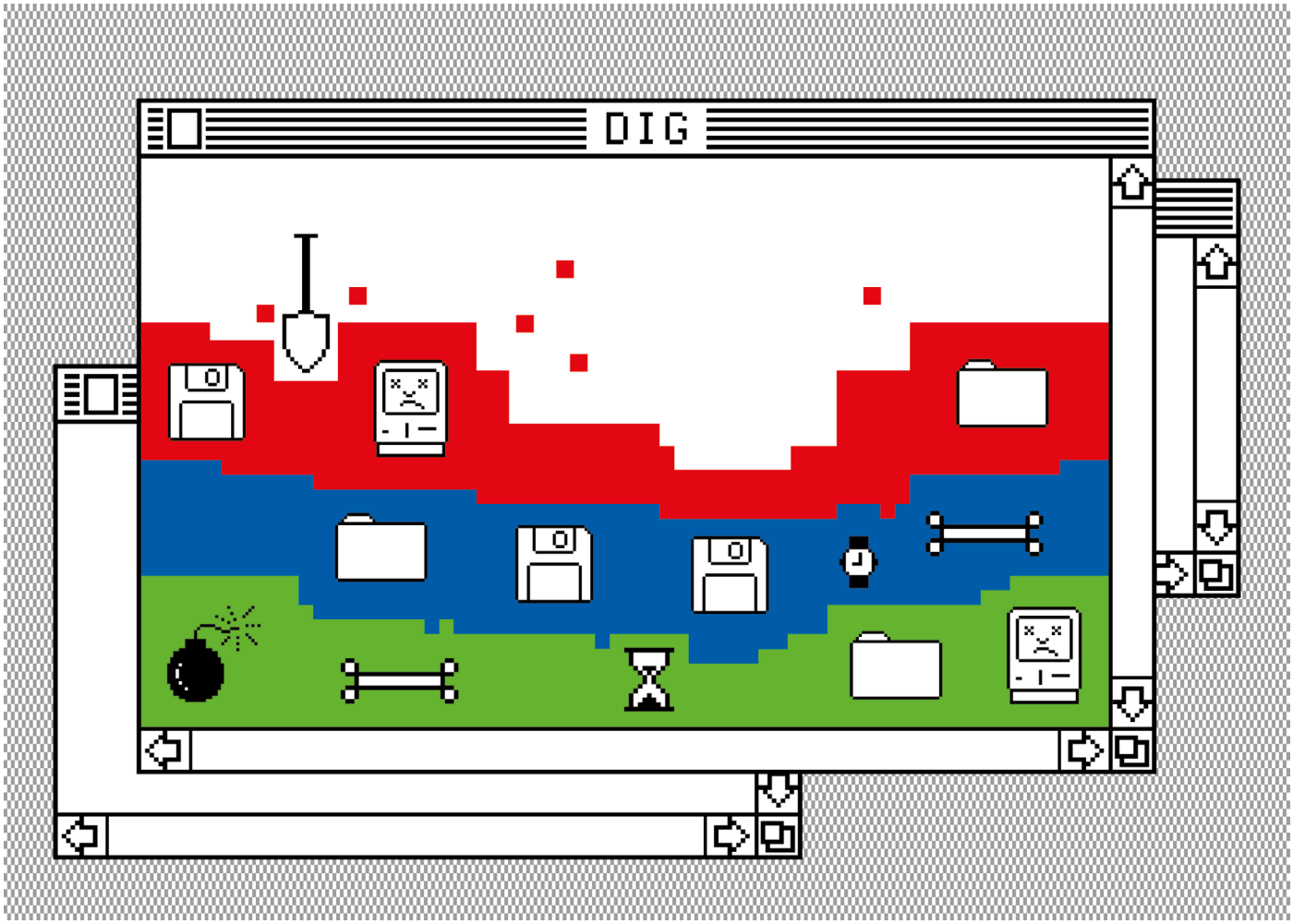


ILLUSTRATION BY THE PROJECT TWINS

THE DIGITAL ARCHAEOLOGISTS

A computational challenge dares scientists to revive and run their own decades-old code. **By Jeffrey M. Perkel**

What Nicolas Rougier needed was a disk. Not a pocket-sized terabyte hard drive, not a compact disc — an actual floppy disk.

