplicity of many published algorithms is a problem we also noted in our previous discussion of the Chinese Postman Tour [40].

2.3. A new Euler cycle algorithm for complete directed graphs

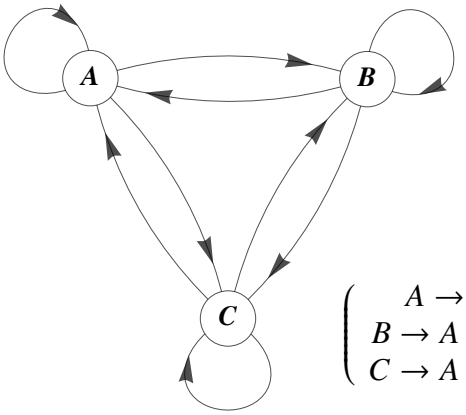
Inspired by Hierholzer's algorithm, we make five observations for an Euler Cycle algorithm for complete graphs:

1. Instead of using an algorithm to find cycles, since the graph is complete we *already* know a decomposition into cycles: namely, for every vertex u there is a cycle of length 1, $u \to u$, and for every pair of vertices u, v ($u \neq v$) there is a cycle of length 2, $u \to v \to u$. We call these *trivial cycles*.
2. We use a recursive algorithm to merge cycles. It starts walking any cycle, and if it crosses another cycle (that has not already been walked) it recursively walks that cycle, then resumes the cycle it was walking. This approach does not need any explicit operations or data structures to merge cycles.
3. Once a cycle has been walked, it is not walked again. The algorithm therefore starts by initialising every trivial cycle as unwalked, and as it walks a cycle it marks it as walked, and hence will not walk it again.
4. Marking trivial cycles can be represented by a Boolean matrix as follows: $walked_{uv}$ is true if the cycle $u \to v \to u$ has been marked as walked (if $u = v$ then the cycle $u \to u$ has been walked). We call this representation of a graph a *cycle matrix*.

Hence we suggest the following Euler cycle algorithm. Using C, vertices are numbered 0 to $N - 1$.

```
void cycle(int u, int v) // walk a cycle from u to u via v
{   if( !walked[u][v] )
    {   // we will have walked the cycle u to u via v
        walked[u][v] = walked[v][u] = 1; // keep cycle matrix symmetric
        recordEdge(u, v); // record walking edge from u to v
        for( int w = 0; w < N; w++ )
            cycle(v, w); // walk any unwalked cycles from v
        if( v != u )  // get back from v to u if not already at u
            recordEdge(v, u);
    }
}
```

As shown in Figure 2, the cycle matrix is symmetric, and might therefore be represented more compactly as a triangular matrix. However, the algorithm implements `walked` as a square symmetric matrix (i.e., `walked[u][v] = walked[v][u]` is invariant) as this simplifies the implementation. It is possible to use a more sophisticated data structure than a matrix to avoid the `for`-loop and to avoid calling `cycle` when it will immediately return: such an optimisation would not change the algorithm, but would significantly obscure its implementation.



$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} A \to A & \cdots & \cdots \\ B \to A \to B & B \to B & \cdots \\ C \to A \to C & C \to B \to C & C \to C \end{pmatrix}$$

Figure 2: Representations of the complete directed graph K_3, where there is an edge $u \to v$ for each directed pair of the 3 vertices $u, v \in \{A, B, C\}$. As can be seen (left), the graph is composed from 6 trivial cycles: 3 single-arrow cycles (e.g., $A \to A$), and 3 double-arrow cycles (e.g., $A \to B \to A$). The corresponding cycle matrix is shown top right. The cycles represented by it are shown explicitly in the larger matrix (bottom right), which has one entry shown explicitly for each trivial cycle. The omitted entries in the matrix are implied by symmetry, as, for example, the top right would be $A \to C \to A$ but this is the same cycle as $C \to A \to C$, which is shown bottom left.

The function `recordEdge` can be any way of recording the next edge along the Euler cycle; just printing it is easiest:

```
void recordEdge(int u, int v) // print an edge walked
{   (void) printf("%d --> %d\n", u, v);
}
```

The initial call of `cycle` to generate a solution is now simply:

```
cycle(0, 0); // Euler cycle starting at 0 returning to 0
```

which generates an Euler cycle that starts and ends at 0, with $0 \to 0$ as its first edge (because `cycle` starts off by going via 0, the second parameter). We discuss alternatives in the next section, below, to randomise the cycle and to avoid always starting with the same edge.

Finally, the `walked` cycle matrix needs declaring and initialising:

```
// represent cycles of a complete graph with N vertices
int walked[N][N];

// initialisation; all cycles initially unwalked
void initialise()
{  for( int u = 0; u < N; u++ )
      for( int v = 0; v < N; v++ )
         walked[u][v] = 0;
}
```

In C, arrays may be initialised to zero when declared, so explicit initialisation is not

strictly required, but we provided explicit initialisation here in case the code needs to be re-entrant or is to be translated into another language. Of course, in many languages `True` and `False` will be used instead of C's convention of 1 and 0.

2.4. Randomised sequences

Performing a sequence of experiments in random order controls the effects of drift and random error, and adding the constraint that every calibration value in the set must be preceded exactly once by every other calibration value in the set normalises any carry-over effects. Karl Popper called such sequences "shortest random-like sequences" [28]. Therefore, for experimental design, we need to modify our algorithm so that the search for an unwalked cycle to recursively follow is randomised. The easiest way to do this is to use a random permutation in `cycle`'s `for`-loop, as follows:

```
void cycle(int u, int v) // follow a cycle from u to u via v
{   if( !walked[u][v] )
    {   // keep the cycle matrix symmetric
        walked[u][v] = walked[v][u] = 1;
        recordEdge(u, v); // record edge from u to v
        for( int w = 0; w < N; w++ )
            // cycles from v via a randomly permuted vertex
            cycle(v, permutation[v][w]);
        // get back if not already at u
        if( v != u ) recordEdge(v, u);
    }
}
```

There are N independent random permutations (indexed by `v`) to avoid dependencies between between vertices: `w` is mapped by `permutation[v][w]` to a random value in the range 0 to $N-1$.

The Knuth-Fisher-Yates shuffle [19, p145–146] is a standard way to initialise such a permutation matrix:

```
int permutation[N][N];
...
for( int u = 0; u < N; u++ )
    {   for( int v = 0; v < N; v++ )
        {   int randomv = randInt(v+1);
            permutation[u][v] = permutation[u][randomv];
            permutation[u][randomv] = v;
        }
    }
```

To avoid always starting an Euler cycle from the same vertex and always starting with the same trivial cycle, the original base call `cycle(0, 0)` must be randomised too:

```
cycle(randInt(N), randInt(N));
```

Since the Knuth-Fisher-Yates shuffle is carefully designed to generate a uniform distribution of permutations, the choices made in the modified `cycle` will be uniformly random.

The function call `randInt(N)` above returns a uniformly distributed integer from 0 to $N-1$ inclusive; it is not standard C, but can be implemented using the standard `rand()` function which produces a pseudo-random integer `0..RAND_MAX`.

```
// return random integer 0..n-1 inclusive
int randInt(int n)
{  return floor(n*((double)rand())/((double)RAND_MAX));
}
```

This function assumes $n \ll \texttt{RAND\_MAX} \approx 2^{31}$ [19, p119], and it is called in a context where $n \leq N$ (where N is the order of the graph). The length of an Euler cycle is N^2, and it is implausible that (even robotic) experimenters have the time to perform sequential experiment runs approaching $N^2 = 2^{62}$ (i.e., over 10^{18}) steps, especially as randomisation is only relevant when repeated sequential experiments are run. Hence the assumption is readily met in all realistic applications.[5]

Finally, to ensure different random sequences each time the program is run, the random number generator must be seeded differently during initialisation. This may be done in C from the current time:

```
time_t t;
srand((unsigned) time(&t));
```

As is good practice, the seed for the random number is stored in a variable, so if any bug is found, a debugger can recover the value of the seed to repeat exactly the same sequence of random numbers for a subsequent test run.

2.5. Solving the wine tasting problem

The basic algorithm, described above, uses integers as the names of vertices. For many applications it may be more appropriate to use strings. We can define vertex name strings:[6]

```
// assuming N = 3
// (C permits N > 3, which will make the next line not fully initialise the array!)
char *wines[N] = { "Merlot", "Pinotage", "Shiraz" };
```

and then convert the edge recording by changing `recordEdge` to print the names of vertices rather than their numbers:

```
void recordEdge(int u, int v)
{   (void) printf("%s $\\rightarrow$\n", wines[u]);
}
```

[5] An alternative would be to use the function `arc4random_uniform()` available in some implementations of C.

