Science in the digital era

Konrad Hinsen

▶ **To cite this version:**

Konrad Hinsen. Science in the digital era. 2024. hal-04762424

HAL Id: hal-04762424
https://hal.science/hal-04762424v1

Preprint submitted on 31 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

Science in the digital era

Konrad Hinsen

Centre de Biophysique Moléculaire
UPR4301 CNRS
Rue Charles Sadron
45071 Orléans Cedex 2
France

Synchrotron SOLEIL
Division Expériences
Saint Aubin - BP 48
91192 Gif sur Yvette Cedex
France

Date: 2024-05-27

Welcome

This is an archival copy of my digital garden whose current on-line version can be consulted at

https://codeberg.org/khinsen/science-in-the-digital-era

This archival copy corresponds to commit 5a48e50cf60fa7ea78d9111bf2465e416902e8a1, which was published on 2024-05-27.

This digital garden contains essays, thoughts, random ideas, and references that relate to the practice of scientific research in the digital era, characterized by computers (personal, high-performance, cloud, …), software, the Internet, global collaborations, social networks, and more. They represent exclusively my personal views and in particular not those of my employer.

There are many empty pages in this collection, and you may wonder why.

One reason is that this digital garden is work in progress. When I work on a page, I often insert links to pages that I intend to write, but haven't written yet. So you see an empty page. If you come back later, you may find some real content there. So… come back often.

The second reason is that empty pages are useful link targets, due to the backlink feature in the online version of my digital garden. At the end of each page, you see a list of other pages that link to the current one. Empty pages thus fulfill the role of a subject index in a traditional book: they help you find where some topic is discussed. Unfortunately, this feature is lost in this archival copy.

License

The pages of this digital garden are covered by the Creative Commons License CC BY-NC-SA 4.0.

According to this license, you are free to: - Share: copy and redistribute the material in any medium or format. - Adapt: remix, transform, and build upon the material.

under the following conditions: - Attribution: You must give appropriate credit. - NonCommercial: You may not use the material for commercial purposes. - ShareAlike: You must distribute your contributions under the same license. # About the author {#About-the-author}

My name is Konrad Hinsen, and I have been a research scientist at CNRS in France since 1998. You can find me on Mastodon

My scientific career started in statistical physics, with a thesis on colloidal suspensions at RWTH Aachen in Germany. My thesis was based mostly on computer simulations that ran on the Cray Y-MP at the Forschungszentrum Jülich. The simulation programs were small Fortran codes that I wrote and tested myself.

After my thesis I moved to computational biophysics, applying various simulation techniques to proteins and (to a much lesser degree) DNA. I fell into a different universe, one where most researchers are users of a small number of simulation packages written by an equally small number of groups. As a consequence, most scientists cannot inspect in detail, let alone modify, the models and methods they apply in their research. Computational disempowerment. While this situation is understandable in its historical context (the models for biomolecules are complex and the size of the simulated systems requires optimized code), I consider it unacceptable in the long run. Models and methods are at the heart of science and we should never allow them to be obfuscated or hidden.

This experience, together with my first encounter with computational (ir)reproducibility around the same time (1995), was the starting point for my second topic of research: the methodology of computational science, or, as I prefer to frame it nowadays, computer-aided research.

My first idea was to make scientific software more accessible through the use of high-level languages, and after I discovered the Python language, I ended up becoming a founding member of the Matrix-SIG, which developed Numerical Python, the predecessor of NumPy. On that basis I then developed the Molecular Modelling Toolkit. In contrast to the popular simulation packages of the time (1997), it was not a program with a fixed feature set, but a toolkit of basic algorithms that researchers could use from their own Python scripts.

The move from Fortran to Python did indeed lower the entrance barrier to becoming a developer, but still most biophysicists did not want to become developers, and, more importantly, models and methods were still hidden in computer code that had to respect constraints related to efficient executability. It takes less time to find and read the relevant code section in MMTK than in older software, but scientists still cannot study and work with their models directly. And then, the lower entrance barrier of Python completely changed to way scientists write and use software, with one big negative impact being the fragility of the scientific Python ecosystem. Whereas the Fortran code of my thesis still compiles and runs, many of my Python scripts of ten years ago have become hard to use, and even harder to trust. Reproducibility has become a major challenge, not only in the Python ecosystem.

The approaches I am currently exploring for giving scientists more control over their models and methods are digital scientific notations and re-editable software, within the larger goal of creating computational media for science. I am also closely following research on similarly-minded topics, such as explainability in machine learning and human-computer interaction. As Richard Hamming famously said, the purpose of computing is insight, not numbers. There is no point in performing massive computations if nobody knows if their results can be trusted or how they can be interpreted scientifically.