For those who missed the 1980s, the original floppy disk was a flexible, flimsy disk inside a square sleeve with a hole in the centre and a notch in the corner, and a couple of hundred kilobytes of storage. In the 1983 cold-war film *War Games*, high-school hacker David Lightman uses one to break into the school's computer and give his girlfriend top marks in biology; he later hacks into a military network, narrowly averting a global thermonuclear war. Rougier's need was more prosaic. He just wanted to transfer a text file from his desktop Mac to a relic of the computational palaeolithic: a vintage Apple II, the company's first consumer product, introduced in 1977.

Rougier is a computational neuroscientist and programmer at INRIA, the French National Institute for Research in Digital Science and Technology in Bordeaux. That file transfer marked the final stage of his picking up a computational gauntlet he himself threw down: the Ten Years Reproducibility Challenge. Conceived in 2019 together with Konrad Hinsen, a theoretical biophysicist at the French National Centre for Scientific Research (CNRS) in Orléans, the challenge dares scientists to find and re-execute old code, to reproduce computationally driven papers they had published ten or more years earlier. Participants were supposed to discuss what they learnt at a workshop in Bordeaux in June, but COVID-19 scuppered those plans. (The event has been tentatively rescheduled for June 2021.)

Although computation plays a key and ever-larger part in science, scientific articles rarely include their underlying code, Rougier says. Even when they do, it can be difficult for others to execute it, and even the original authors might encounter problems some time later. Programming languages evolve, as do the computing environments in which they run, and code that works flawlessly one day can fail the next.

In 2015, Rougier and Hinsen launched *ReScience C*, a journal that documents researchers' attempts to replicate computational methods published by other authors, based only on the original articles and their own freshly written open-source code. Reviewers then vet the code to ensure it works. But

even under those idealized circumstances — with reproducibility-minded authors, computationally savvy reviewers and fresh code — the process can prove difficult.

The Ten Years Reproducibility Challenge aims "to find out which of the ten-year-old techniques for writing and publishing code are good enough to make it work a decade later", Hinsen says. It was timed to coincide with the 1 January 2020 'sunset' date for Python 2, a popular language in the scientific community, after 20 years of support. (Development continues in Python 3, launched in 2008, but the two versions are sufficiently different that code written in one might not work in the other.)

"Ten years is a very, very, very, very long time in the software world," says Victoria Stodden, who studies computational reproducibility at the University of Illinois at Urbana-Champaign. In establishing that benchmark, she says, the challenge effectively encourages researchers to probe the limitations of code reproducibility over a period that "is roughly equivalent in the software world to infinity".

The challenge had 35 entrants. Of the 43 articles they proposed reproducing, 28 resulted in reproducibility reports. *ReScience C* began publishing their work earlier this year. The programming languages used ranged from C and R to Mathematica and Pascal; one participant reproduced not code but a molecular model, encoded in Systems Biology Markup Language (SBML).

Akin to archaeological digs for the digital age, participants' experiences also suggest strategies for maximizing code reusability in the future. One common thread is that reproducibility-minded scientists need to up their documentation game. "In 2002, I felt like I would just remember everything forever," says Karl Broman, a biostatistician at the University of Wisconsin, Madison. "It was only later that it became clear that you start to forget things within a month."

We redo science

Rougier's entry reproduces the oldest code in the challenge[1], an image magnifier for the Apple II that he wrote aged 16 and published in a now-defunct French hobbyist's magazine called *Tremplin Micro*. (The oldest scientific code in the challenge, described in an as-yet-unpublished paper submitted to *ReScience C*, was a 28-year-old program written in Pascal for visualizing water-quality data.) Thirty-two years later, Rougier no longer remembers precisely how the code, with its arcane AppleSoft BASIC instructions, works — "which is weird, because I wrote it". But he was able to find it online and make it run on a web-based Apple II emulator. That, he says, was the easy bit; the hard bit was running it on an actual Apple II.