[6] Not shown here, but our wine sequence generating program, which is generated automatically from this paper using our tool **relit** (see Section 3.1), checks at run time that `N` correctly matches the number of wines declared. (It is a shame that this cannot be performed with a static check in C.)

Note how we used LaTeX's `\rightarrow` to generate a "→" symbol to make the experiment sequence look a bit neater when typeset by LaTeX.

Finally, we need to finish with a final explicit `printf("%s\n", wines[0])`, because printing the second vertex was otherwise lost in changing the `recordEdge` to only print the first vertex rather than pairs of vertices.

With these changes, the code generates the following sequence: Pinotage → Pinotage → Shiraz → Merlot → Pinotage → Merlot → Merlot → Shiraz → Shiraz → Pinotage. (This actual sequence was typeset here by LaTeX inputting a file generated by running the C program, which itself was generated automatically by **relit** from the code explicitly published in this paper.)

It is easy to confirm by hand that this is indeed an Euler cycle (perhaps by mapping wines to the letters A, B, C and ticking off the edges in Figure 2). Note that, as an Eulerian cycle ends at the starting vertex, an additional drink of Pinotage (in this worked example)[7] is required: thus a balanced scientific experiment would repeat sequential experiments with randomly selected starting vertices each time, using the ideas of Section 2.4. Randomisation avoids the potential bias caused by drinking too much Pinotage — in the sequence above Pinotage happens to be drunk both first and last, and hence once more than any other wine. Randomisation also avoids possible conscious or unconscious experimenter bias caused by the researcher choosing, say, to always start or finish with their favourite wine. After enough randomised sequential experiments, these biases will be evened out.

Wine buffs will be tempted to go to a city like Königsberg, drink some wine, cross a bridge, consider the wine on the other side of the river, cross another bridge, and so on — and recording the results. However, trying it for larger N may result in a new meaning for "drunken walk."

Part II

Reproducible algorithms

3. Finding reproducible algorithms

While we continue to take it for granted that algorithms are described vaguely — in English, in pseudo-code, with illustrative fragments of uncompilable code, or left as exercises for the reader — unnecessary errors often persist in their real-life implementations. (Some of the problems are reviewed in [42; 44].) Often, of course, publications are not always aiming to describe the algorithm as such, but to do something else —

[7] The wine experiment sequence was generated automatically by running the randomised version of the code shown in this paper; it was run and the results saved to a file `winelist.tex`, the last line of which was extracted (several times) for this paragraph by using Unix's `tail -n 1 winelist.tex`. Each run of the program will generate a different random example for this paper.

like teach students, analyse complexity, prove some theorems, discuss how to optimise them, and so on. This "dual use" creates things that look very much *like* algorithms but which cannot be reliably used in practice *as* algorithms; the likely confusion calls to mind Feynman's critique of cargo cult science [9]. In fact Feynman was devastatingly critical of work that was not reproducible because it wasted everyone's time who tried to continue the work; Mlodinow describes an incident where Feynman comments on lack of reproducibility due to fraudulence [25].

Ironically, then, in one of the very few areas — programming — where we could be completely explicit about our work (e.g., if an algorithm works, it is text that can surely be published explicitly) there is a default culture of vagueness and "abstraction" that undermines scientific reproducibility if not progress. English, pseudo-code, fragments, exercises ... none are scientific statements: they are irrefutable [28].

Often software practice and experience emphasises the efficiency of running programs, but we should also be concerned with the end-to-end time to develop a program, confirm it is correctly implemented, and to run it. In many cases, the development time dominates the run time. Unlike the bulk of the algorithms textbook literature, our goal was to have a reliable program quicker, not a faster program later. As we shall argue, literate programming and its variants are powerful approaches to be more scientific when publishing algorithms, and hence to end up with more reliable programs working in the wider world. In this paper, we developed a new variant of literate programming for this very reason.

Programs that can be compiled and run involve details that are generally not relevant to discussions of algorithms. Furthermore, writing a paper or book is a human process, so transcription errors may creep into any algorithms presented.

There is even the danger that publications may be negligent: sometimes, authors do not check their published code adequately and referees take the correctness of the code on faith (partly because it is too hard to reconstruct the code from the paper, and too hard to disentangle whether problems are due to the published code or the referee's own errors in the reconstruction of it). "Negligent" is a harsh word, but it covers a wide range of common problems ranging from deliberate fraud, unintentional exaggeration, accidental uncorrected errors, and well-intentioned aspirational comments — like, the program would *obviously* work like this (even if it does not quite work yet). Even trivial and excusable errors, like typos and spelling errors, undermine the reproducibility of program code.

All this means that finding an algorithm in the literature that can be used to solve a real problem is fraught with difficulties. In our case, having developed an algorithm to solve our problem — because it is a very common problem for experimenters who may not have sufficient programming expertise — we wanted to take care that what we published (i.e., this paper) would be both clear and able to be copied from the paper as real, executable code without problem. We wanted to make it readily — and reliably — available.

At the start, we developed our algorithm first in Mathematica. Mathematica is excellent for presenting the *results* of programs in papers, as it allows documentation, code

and results to be interleaved in a polished typeset document where Mathematica itself manages all the editing. There is a *Mathematica Journal* full of papers constructed with it. However, Mathematica it is not good for writing papers *about* algorithms, as it requires code to be runnable exactly as and in the order as written; although you can hide complete blocks of code from the reader, the constraints on expression are onerous (for example, you cannot show an interesting line from within a block of otherwise hidden code). In a paper you generally want to discuss fragments or lines of an algorithm in the order that suits the exposition — you do not want to be constrained by the declaration or block syntax of the implementation language. Notwithstanding these concerns with Mathematica, we were excited by our algorithm and ported it to C (which is simpler, better known and non-proprietary). We then started to write in LaTeX. LaTeX [22] of course is a very flexible typesetting system, accepted by many journals. We were aiming to publish and share our findings as widely as possible. Finally, we ended up with the present paper.

As we wrote, we reviewed and revised this paper. We naturally made many changes to the original Mathematica program code. For example, we had initially used variables `i`, `j`... in the program, but for writing the paper we decided to use `u`, `v`... as these are conventional names for graph vertices.[8] So, over time, the original code and its description in the manuscript drifted far apart; yet, in principle, it is essential that the manuscript was synchronised with the source code so that it described the source code without error. Ideally, the paper should be automatically changing to reflect updates to the source code — but we were doing it by hand.

In a word, we did not want to be sloppy, yet our initial approach to writing the paper was making life hard. It was tempting to take an easy approach, and only describe our algorithm in words or pseudo-code, being a bit vague about the details. Conventionally, neither the readers of the paper nor we, the authors, would worry about slight discrepancies, because they would be invisible and unknown.

An obvious way to help ensure that executable code in a manuscript is correct is to cut and paste the relevant code in the paper to construct a program. This is very awkward if the code is spread out in the paper. Sometimes LaTeX commands will mean that the typeset code will differ from the intended correct source code. For many reasons, then, the code in the paper is not complete and usually has to be edited to make it work. In short, a "cut and paste" approach is not reliable.

But why do something by hand repeatedly when you can design a tool to do it with far more generality and reliability? Once a tool is written to do this chore automatically, the authors are freed from worrying about maintaining and checking the code in the document as it is repeatedly edited and revised: it should be done automatically. Reproducible authoring then becomes easier and more reliable. A win win.

This idea is very similar to Knuth's *literate programming* [18] which combines source code with explanatory documentation; however the format of the documentation gen-

[8] Unfortunately wine names starting with u, v, w, are not commonly recognisable.

erated by literate programming is not suitable for a journal manuscript, as it introduces named sections and other paraphenalia — which may be very convenient for programming, but is quite arbitrary notation to impose on a paper intended for a general readership. However, **relit** retains the section naming convention (using names like `<this>`), but by design none of them would normally appear in a paper. (Such names only occur in the present paper due to our explaining **relit**; there are none and none needed in Part I of this paper where we explain the example algorithm.)

Literate programming has the disadvantage for us (and for many authors) that the author has to start with the literate program, and in this instance we had already drafted the manuscript as a LaTeX document. **Warp** is a type of literate programming that extracts code from a normal program commented in XML, thus avoiding the separate processing that literate programming normally requires to generate the executable program [41]. Our earlier paper on the related Chinese Postman Tour [40] used **warp** to present accurate Java code.[9] **Loom** [14] (originally written by Janet Incerpi and Robert Sedgewick for their classic algorithms book [33]) is another approach, similar to **warp**, but allows the use of Unix filters to perform arbitrary transformations of code (e.g., to handle special symbols) that is then inserted into arbitrary documents.