The hardware wasn't the problem — Rougier had an Apple II in his office, salvaged when a colleague was cleaning out their office. "For the younger people it's, 'oh, what's this?'," he says. "So you explain, 'this is a computer'. And for older people, it's, 'oh, yeah, I remember this machine'." But because the Apple II pre-dates both USB cables and the Internet — and because modern computers cannot directly talk to old disk drives — Rougier needed some custom hardware, not to mention a box of vintage floppies, to allow the computer to load the code. Those he found on Amazon, marked 'new' but dating from 1993. After triple-writing his data to ensure the bits were stable, the disks worked.

Bruno Levy, a computer scientist and director of an INRIA research centre in Nancy, reviewed Rougier's write-up. Levy also has an Apple II, and posted a short video of the result to Twitter. With a sturdy 'click clack' at the old-school keyboard, he calls up the code and runs it, a stylized "We redo science!" screen rendering slowly in monochromatic green.

"Researchers probe the limits of code reproducibility over a period that 'is roughly equivalent to infinity'."

Extinct hardware, dead languages

When Charles Robert, a biophysical chemist at the CNRS in Paris, learnt about the challenge, he decided to use it to revisit a research topic he hadn't looked at in years. "It gave me an additional kind of kick to get going in that direction again," he says.

In 1995, Robert was modelling the three-dimensional structure of eukaryotic chromosomes in computational notebooks running Mathematica, a commercial package. Robert has Mathematica on his MacBook, but for fun, he spent €100 (US$110) on a Raspberry Pi, a single-board hobbyist computer that runs Linux and has Mathematica 12 pre-installed.

Robert's code ran largely without issue but exposed difficulties[2] that can arise with computational notebooks, such as deficiencies in code organization and code fragments that are run out of order. Today, Robert circumvents these problems by breaking his code into modules and implementing code tests. He also uses version control to track changes to his code and notes which version of his software produced each set of results. "When I look at some of my old code, I cringe sometimes and think how I would do it better now," he says. "But I also think that process helped to lock in some of the lessons I've picked up since then."

Robert's success in the challenge is typical: only two of the 13 reproducibility write-ups published so far document failed attempts. One was from Hinsen, who was stymied by the magnetic tapes on which he methodically stored his code in the early 1990s[3]. "That's the problem of actually making backups but not checking that you can still read your backups ten years later," he says. "At some point you have this nice magnetic tape with a backup, and no reader for it any more." (Hinsen also published a successful attempt.[4]) Other researchers who failed to complete the challenge blamed a lack of time, especially in light of the pandemic.

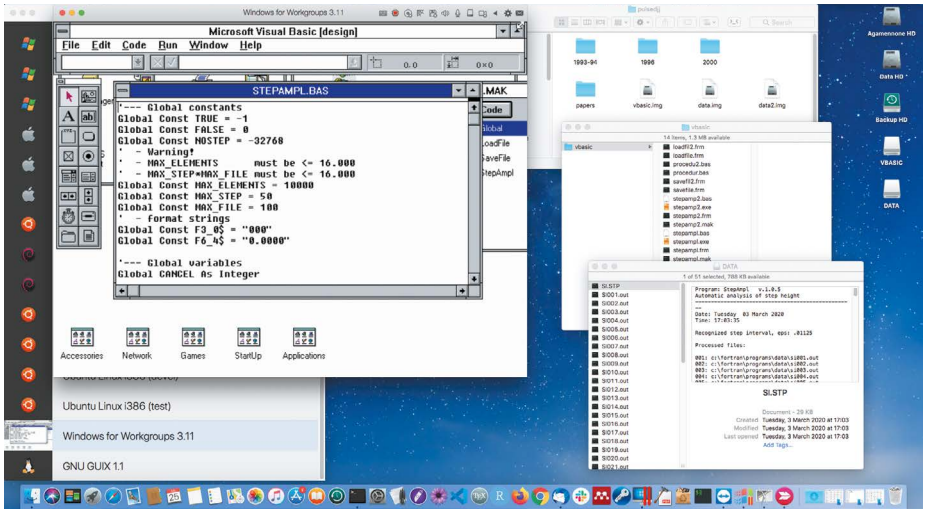
Another common issue that participants faced was that of obsolete computing environments. In 1996, Sabino Maggi, now a computational physicist at the Italian National Research Council's Institute of Atmospheric Pollution Research in Bari, used the computer language Fortran to model a superconducting device called a Josephson junction, processing the results with Microsoft Visual Basic. Fortran has changed little in the intervening years, so after a few tweaks Maggi's code compiled without issue. Visual Basic posed a bigger problem.