In the present paper, however, we had already been working on the code *in* the paper, in the usual informal way. To avoid this becoming increasingly sloppy (or, conversely, a huge burden to manage), we therefore developed a novel "reversed" literate programming approach: using it, the code is exactly as written in this paper (i.e., what you are now reading) *and* it can be automatically extracted to generate a program that is directly executable. Readers of this paper can be assured the code is reproducible, and if they wish they can start with the paper, use our tool, and automatically extract all the code themselves and run it or develop their own programs directly from it.

The point is: we know that the code shown in this paper works, and moreover, we have a lightweight, fully automatic process that goes directly from this paper to executable code. What you now see as published may not be all of the code,[10] but the code shown *does work* as shown.

Figure 3 compares the main forms of literate programming, including our new approach. There are of course many other related approaches, ranging from the very simple such as our reverse literate programming to the highly sophisticated, such as PreTeX [21] that are designed for large complex projects and have commensurate learning curves: for a review see [41] and [46] for up-to-date summaries of available tools.

[9] The Chinese Postman finds the shortest cycle in a weighted graph that is not necessarily Eulerian (i.e., some edges may need to be walked more than once); it is a non-trivial generalisation of the problem discussed in this paper.

[10] There is exactly one line missing from the main section of the published paper — as our tool can tell you; see Section 3.3.

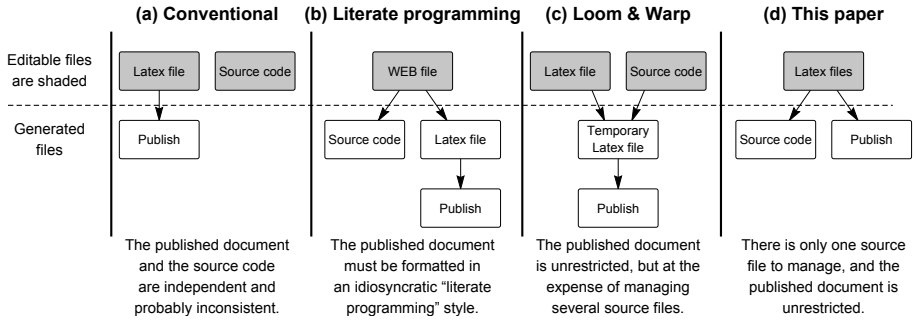


Figure 3: Comparing various ways to write about programming. Note that in the conventional approach (a) there is no guarantee that the published paper faithfully represents the source code, as the paper and source code can be (and will be) edited independently: what is published has no automatic connection to the source code. (Although not made clear in the diagram, typically source code and LaTeX documents will be split into multiple files for convenience. In principle, all methods can handle multiple files.)

3.1. Reverse literate programming — the details

Code in LaTeX documents is often written in "snippets" in the following conventional style:

```
\begin{verbatim}
printf("example code");
\end{verbatim}
```

When typeset, this will appear as "`printf("example code");`" somewhere in the typeset document — in fact, as it does in this very sentence.

However, the code the reader can see above might be — and usually would be — just text manually written in the original LaTeX document: it may have errors, it may not be real program code, and it may not have been checked by compiling it — it may have been manually copied and pasted from some program, and probably errors will have been introduced while trying to follow LaTeX conventions (such as handling the % and \ characters, which have different meanings in LaTeX and in program code).

To assure code is correct, we need to be able to generate a file from the code snippets like this, then the file and the code it is made up from can be checked. Unfortunately, the snippets may be scattered in any order throughout the paper, and perhaps the author does not want to burden the reader with all the details of the entire program so often additional code will also be required to make a compilable program.

In our approach, we name snippets, and then use our tool **relit** to collect the named snippets to assemble them into one or more source code files that can then be compiled and run in the usual way. We allow snippets to be hidden from the reader — for instance, placing them beyond the "end" of the document, that is placing them after the LaTeX's `\end{document}` command which signals the end of the typeset document. **Relit** instructions can also be placed in separate files that are not processed by LaTeX at all. Hiding snippets allows the LaTeX source file to define *all* the code required, even

when not all of it is relevant to the reader (for example, the reader does not always need to see standard declarations).

A **relit** name is defined by preceding any part of the LaTeX document with a special comment:

```
%define name /start pattern/±offset , /end pattern/±offset
```

The pattern style (general regular expressions are permitted) is deliberately reminiscent of the form used by the Unix utility **ed**, and the entire line (starting with the standard LaTeX comment symbol %) is a LaTeX comment so it is ignored completely in the typesetting of the document. The effect is that *name* is defined to be the text in the file over the specified range of lines. For example, the LaTeX for the code snippet above was preceded by a definition of the name `demo` as follows:

```
%define demo /verbatim/+1, /verbatim/-1
\begin{verbatim}
printf("example code");
\end{verbatim}
```

Here, the `%define` command defines the name `demo` to be all the text — as it happens, only one line in this case — between the line after one containing `verbatim` (which happens to be the line `\begin{verbatim}`) upto the line before the next line containing `verbatim` (which happens to be the line `\end{verbatim}`).

To create this example, for it to be typeset as shown above, we used **relit** to generate a file, and then that file was input (several times) into the paper to show the actual value of `<demo>` in each place where the text was needed. Therefore, the original text actually occurs only once in the source file, and if it is edited there, *all* of the examples and discussion get automatically updated. **Relit** thus ensures that all the examples are consistent; if we decide at a later date to improve anything there is only one place to edit — and it is impossible to forget to update each example, as updating happens automatically. Of course, repeatedly using exactly the same code into a paper may be unusual, but the point is **relit** has no problem, and in fact it was helpful for the present paper to have the flexibility to be able to do so.

Relit generates source files by writing file definitions analogously to name definitions, such as:

```
%generate filename ., /%end/-1
text ...
%end
```

However, to help assemble snippets together, within *text* any occurrence of `<`*name*`>` is replaced by its definition, and so on recursively. For example,

```
%generate hello.c ., /%end/-1
int main(int argc, const char *argv[])
{ <demo>
  return 0;
}
%end
```

will generate a file called `hello.c` that should compile and do whatever `<demo>` does. In fact, with our example, the C compiler will complain "`include the header <stdio.h> or explicitly provide a declaration for printf`" because in fact `hello.c` does not provide the needed declaration for `printf`. That is the point: the approach enables us to write a paper *and* easily check whether the code we are writing is valid. Note that **relit** can also generate makefiles [24] as well, so the entire code, compiling and testing protocols and indeed arbitrary transformations to generated files can be conveniently combined into a single LaTeX source document and maintained in one place.

Illustrating **relit**'s commands out of context (i.e., not showing the surrounding English descriptions of the code), as above, does not explicitly show the powerful leverage the approach provides for reflection during the authoring process. The description and explanation of code embeds the code in the same place; it is thus easy to improve the description *and* simultaneously improve the code, and conversely. The author can think in either language, and slip in and out of either description without losing track of their thoughts.

As a matter of fact, we realised we needed to add some explicit code to the paper only after we had implemented **relit** and found some oversights in our original description. Reverse literate programming helped us find the problems: various declarations and a bit of wider context were needed to make the algorithm described in this paper properly compile and run without error. The following code generation, creating a file `euler.c`, works correctly once **relit** substitutes the values for the various names, which have been defined elsewhere in this paper:[11]

```
%set-tag \seen{}
%generate euler.c ., /%end/-1
#define N 3 // for a graph with N vertices
<common-declarations>
<declare-walked>
<define-basic-recordEdge>
<define-non-randomised-cycle>

int main(int argc, const char *argv[])
{ <main-body>
  return 0;
}
%end
```

[11] The code shown here *is* the actual code that **relit** used — we just placed the **relit %** comment inside the LaTeX `verbatim` environment so it could be seen.

This code was written exactly as shown, and it is easy for the authors to check it actually works: with **relit** it generates the file `euler.c` which can be compiled and run, and will get the expected results. This implies the code `<main-body>` and so on recursively also compiles and works. Using **make** [24] with a makefile (also generated by **relit** from the same document) made it very easy to automatically update the code files and unit test them repeatedly as the publication evolved.[12]

Of course it is unlikely that a normal paper (that is, one not explaining **relit**) would present the code shown explicitly above: usually, the `%generate` command would have appeared after LaTeX's `\end{document}` so it would disappear and not be typeset as part of the published paper. Similarly, any details that need not be part of a typical published paper can be hidden. As we will discuss in Section 3.3, **relit** can tell the author what code is visible and what code has been hidden to avoid any uncertainty.

Notice that **relit** allows the generated program code to be put in the correct order required for a compiler, but the code visible in the paper can be presented to the reader in whatever order is best for the narrative of the publication. For example, although `<main-body>` appears near the end of the code, it might be explained to the reader before showing increasing levels of detail of the implementation.

Name definitions can occur freely before or after they are used. For example, in this paper the name `<common-declarations>` (used above) is defined later, after the end of the LaTeX document: the headers are a detail we feel readers of the present paper do not need to see written out in full, but they are necessary to be able to generate executable code. There are no restrictions on the order of generating files: they, too, can be specified in any order that best suits the needs of the author.

This complete freedom of order distinguishes **relit** from other approaches such as used in Jupyter [29] and Mathematica [47], which both impose an order on the document to suit the compiler rather than the reader or author.[13] **Org-mode** [32] is an EMACS editing environment that manages stuctured documents, like notebooks, that can generate other files, including code and LaTeX documents. Indeed, all of these approaches use special or essentially proprietary tools to edit and maintain the documents (whereas **relit** does not). In addition, these and similar tools typically require units of code to be syntactically complete, which is not required by **relit**, as made clear with the simple opening example in Section 1.2 and with the more complex Knuth-Fisher-Yates shuffle example in Section 2.4.

As well as in the document, as shown above, names can also be defined in **relit**'s command line parameters, so any textual information can be imported when the tool is run — such as a version number, the date, or even output from Unix tools such as

[12] By default **relit** only updates files when their content changes, which makes using **make** efficient, as most edits to a document do not update any files generated from it.

[13] Mathematica is interesting in that Mathematica notebooks are first class Mathematica expressions, and therefore can be processed in arbitrary ways by the notebook itself. Thimbleby [39] gives an example of a published paper generated by Mathematica deleting the author's explicit Mathematica in the notebook to leave just the final paper without the underlying program it is based on.

expect [23] to refer to unit test diagnostics.

Our tool **relit** provides all the normal checks, such as reporting if names are defined and never used, used recursively, multiply defined, etc. Indeed, some authors have used deliberate "errors" as a way of providing metacomments: defining an unused name results in its name and value being reported to the user — so the text is highlighted, which can be used as a reminder to fix an issue with the document.

Relit is a Unix tool, written as a short C program, 480 lines long and, together with this paper and documentation, is available from `github.com/haroldthimbleby/relit`[14]

3.2. Generating any text, not just program code

In general, authors of papers may want any text generated, not just program source code; for example, we showed the text generated by running our algorithm on a selection of wines in Section 2.5. In general the generated text may require further processing or testing.

There are many ways to do this: **loom** for example, generalises LaTeX's `\input` command to allow arbitrary processing, but this has the disadvantage that the approach requires an intermediate LaTeX file to be generated. Instead, our simple approach is to use `%generate` to create a makefile or any shell script: then any processing whatsoever can be performed, and of course it will typically generate files that are then included in the LaTeX paper. Indeed, this is how we generated the sample list of wine tasting — using the makefile generated by **relit**, Unix's **make** then generated a C source program `wine.c` from the paper you are reading, which was then compiled and run (in the same run of **make**), obtaining results saved to a file. Finally, that file was read in at the appropriate point when this paper was typeset (in Section 2.5), using `\input` to read in the program's saved output to insert it into the paragraph where it was needed.

Of course, files created by **relit** programming can be processed by other tools in arbitrary ways, for example stream editors to decode LaTeX typesetting conventions (e.g., in normal LaTeX, the common programming symbol `&` has to be written `\&`).

3.3. Checking hidden material — using tags

The authors of a paper should be able to check that the explicit code shown in their paper is exactly what they want the reader to see and that nothing critical has been omitted. On the other hand, a published paper should conceal details that are distracting or irrelevant to its core message: it need not show all of the code a runnable program requires, but just enough to get the idea across. The tension between these, being explicit and being concise, is a recipe for error: what is hidden, by definition, cannot be seen, yet some omitted information may be required for achieving a complete program.

[14] *See footnote 1 on page 3.*

Relit	All source code and published paper in the LaTeX source file or files. Code may be hidden, but only with deliberate effort. Intended for peer reviewed publications. Diagnostics for both missing and defined but unused code, etc.
Loom and Warp	Some code is visible, but most is hidden in separate files. Good for publications.
Literate programming	No code is hidden and all the code and document is in one place. Ideal for internal documentation rather than journal publications.
Conventional approach	No connection between published paper and program source code; there is no link between the source code and the published paper. Errors and inconsistencies are easy to introduce; no diagnostics possible.

Figure 4: What the author writes and what the reader sees should be closely aligned, ideally with the reader able to deduce full working code from the published paper. However, setup and other code is often hidden from the reader, and probably held in files separate from the published paper. The more code that is hidden, the more likely it will drift into complexity; critical details may be accidentally concealed from the reader. Conventional literate programming hides nothing, but typically makes the result too large to be convenient as a journal paper. **Loom** and **Warp** help ensure code published is correct, but it may be incomplete. The present paper's approach, **relit**, ensures that what the authors may choose not to publish can easily be checked with simple diagnostics (see Section 3.3) — and what is not published remains in the original LaTeX files, so remaining visible at least to the author.

Worse, the reader of the paper may not be certain what is missing, and they may not have the skills or time needed to reconstruct it correctly. Compounding the problem is the so-called "curse of knowledge" [27]: the authors of the paper have privileged knowledge (in principle they know everything about what they are talking about) and they may therefore be unaware that some things have not been explicitly mentioned in the paper — it is very hard to distinguish between what they know in general about what they are writing and what they think they know (perhaps inaccurately) is in the paper.

Tools like **warp** and **loom** help the authors ensure that code published has been obtained directly from the working programs, but it is still possible to write code in the paper that has never been tested or compiled, and also to leave a lot of essential context in the program that **warp** or **loom** do not draw into the document. Figure 4 summarises the challenges.

Unfortunately, in reverse literate programming, generated code can include names that are defined anywhere in the source documents, and the defined values themselves may or may not be visible in the published paper, and so on recursively. It is therefore impractical to manually determine what code is visible in the published paper and what, if any, is not visible.

Our tool, **relit**, allows code to be tagged, and although the tags can be used for any purpose, a useful application is to keep track of what code is visible and what is not. The approach is simple; the `define` and `generate` commands can be followed by optional tags:

```
%define name start, end [, tag]
%generate filename start, end [, tag]
```

For every file generated, **relit** additionally generates a duplicate file but marked up with the tags, with the relevant tags output as each *name* (or *filename*) is expanded. The tags are arbitrary text as supplied by the author, but will typically be LaTeX macro names that can be defined to highlight text in any way that the author chooses. Of course if the tags include names like <*stuff*> then they will be expanded as normal (the names can be defined anywhere, generally after the end of the document). This feature is useful if the tag is complicated (e.g., writing <tag> is easier and more reliable than writing out a tag in full every time it is needed) or if the author wants a tag to have many lines of text.

With tagging, we can readily obtain typeset text showing where code has come from.

However, since tagging each *name* definition is a bit tedious — and therefore itself error-prone — a default tag can be defined to apply to all future **relit** commands:

```
%set-tag tag
```

That *tag* is then automatically applied to all subsequent definitions (and files) until it is superseded, or overridden by explicit tags. Hence, typically a LaTeX document will start:

```
...
\begin{document}
%set-tag \seen{}
...
published document including visible definitions
```

and then have the following at its end:

```
...
\end{document}
%set-tag \unseen{}
...
hidden definitions and files
```

Here, the LaTeX code \end{document} signals the end of the published LaTeX document, and all subsequent text will be hidden from view in the published document. Normally this part of a LaTeX document is empty, or has accumulated "junk" text and thoughts the authors cannot steal themselves to *really* delete — the space has a useful role in co-authored documents, where one author wants to delete text from the published document, but does not want another author to lose some idea without a chance to reconsider it before it is deleted. In our case, with reverse literate programming, the hidden space can also be used for defining program code that is needed for compiling and testing, but is considered too much detail for visible inclusion in the published document. Although there is an advantage having everything in one file to simplify editing, if desired, **relit** allows the code to be placed in separate files if "hiding" it beyond the end of the LaTeX seems contrived.