"Visual Basic," Maggi writes in his report[5], "is a dead language and long since has been replaced by Visual Basic.NET, which shares only the name with its forefather." To run it, he had to recreate a decades-old Windows virtual computer on his Mac. He loaded it with Microsoft DOS 6.22 and Windows 3.11 (both from around 1994) as well as Visual Basic, using installation disks he found online. "Even after so many years, the legitimacy of installing proprietary software in an emulator might be questionable," Maggi concedes. But as he had valid licences for these tools at the time of his original research, he says, he felt "at least morally authorized" to use them.

But which version of Visual Basic to try? Microsoft released multiple versions of the language over the years, which were not always backwards-compatible. Maggi could no longer recall which version he was using in 1996, and a basement water leak had destroyed the old notebooks in which he had logged those details. "I had to start from scratch," he says.

Ludovic Courtès, a research engineer at INRIA in Bordeaux, reproduced a 2006 study comparing different data-compression strategies, whose code was written in C (ref. 6). But changes to the application programming interfaces (APIs) that programmers rely on prevented his code from compiling using current software libraries. "Everything has been evolving — except, of course, some of the pieces of software that were used for the paper," he says. He ended up having to roll back half a dozen computational components to older versions — a 'downgrade cascade'. "It's a bit of a rabbit hole," he says.

Today, researchers can use Docker containers (see also ref. 7) and Conda virtual environments (see also ref. 8) to package computational environments for reuse. But several participants chose an alternative that, Courtès suggests, "could very much



A Mac emulating a 1994 Windows computer to run Microsoft Visual Basic.

SABINO MAGGI

represent the 'gold standard' of reproducible scientific articles": a Linux package manager called Guix. It promises environments that are reproducible down to the last bit, and transparent in terms of the version of the code from which they are built. "The environment and indeed the whole paper can be inspected and can be built from source code," he says. Hinsen calls it "probably the best thing we have right now for reproducible research".

Documentation needed

In his reproducibility attempt[9], Roberto DiCosmo, a computer scientist at INRIA and the University of Paris, highlighted another common difficulty for challenge participants: locating their code in the first place. DiCosmo tackled a 1998 paper that described a parallel programming system called OcamlP3l. He searched his hard disk and back-ups, and asked his 1998 collaborator to do likewise, but came up empty. Then he searched Software Heritage, a service DiCosmo himself had founded in 2015. "There it was, incredible," he says.

Software Heritage regularly crawls code-sharing sites such as GitHub, doing for source code what the Internet Archive does for web pages. Developers can also request that the service archive their repositories, and the challenge rules required participants to do so. DiCosmo didn't start his search at Software Heritage, because the service did not exist when he developed OcamlP3l. Somebody must have posted his code to the now-extinct repository Gitorious; Software Heritage archived the site before it shut down, bringing OcamlP3l along for the ride.

Of course, finding the code doesn't mean it's obvious how to use it. Broman, for instance, reports that missing documentation and "quirky" file organization meant he had difficulty working out exactly which code he needed to run to reproduce his 2003 study[10]. "And so I had to resort to actually reading the original article," he writes.