The illustrative tags \seen{} and \unseen{} used above are arbitrary; one might choose to use \color{black} and \color{red} instead, say. In the complete example shown

below (having defined \unseen{} appropriately) the "*** hidden code ***" marker is provided automatically and therefore correctly.[15]

The code shown next, below, is one of the example algorithms already discussed in this paper: the highlighted text generated by **relit** reveals code that was not previously shown in the paper but which was generated for the compiled test programs. In other words, the highlighted code is included below, but it as used elsewhere in this paper that line was hidden from sight from readers (by being placed after \end{document}). To be clear, the code shown below was generated by **relit**, including the highlighting, and it was then inserted into this LaTeX document — essentially, the program code was typeset by writing \input{*file*}.

```
#define N 3 // for a graph with N vertices
#include <stdio.h>                                                    *** hidden code ***
// represent cycles of a complete graph with N vertices
int walked[N][N];
void recordEdge(int u, int v) // print an edge walked
{   (void) printf("%d --> %d\n", u, v);
}
void cycle(int u, int v) // walk a cycle from u to u via v
{   if( !walked[u][v] )
    {   // we will have walked the cycle u to u via v
        walked[u][v] = walked[v][u] = 1; // keep cycle matrix symmetric
        recordEdge(u, v); // record walking edge from u to v
        for( int w = 0; w < N; w++ )
            cycle(v, w); // walk any unwalked cycles from v
        if( v != u )  // get back from v to u if not already at u
            recordEdge(v, u);
    }
}
int main(int argc, const char *argv[])
{ cycle(0, 0); // Euler cycle starting at 0 returning to 0
  return 0;
}
```

The authors of this paper are happy that these standard declarations are not taking up space in the published paper — in any case, if a reader of this paper faithfully copies the published code and omits these lines, good compilers will provide error messages that help solve their problems (as we discussed above, in Section 3.1). Alternatively, a reader of this paper can download the source code of this paper, run **relit** on it, and then they will have all the executable files for all of the examples.

To summarise: using tags lets an author assure themselves that readers of their paper have the right details to reproduce the algorithm as exactly as they wish — balancing detail with verbosity and pedantry. As shown above, highlighted in the diagnostic output from **relit**, the present paper fails to disclose one line of code. However we, the authors, consider this omission an unnecessary, obvious and distracting implementation detail — in fact, we know if this line was omitted by a reader, their C compiler would help them correct it, so it is not a serious problem.

Apart from the original literate programming approach that tells the reader everything, which may be overwhelming, we know of no other approach that gives the benefits of

[15] Correctness here depends on the author not cheating! LaTeX is programmable, so a determined author could defeat the reliability of the tagging mechanism if they were so inclined.

Figure 5: An extract from a simple **relit** TEX-mode index, as made for this paper. (It is possible page numbers are incorrect in this printing, depending on how the publisher handles LATEX indexing.) Only names and files explicitly shown in this paper have been included (i.e., definitions where the normal **relit** process has not shown names to the reader, along with all hidden material after the `\end{document}`, have been excluded).*

concise algorithm publication combined with such strong assurances of reproducibility.

3.4. A more flexible TEX-mode

As described above, **relit** uses `%` to introduce commands, like `%generate` and `%define`; the advantage of this approach is that LATEX completely ignores **relit** commands because they are in the form of LATEX comments. However, in addition, **relit** provides an integrated "TEX-mode" that makes the syntax it uses real TEX or LATEX code; this is more sophisticated, requires a little TEX programming, but has some significant advantages that some authors may appreciate. In TEX-mode, **relit** becomes user-programmable, which of course is impossible with the `%`-comment form of commands.

In TEX-mode, the author defines **relit** commands to do arbitrary typesetting (as well as continuing supporting **relit**), such as printing where they are used in the document to help prepare the paper or keep track of how **relit** is being used, or an index of **relit** names can be constructed, which would help in large projects. An example (from this paper) is shown in Figure 5.

The TEX-mode syntax corresponds directly to the original syntax, and is summarised in Figure 6.

Here is an example of use the TEX-mode so the normally invisible **relit** commands used to save text have been made visible by defining `\relit` appropriately in LATEX.

*This paper, which is both a paper explaining an algorithm in the usual way and a paper explaining **relit** in detail, is complicated by unusually having to write some **relit** commands *inside* LATEX `verbatim` environments, so the reader can see them but where simultaneous indexing is impossible. (In normal use, **relit** commands would be outside `verbatim` and similar commands and hence they would be processed by LATEX and would normally be invisible to the reader.) Thus, index entries marked * in Figure 5 were provided by using duplicated **relit** commands written just outside the `verbatim` environments so LATEX could evaluate them to make the starred index entries as shown.

Normal mode	**TEX-mode**
% define *name* re_1, re_2	\relit{define *name* re_1, re_2}
% define *name* re_1, re_2, *tag*	\relit[*tag*]{define *name* re_1, re_2}
% generate *name* re_1, re_2	\relit{generate *name* re_1, re_2}
% generate *name* re_1, re_2, *tag*	\relit[*tag*]{generate *name* re_1, re_2}
% set-tag *tag*	\relit{set-tag *tag*}
% *any comments permitted*	\relit{ends}

Figure 6: Corresponding normal and TEX-mode syntax for **relit** commands. The final case in the table allows \relit{ends} to mark the end of definitions (anything convenient, such as text in a comment can also be used).

Defining \relit like this can be useful when reviewing a document.

RELIT: generate TeX-mode-demo.tex /This/, /relit.ends/-1 ←RELIT

This is a simple example that generates a file TeX-mode-demo.tex, which will contain the original LATEX source for this paragraph after **relit** is run.

RELIT: ends ←RELIT

For completness, the generated text (input from the file **relit** generated) is shown below inside a LATEX quote environment:

> This is a simple example that generates a file TeX-mode-demo.tex, which will contain the original LATEX source for this paragraph after **relit** is run.

Relit warns if TEX-mode and non-TEX-mode commands are mixed in the same files: the point of TEX-mode is that authors can use TEX to keep track of *all* **relit** commands, but using the % **relit** notation as well (which of course TEX completely ignores) would undermine reliable tracking.

The reason that **relit** makes TEX-mode an option, rather than being the only method of use, is that the feature introduces a conceptual layer of complexity the standard approach does not require. Using TEX-mode requires not just defining some TEX commands (a matter of copying them from somewhere that already works) but also understanding how to control TEX's syntax with regular expressions, which is harder as you can no longer use \, %, {, }, etc, in the same way. If you make mistakes in TEXmode strange things can happen that are hard to debug because TEX/LATEX are complex; in contrast if mistakes are made in the normal %-mode, nothing strange happens in LATEX (because **relit** commands are ignored by LATEX) and **relit** reports any errors.

3.5. Name subscripts

Relit allows authors to name and combine snippets of code, but surprisingly often the author has trouble thinking of names. For example, the publisher Elsevier wants to have a list of figure captions, and if **relit** is used to collect the figure captions, then we either need the author to invent many names and keep track of them, or we need a way of subscripting names — then the author only needs to invent one name for each class of things (such as figure captions) that they want to keep track of.

Introducing programming into **relit** would make it more complex than seems warranted, but subscripts are very useful even with no arbitrary arithmetic allowed on them.

The approach **relit** takes is that any name can have a `#` symbol in it, and this represents the subscript. Each time a name is used with a `#`, the subscript for that name increments, having started at 1. Hence,

```
%generate fig#-caption.txt ...
%generate fig#-caption.txt ...
%generate fig#-caption.txt ...
```

will generate three files called, respectively, fig1-caption.txt, fig2-caption.txt and fig3-caption.txt. If the author inserts another similar `%generate`, say between Figures 1 and 2, then the caption subscripting simply introduces a new numbered name in the sequence. The author does not have to keep inventing and keeping track of any new names — they can just copy-and-paste one subscripted name.

Using subscripted names in `%generate` is thus quite straight forward. When subscripted names are used in `%define` commands, the effect is identical: a subscripted name is defined, and the subscript increments each time it is redefined. Somewhere, unlike with file names, the author will of course want to use these defined names. To use a name, the author writes `<name1>`, `<name12>`, or whatever, as usual. **Relit** will report errors if incorrect subscripts are used, if a defined subscript is not used, and if an explicitly subscripted name is defined (like `name4` when `name#` has also been defined and has counted, in this case, to 4).

In fact, in **relit** names and file names work completely identically (they are in the same namespace), except that `%generate` creates a file as a side-effect as well as defining the name. Thus, if an author generates a file, the file name can also be used as an ordinary **relit** name, so that its value can be used anywhere else in the file, for instance to generate a set of related files. (This feature is used in the present paper: by generating such "nested" files, we were easily able to run Unix's **wc** on snippets of code within larger files.) The way subscripts work is also identical, as described above.

4. Further work on relit

The experience of developing and using a tool drives new ideas, creating a tension between polishing the legacy or generalising and extending the scope and reach of the ideas. Here are several key potential developments:

1. The syntax of **relit** could be improved. For example, instead of defining names or generating files, **relit** commands could be considered more generally as specifying arbitrary text from an expression, which is then processed. Like **loom**, commands could pipe their output to files or other Unix processes. This would generalise **relit**. At present, the same effect can be achieved, albeit after one extra step, by using makefiles and other external processes filtering the files

that are generated — but this is "bad practice" as it involves creating an arbitrary namespace, specifically the names of the temporary files.

2. An unnecessary restriction in the current version of **relit** is that regular expressions specify *lines* of text rather than strings. If an author wants to write code like `\texttt{sin(x)}` then we currently cannot pull out the value "`sin(x)`" unless the `sin(x)` is written on a line on its own, which is an awkward convention and introduces unnecessary blanks in the author's text.
3. **Relit**'s syntax is fixed, and must be adapted if it is to be applied to international languages other than English or to programming languages other than LaTeX. At present, **relit** is open source and available for any such improvements, and adapting its (currently built-in) syntax would be trivial as the parser is uses a standard regular expression engine.
4. **Relit** is an example of a "parallel language": a language grafted into an existing language, in this case, **relit**'s simple commands are squeezed into LaTeX. Parallel languages are very common, but all of them are *ad hoc* and are compromises. Think of HTML, CSS, PHP and JavaScript that have developed conventions so that they can co-exist; or Java and JavaDoc; C and its `#define` macro-language which defies C's own syntax; or XML and its metadata; even LaTeX and TeX. Conversely, JSON has no parallel language. More research is needed on parallel languages, so that when new languages are introduced they can be extended in the future without unnecessary and perhaps fragile compromises.
5. **Relit** could be extended with its own parallel languages. **Relit** allows LaTeX and code to be interleaved, but an author may want additional information or annotations, such as program specifications, test cases, invariants, contracts, author names, versioning information, etc: these might be neither program code the author wants the reader to see, nor program code that should be compiled.
6. Although **relit** readily allows a single LaTeX document to generate many programs (e.g., in the present paper we generated a Euler cycle and a randomised Euler cycle), it does not provide any built-in way to keep track of the evolution of code. Of course, existing tools can do this; for example, running **diff** on the generated programs from this paper shows several lines are different, for instance that the two comments below are different:

```
21c8
< void cycle(int u, int v) // follow a cycle from u to u via v
---
> void cycle(int u, int v) // walk a cycle from u to u via v
```

   — thus revealing a minor sloppy change between the two versions of the code given originally in Sections 2.3 and 2.4 respectively. (As usual, the lines above were generated automatically, and editing the *original* lines in this paper in Sections 2.3 and 2.4 will change the lines above correspondingly.)
7. In this paper we have argued for the value of tools like **relit**. For good reasons, we believe it enormously helps reproducibility and dependability of publications. Whether it *actually* helps other scientists, and whether it can be improved to help more authors to achieve their publication goals, is an open question.

8. Finally, a harder question, which is more useful to answer is: of several **relit**-like systems (**org-mode** [32] and others), which are more effective and why? We currently lack the theories and principles of doing and disseminating computer science so these critical questions are hard. Our final section, next, explores these bigger issues.

Part III

Reproducibility: discussion and conclusions

5. The benefits of automatic honesty

Richard Feynman's 1965 Physics Nobel Prize lecture (shared with Sin-Itiro Tomonaga and Julian Schwinger) reminds us that we polish our scientific papers to convey what we have achieved rather than how we got there. As we refine and improve our articles, their connection with the original work — our programs — can become tenuous and ultimately deceptive.

This sloppiness creates a misleadingly "productive" culture which favours quantity over quality of written output. It becomes accepted practice, because everyone seems to benefit from taking the easiest route to publish more. It gives an advantage for the scientists who do it [37]: cargo cult science is easier to publish. Indeed, what authors even read all the papers they cite [36]?

It is noteworthy and arguably a symptom of this cargo cult that the ACM Computing Classification System (CCS, `dl.acm.org/ccs`) does not even classify the core activity publishing — yet publishing (including LaTeX itself) is clearly a very active field within computer science research. Evidently, thinking seriously about publishing (which involves thinking about uncomfortable subjects like quality in publishing) is not as exciting as just publishing!

As discussed in Part I of this paper, before we had invented **relit**, we ourselves encountered the problem of discrepancies between source material and published material while were searching the literature to help us in writing early drafts of this paper: improving the English explanations in the paper naturally led us to talk about variables u, v yet the program we were writing about still had variables called i and j in it. This is an example of how it is very easy to slip without noticing from clarifying your writing to simply making it up — here, we fell into the trap of talking about u and v because those are standard ways of talking about graph vertices, but the program's original i, j were out of sight (at that time, before we had developed **relit**), and therefore incorrectly described *and we did not notice*.

There is a legitimate stage in drafting papers where authors can freely write about what they hope and intend will be true as placeholders for future work [43]. "Our program *will* work like this ..." But if what they write has no rigorous connection with reality it will never be straightforward to know when the aspirational gap has been closed,

and the authors thus risk eventually publishing misleading and time-wasting papers. Readers generally have no way to distinguish sketches and visionary ideas from actual achievements.

Feynman warned about the utter honesty — the special kind of integrity — essential to do good science [9]. But with computer tools to help we can go further than conventional scientists: we can make this honesty automatic. And when honesty is automatic, it becomes easy and reliable.

So automatic tools and techniques to encourage reproducibility, like **relit**, can help you help yourself — and help your readers. Your readers should not have to work out things for themselves you did not know you were not telling them. Our scientific writing should not rely on readers having to use their insight to interpret and clarify what we write; different readers may have different insights, and then it is not clear what our science is communicating.

When describing something that is complicated, there is a temptation to simplify and take short cuts. In the worst case, this results in publications that describe what ought to happen, what we hope happens, but there are omissions that mean the claims are not easily reproducible. Somehow it is too easy to reframe our programs so that it sounds as if they work; all the details are too hard to check even for the original authors — and as we convince themselves what we write is correct, then there seems to be less and less need to go to the trouble of checking our code. When we conceal errors — and, worse, when we conceal errors from ourselves — although our work may look good, it holds back progress [38].

If a tool like **relit** is used, the code in the paper is exactly what works. If describing it gets complicated, this cannot be denied. Instead of simplifying the narrative, the author is encouraged to improve the code so it becomes easier to describe. This means *both* the code and the paper improve, and improve together.

It must be pointed out that not everybody agrees under all circumstances. For example, Knuth writes:

> "The author [Knuth] feels that this technique of deliberate lying will actually make it easier for you to learn the ideas." [17, p*vii*]

His argument is that a reader learns, and at first things need to be simplified with "white lies" as he calls them [17, p44], and then, later with more knowledge and skill, the reader can learn from more elaborate explanations that would have been incomprehensible earlier. However, this is Knuth's strategy for writing his substantial and very successful *TEXbook*, which describes in a single document a very complex system, namely TEX itself. He knows readers will have an arduous process ahead of them, and the book tries to fit the needs of both beginners and experts. We believe that in contrast, in normal scientific publishing, particularly in publishing comparatively short peer reviewed papers — which normally focus on one or two ideas — the utter honesty championed by Feynman has important if not overriding advantages. Another point of view is that perhaps Knuth should have improved TEX so it did not require lying to explain it so well. On the other hand, one of the enormous strengths of TEX is that it has *not* "im-

proved" over time, so it is remarkably portable and dependable, unlike many programs that are continually "improving" with new changes forcing users to upgrade.

While the *TEXbook* is an exceptional piece of writing, elsewhere Knuth himself has said science is what we can explain well enough to computers [26], and that everything else is art.

Our position is that if people are publishing scientific papers about algorithms that cannot, as written, "be explained to computers" then their papers are too obscure to be called science. Such papers compromise reproducibility. Science advances when art becomes science; we hope, then, that ideas like reverse literate programming, and **relit** in particular, will help move the dark arts of publishing programming into an improving science of programming.

6. Conclusions

We developed an elegant algorithm to help design reliable sequential experiments, and in the process of writing about it, trying to do reproducible research, we developed a tool to help publish reliable papers about any algorithms in any language. This paper then combined a description of the algorithm and a description of the tool, **relit**. Our tool enabled us to write this paper with confidence that the code shown in the paper is reliable.