"It's not unusual for the number of lines of documentation [in well-organized programs] to actually exceed your code," says Karthik Ram, a computational-reproducibility advocate at the University of California, Berkeley. "Having as much of that in there, and then having a broader description of how the analysis is structured, where the data come from, some metadata about the data and then about the code, is kind of key."

Melanie Stefan, a neuroscientist at the University of Edinburgh, UK, used the challenge to assess the reproducibility of her computational models, written in SBML. Although the models were where she expected them to be, she could not find the values she had used for parameters such as molecular concentrations. Also not well documented were key details of data normalization. As a result, Stefan was unable to reproduce part of her study. "Even things that are kind of obvious at the time that you work on a model are no longer obvious, even to the same people, 10 or 12 years later – surprise!" she deadpans.

"Software is a living thing. And if it's living it will eventually decay, and you will have to repair it."

Reproducibility spectrum

Stefan's experience galvanized her to initiate laboratory-wide policies focusing on documentation – for instance, supplementing models with files that say, "to reproduce figure 5, this is exactly what you need to do".

But developing such resources takes time, Stodden notes. Cleaning and documenting code, creating test suites, archiving data sets, reproducing computational environments – "that's not something that's turnkey". Researchers have few incentives to do those things, she adds, and there's scant consensus in the scientific community on what a reproducible article should even look like. To complicate matters, computational systems continue to evolve, and it's hard to predict which strategies will endure.

Reproducibility is a spectrum, notes Carole Goble, a computer scientist and reproducibility advocate at the University of Manchester, UK. It ranges from scientists repeating their own analyses, to peer reviewers test-driving code to show that it works, to researchers applying published algorithms to fresh data. If nothing else, Goble says, release your source code, so that in future, others can browse it and rewrite it as needed – "reproducibility-by-reading", as Goble calls it. "Software is a living thing," she says. "And if it's living it will eventually decay, and you will have to repair it, and you'll have to replace it."

Counter-intuitively, many challenge participants found that code written in older languages was actually the easiest to reuse. Newer languages' rapidly evolving APIs and reliance on third-party libraries make them vulnerable to breaking. In that sense, the sunsetting of Python 2.7 at the start of this year represents an opportunity for scientists, Rougier and Hinsen note. Python 2.7 puts "at our disposal an advanced programming language that is guaranteed not to evolve anymore", Rougier writes[1].

Whichever language and reproducibility strategies they use, researchers would be wise to put them to the test, says Anna Krystalli, a research software engineer at the University of Sheffield, UK. Krystalli runs workshops called ReproHacks for researchers to submit their own published papers, code and data, and challenge participants to reproduce it. Often, she says, they cannot: crucial details, obvious to the authors but opaque to others, are missing. "All the materials that we're producing, if we don't actually use them or engage with them then we don't really know if they are reproducible," Krystalli says. "It's much harder, actually, than people think."

Jeffrey M. Perkel is technology editor for *Nature*.

1. Rougier, N. P. *ReScience C* https://doi.org/10.5281/zenodo.3886628 (2020).
2. Robert, C. H. *ReScience C* https://doi.org/10.5281/zenodo.3886412 (2020).
3. Hinsen, K. ReScience C https://doi.org/10.5281/zenodo.3889694 (2020).
4. Hinsen, K. *ReScience C* https://doi.org/10.5281/zenodo.3886447 (2020).
5. Maggi, S. *ReScience C* https://doi.org/10.5281/zenodo.3922195 (2020).
6. Courtès, L. *ReScience C* https://doi.org/10.5281/zenodo.3886739 (2020).
7. *Nature* **575**, 247–248 (2019).
8. *Nature* **573**, 149–150 (2019).
9. Di Cosmo, R. & Danelutto, M. *ReScience C* https://doi.org/10.5281/zenodo.3947641 (2020).
10. Broman, K. W. *ReScience C* https://doi.org/10.5281/zenodo.3959516 (2020).