Our algorithm generates Euler cycles or randomised Euler cycles for complete directed graphs. It can be used to generate randomised $(N, 2)$ de Bruijn sequences for efficient sequential experimental design which controls for random variation, systematic error, and carry-over effects. The simplicity of our algorithm, written in basic C, means that it is easy to understand and translate into other languages, which makes it accessible to those engaged in experimental research.

Conventional literate programming prevents transcription errors by linking source code and manuscripts. Our novel "reverse literate programming" approach, which we have described and used in the preparation of the present paper, has allowed us to simultaneously edit the program code and its description in this paper at will, and repeatedly automatically generate an executable program we could run and use to confirm the integrity of the code as described exactly in the paper. Indeed, the paper had several examples, in different programming languages, collectively showing the flexibility of our approach. We hope that interest in literate programming is literally relit by our demonstration of its combination of simplicity, practicality, and value for reproducibile research.

This paper, including its full LaTeX source from which all code and examples can be generated, as well as all **relit** documentation are available from `github.com/haroldthimbleby/relit`[17]

[17] *See footnote 1 on page 3.*

Acknowledgements

We are very grateful to Paul Cairns, Rod Chapman and Bob Laramee for helping us greatly improve the paper.

This research was funded by EPSRC under grant no. [EP/L019272/1].

References

[1] G. K. Aguirre, M. G. Mattar, and L. Magis-Weinberg. de Bruijn cycles for neural decoding. *Neuroimage*, 56(3):1293–1300, 2011. doi: 10.1016/j.neuroimage.2011.02.005.

[2] P. E. C. Compeau, P. A. Pevzner, and G. Tesler. How to apply de bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011. doi: 10.1038/nbt.2023.

[3] T. H. Cormen. Thomas H. Cormen Professor Department of Computer Science. `http://www.cs.dartmouth.edu/~thc/#solutions`, 2016.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Boston, MA, 3rd. edition, 2009.

[5] N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

[6] P. Diaconis and R. Graham. *Magical Mathematics: The Mathematical Ideas That Animate Great Magic Tricks*. Princeton University Press, Princeton NJ, 2012.

[7] B. Eagle, D. J. Williams, and J. Dingley. Investigation of two prototypes of novel noncontact technologies for automated real-time capture of incremental drug administration data from syringes. *Anesthesia & Analgesia*, in press.

[8] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Comment. Academiae Sci. I. Petropolitanae*, 8:128–140, 1736.

[9] R. P. Feynman. Cargo cult science, 1974 CalTech commencement address. In R. P. Feynman and R. Leighton, editors, *Surely You're Joking, Mr. Feynman! Adventures of a Curious Character*, London, UK, 1992. Vintage.

[10] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver and Boyd, London, 1925.

[11] M. Fleury. Deux problèmes de géométrie de situation. *Journal de Mathématiques Élémentaires*, 2:257–261, 1883.

[12] I. J. Good. Normal recurring decimals. *Journal of the London Mathematical Society*, s1-21(3):167–169, 1946. doi: 10.1112/jlms/s1-21.3.167.

[13] R. W. Hall. Math for poets and drummers. `http://people.sju.edu/ rhall/mathforpoets.pdf`, 25 November 2015.

[14] D. R. Hanson. Printing common words. *Communications of the ACM*, 30(7): 594–598, 1987.

[15] C. Hierholzer. Ueber die möglichkeit einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873. doi: 10.1007/BF01442866.

[16] B. Hopkins and R. J. Wilson. The truth about Königsberg. *The College Mathematics Journal*, 35(3):198–207, 2004.

[17] D. E. Knuth. *The TEXbook*. Addison-Wesley, Boston, MA, 1986.

[18] D. E. Knuth. *Literate Programming*, volume 27 of *Center for the Study of Language and Information Lecture Notes*. Stanford University, Palo Alto, CA, 1992. doi: 10.1093/comjnl/27.2.97.

[19] D. E. Knuth. *Seminumerical Algorithms*, volume 2. Addison-Wesley, Boston MA, 3rd edition, 1998.

[20] D. E.. Knuth. *Combinatorial Algorithms*, volume 4A. Addison-Wesley, Reading MA, 1998.

[21] R. L. Kruse. Managing large projects with PreTEX: A preprocessor for TEX. In *TEX Users Group Annual Meeting*, pages 1070–1074, 1999.

[22] L. Lamport. *LATEX: a Document Preparation System: User's Guide and Reference Manual*. Addison Wesley, Boston, MA, 2nd. edition, 1994.

[23] D. Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reilly Media, Sebastopol, CA, USA, 1994 (corrected printing 1996). ISBN 978-1-565-92090-3. URL `http://expect.sourceforge.net`.

[24] R. Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, Sebastopol, CA, USA, 3rd. edition, 2004.

[25] L. Mlodinow. *Some Time with Feynman*. Penguin: Penguin Press Science, 2004.

[26] M. Petkovšek, H. S. Wilf, and D. Zeilberger. $A = B$. A K Peters, Wellesley, MA, 1996. Foreword by D. E. Knuth.

[27] S. Pinker. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century*. Penguin, London, 2015.

[28] K. R. Popper. *The Logic of Scientific Discovery*. Routledge Classics, London, 2002.

[29] Project Jupyter team. Project jupyter. *jupyter.org*, 2016.

[30] N. Ramsey. Literate programming simplified. *IEEE Software*, (September): 97–105, 1994.

[31] C. F. Sainte-Marie. Solution to problem number 48. *L'Intermédiaire des Mathématiciens*, pages 107–110, 1894.

[32] E. Schulte, D. Davison, T. Dye, and C. Dominik. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 2012. doi: 10.18637/jss.v046.i03.

[33] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading MA, 1983.

[34] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, Boston, MA, 4th edition, 2011.

[35] R. Sedgewick, K. Wayne, and N. Liu. Directedeuleriancycle.java. In `http://algs4.cs.princeton.edu/42digraph/DirectedEulerianCycle.java` Sedgewick and Wayne [34].

[36] M. V. Simkin and V. P. Roychowdhury. Copied citations create renowned papers? arXiv.org>cond-mat>arXiv:cond-mat/0305150, 2003. URL `https://arxiv.org/abs/cond-mat/0305150`.

[37] P. E. Smaldino and R. McElreath. The natural selection of bad science. *Royal Society Open Science*, 3(160384), 2016. doi: 10.1098/rsos.160384.

[38] M. Syed. *Black Box Thinking*. John Murray, London, 2015.

[39] H. Thimbleby. Analysis and simulation of user interfaces. In S. McDonald, Y. Waern, and G. Cockton, editors, *Human Computer Interaction 2000, Proceedings British Computer Society Conference on Human-Computer Interaction*, volume XIV, pages 221–237, 2000. ISBN 1–85233–318–9.

[40] H. Thimbleby. The directed chinese postman problem. *Software — Practice and Experience*, 33(11):1081–1096, 2003. doi: 10.1002/spe.540.

[41] H. Thimbleby. Explaining code for publication. *Software — Practice & Experience*, 33(10):975–1001, 2003. doi: 10.1002/spe.537.

[42] H. Thimbleby. Give your computer's IQ a boost. *Times Higher Education Supplement*, 9 May, 2004. URL `http://www.timeshighereducation.co.uk/story.asp?sectioncode=26&storycode=176549`.

[43] H. Thimbleby. Write now! In P. Cairns and A. Cox, editors, *Research Methods for Human-Computer Interaction*, pages 196–211. Cambridge University Press, 2008.

[44] H. Thimbleby. Heedless programming: Ignoring detectable error is a widespread hazard. *Software — Practice & Experience*, 42(11):1393–1407, 2012. doi: 10.1002/spe.1141.

[45] M. S. Turan. Evolutionary construction of de Bruijn sequences. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, AI Sec'11*, pages 81–86. ACM, 2011. doi: 10.1145/2046684.2046696.

[46] Wikipedia. *Comparison of documentation generators*. accessed 2016. URL `http://en.m.wikipedia.org/wiki/Comparison_of_documentation_generators`.

[47] S. Wolfram. *The MATHEMATICA Book*. Cambridge University Press, Cambridge, 4th edition, 1999.

Authors' biographies

Harold Thimbleby PhD, CEng, FIET, FLSW, FRCP (Edinburgh), Hon. FRSA, Hon. FRCP is at Swansea University, Wales. His research focuses on human error and computer system design, particularly for healthcare.

Harold has written several books, including *Press On* (MIT Press, 2007), which winner of the American Association of Publishers best book in computer science award. He won the British Computer Society Wilkes Medal. He is emeritus Gresham Professor of Geometry (a chair founded in 1597), and has been a Royal Society-Leverhulme Trust Senior Research Fellow and a Royal Society-Wolfson Research Merit Award holder. He has been a member of the UK Engineering and Physical Sciences (EPSRC) research council Peer Review College since 1994.

See his web site, `www.harold.thimbleby.net`, for more details.

Dr. David Williams MBChB, FRCA, Dip. DHM, PGCME is at Swansea University, Wales.

David is a Consultant Anæsthetist at Morriston Hospital, Swansea; and Honorary Associate Professor and Lead for Simulation Training at Swansea University Medical School.

He has research interests in: Human Factors, computing, and medical device design.

In addition to over 100 peer reviewed publications, he is the Director of four university spin-out companies, and has won several international product design awards. He is an editor of *Anæsthesia and Intensive Care Medicine*.

Appendix: Automatically generating figure captions

Many journals require a list of figure captions, as Elsevier does. The following list of figure captions was generated automatically using **relit** to generate a subscripted set of files, one for each figure caption, as described. This was very easy to do using **relit**'s subscript (`#`) feature, and **relit** generated a sequence of uniquely-named files, `caption-1.tex`, `caption-2.tex`, etc. Section 3.5 provides more details.

If we had been concerned, we could have used **relit**'s tags (see Section 3.3) to distinguish captions that the reader of the paper can see and those that they cannot (for some reason) see — for example, during editing a figure might have been moved beyond the `\end{document}` so it was technically preserved in the LaTeX source file, but was not actually part of the published document. Such things typically happen with co-authored papers, where one author wants to get rid of a figure (or other item) but dare not delete it completely and unnecessarily upset the other author — the text will still be there, but it will not form part of the published paper. As it happened, we decided to keep all the figures!

The list below shows how **relit** has automatically generated all of the caption files so that they are easy to include in a straight forward LaTeX loop. However, since the full captions are of course printed with each figure in this paper, to save space we used a simple Unix filter to shorten each caption with an ellipsis.

Figure 1 caption: "Conventional literate [· · ·] serted back into this paper."
Figure 2 caption: "Representations of th [· · ·] which is shown bottom left."
Figure 3 caption: "Comparing various way [· · ·] can handle multiple files.)"
Figure 4 caption: "What the author write [· · ·] ible at least to the author."
Figure 5 caption: "An extract from a sim [· · ·] excluded).*"
Figure 6 caption: "Corresponding normal [· · ·] comment can also be used)."

Appendix: Typesetting instructions

Simply, just run **make**, and everything should happen automatically. However, for the purposes of this paper, explaining what happens step-by-step is more useful.

First, then, build the executable **relit** by compiling its C source file:

```
cc relit.c -o relit
```

Now use **relit** to generate the files from this paper by running:

```
./relit relit.tex
```

For the present paper, some of these files are C and Java programs that need compiling and running to generate the examples used in the paper. **Make** has been set up to do all this correctly — in fact, the makefile is also generated by **relit** — so the details do not need to be shown here.

Now just treat everything in the standard LaTeX way. Thus, the typeset document is generated as follows:

Run `pdflatex relit.tex` to get `relit.aux`, `relit.idx`, etc
Run `bibtex relit.aux` to get `relit.bbl`
Run `makeindex relit.idx` to get `relit.ind`

This will generate the bibliography and the index (used in Figure 5) for this paper. As usual, you will typically need to run all this least twice: to get cross references correct and to get the index page numbers correct — you have to run **makeindex** twice, once to generate an index, then after running **pdflatex** again, to get the right page numbers after the index has been inserted into the document — inserting the index itself into the document changes page numbers, so another run is needed to get the right page numbers.

Useful, but not for publishing

Below is a list of the generated files (i.e., files generated directly or indirectly by **relit** needed to typeset this paper correctly) in the order as they are used in the document. There are repetitions, because some are used more than once in this paper, for example the file lastwine.tex contains just one word, "Pinotage," and is used several times in one paragraph (which was written on a single line in the LaTeX source file) as well used in this very paragraph as the final file included. (If any files do not exist or cannot be read, the list below will have appropriate error messages in it.)

java-output – source line 198 (typeset on page 4)
linesofcode.tex – source line 298 (typeset on page 10)
corelinesofcode.tex – source line 303 (typeset on page 10)
sedgewickeslinesofcode.tex – source line 303 (typeset on page 10)
allLinesofcode.tex – source line 306 (typeset on page 10)
winelist.tex – source line 585 (typeset on page 17)
lastwine.tex – source line 587 (typeset on page 17)
lastwine.tex – source line 587 (typeset on page 17)
lastwine.tex – source line 587 (typeset on page 17)
nameDemo.c – source line 649 (typeset on page 21)
nameDemo.c – source line 672 (typeset on page 22)
linesofrelit.tex – source line 740 (typeset on page 25)
demo-tagged-euler.txt – source line 835 (typeset on page 28)
relit.ind – source line 858 (typeset on page 29)
TeX-mode-demo.tex – source line 898 (typeset on page 30)
diff.txt – source line 950 (typeset on page 32)
shorter-caption-1.tex – source line 1083 (typeset on page 41)
shorter-caption-2.tex – source line 1083 (typeset on page 41)
shorter-caption-3.tex – source line 1083 (typeset on page 41)
shorter-caption-4.tex – source line 1083 (typeset on page 41)
shorter-caption-5.tex – source line 1083 (typeset on page 41)
shorter-caption-6.tex – source line 1083 (typeset on page 41)
lastwine.tex – source line 1116 (typeset on page 43)

Figure captions for Elsevier

Figure 1 caption

Conventional literate programming tools work as illustrated in Figure 1(a), which is based on a diagram in the paper on the literate programming tool **noweb** [30]: as shown, the author edits a source file (e.g., `wc.nw`) and the tool generates two files: a compilable program (`wc.c`) and a stylised LaTeX documentation file (`wc.tex`). Contrast this with Figure 1(b), which shows the comparable diagram for **relit**. With **relit**, the author edits a file (now `wc.tex`) from which **relit** can generate many files, including `wc.c`, which is compilable. Additional files generated by **relit** can be used for any purpose; for instance, in the present paper one of the generated files is a makefile so generated files were compiled and executed with their outputs inserted back into this paper.

Figure 2 caption

Representations of the complete directed graph K_3, where there is an edge $u \to v$ for each directed pair of the 3 vertices $u, v \in \{A, B, C\}$. As can be seen (left), the graph is composed from 6 trivial cycles: 3 single-arrow cycles (e.g., $A \to A$), and 3 double-arrow cycles (e.g., $A \to B \to A$). The corresponding cycle matrix is shown top right. The cycles represented by it are shown explicitly in the larger matrix (bottom right), which has one entry shown explicitly for each trivial cycle. The omitted entries in the matrix are implied by symmetry, as, for example, the top right would be $A \to C \to A$ but this is the same cycle as $C \to A \to C$, which is shown bottom left.

Figure 3 caption

Comparing various ways to write about programming. Note that in the conventional approach (a) there is no guarantee that the published paper faithfully represents the source code, as the paper and source code can be (and will be) edited independently: what is published has no automatic connection to the source code. (Although not made clear in the diagram, typically source code and LaTeX documents will be split into multiple files for convenience. In principle, all methods can handle multiple files.)

Figure 4 caption

What the author writes and what the reader sees should be closely aligned, ideally with the reader able to deduce full working code from the published paper. However, setup and other code is often hidden from the reader, and probably held in files separate from the published paper. The more code that is hidden, the more likely it will drift into complexity; critical details may be accidentally concealed from the reader. Conventional literate programming hides nothing, but typically makes the result too large to be convenient as a journal paper. **Loom** and **Warp** help ensure code published is correct, but it may be incomplete. The present paper's approach, **relit**, ensures that what the authors may choose not to publish can easily be checked with simple diagnostics (see Section 3.3) — and what is not published remains in the original LaTeX files, so remaining visible at least to the author.

Figure 5 caption

An extract from a simple **relit** TeX-mode index, as made for this paper. (It is possible page numbers are incorrect in this printing, depending on how the publisher handles LaTeX indexing.) Only names and files explicitly shown in this paper have been included (i.e., definitions where the normal **relit** process has not shown names to the reader, along with all hidden material after the `\end{document}`, have been excluded).*

Figure 6 caption

Corresponding normal and TeX-mode syntax for **relit** commands. The final case in the table allows `\relit{ends}` to mark the end of definitions (anything convenient, such as text in a comment can also be used).