Agency

Agency is about exploring and shaping one's environment to achieve one's goals. Technology, including software, can both augment and constrain agency. Most technology increases the agency of some agents while constraining others. Agents can be individuals, coordinated groups of individuals (teams, families, associations), or formally structured organizations of sizes varying from a small company to a nation state or a multinational corporation.

Exercing agency requires an effort. Given that all agents have limited resources, they may well prefer technologies that grant them restricted but easier to exerce agency. That's what we all do when buying bread rather than making our own: we get less control over what we eat, but we also get bread with less effort. Agency is thus not something that agents wish to maximize. It is merely one out of many criteria in decision making.

In scientific research, agency is the ability to explore a scientific question and to critically examine the work of others. Scientific agents are individual researchers, research teams, laboratories, large-scale collaborations, or institutions such as universities or national research organizations.

The ubiquity of software in today's research has made software a critical aspect of scientific agency. Each piece of software augments the agency of some agents, to varying degrees depending on their technical competence and the learning effort they invested. On the other hand, when the use of some piece of software is imposed, by regulations or by social norms, the agency of some agents is constrained by it. Whereas the empowering aspect of scientific software is widely advertised, in particular by its developers, the category of agents that is empowered is rarely made explicit, and the category of agents that is constrained is hardly mentioned at all.

As a help for evaluating someone's agency over a piece of software, I propose

the following descending scale:

Full control. The agent understands the software in sufficient detail to know what it does, verify its correct working, debug it, judge its adequacy for a given scientific context, and adapt it to different questions.

Autonomous use. The agent understands the software in sufficient detail to know what it does, confirm its correct working, and judge its adequacy for a given scientific context. The agent is not able to modify the software.

Trusting use. The agent can use the software based on available documentation, but must trust others for verifying its correctness. The agent has a mental model of how the software works, but cannot interpret the source code, for lack of access to it or for lack of technical competence.

Constrained use. The agent can execute the software, supply the information it requests, and retrieve information it provides. The agent does not have a detailed mental model of what the software does.

Finding one's place on that scale is not an easy task, because it depends on many aspects. It depends on the composition and competence of the agent, which can be highly variable (think of a research team composed mainly of PhD students and postdocs on short-term contracts). It depends on the software under scrutiny, but also on the technological context in which it is accessible. Consider a notebook about a data analysis. With just the notebook file and a description of its computational environment, users have high agency but only if they can handle deployment. The same notebook run on-line via Binder requires much less technical competence, but also restricts agency because users can neither inspect nor change the computational environment. This distinction also highlights the importance of the software's dependencies. The users' agency over dependencies is typically lower, and that may well be fine, but for software like scripts and notebooks that provide only the shallow top layer of a computation, agency over dependencies can be the most import criterion in the evaluation of total agency.

Tooling can augment agency, as is nicely illustrated by computational notebooks, which increase agency by facilitating interaction with the code. Glamorous Toolkit illustrates that this idea can be taken much further if one is willing to consider alternatives to the development tools inherited from the software industry, which by design draw a sharp borderline between developers and users, augmenting the agency of the former but explicitly reducing the agency of the latter.

Agent-based model

This page is empty

Between Scripts and Applications. Computational Media for the Frontier of Nanoscience

An article (also available as a free preprint) that describes common problems with today's state of the art in computer-aided research, and proposes a solution via the introduction of computational media.

The article describes a study of how a team working in experimental biomolecular nanoscience uses computational tools in their daily work. The computational environment of this team is quite typical for the natural sciences today:

- multiple platforms (Windows, Linux, macOS, …)
- black-box applications that provide a rigid set of well-known functionality
- badly documented scripts that were written by researchers for a specific project, with no intention of maintenance
- dependencies of those scripts (that nobody could even list exhaustively)

This situation creates computational disempowerment, as scientists cannot work safely and productively with this mess, but they cannot work at all without it.

The authors report on the experimental introduction of a computational medium (Webstrates) into the workflow of this research team.

Binary code

This page is empty

Blockchain

This page is empty

Bugs

Software has bugs. *All* software. A lot of effort is spent in software engineering on coming up with tools and procedures to reduce the likelihood of bugs in software, sometimes with the dream of zero bugs.

I find this attitude surprising. No technology is perfect. All devices have limitations, all physical devices break at some point. So the real question is: why do we expect software to be different?

Part of the answer is probably overly ambitious goals. Automation requires high reliability of the automated steps, otherwise the reliability of the automated system is not only weak, but unpredictable. Much software in use consists of many layers of automation. Deploying such software would be irresponsible if we accepted that its components are unreliable. Denial lets us move on, scale up automation, save money, and let the users suffer from the resulting mess.

The main experiential insight I got from using Smalltalk is the value of debuggability. Lots of Smalltalk software I have used has bugs. But after a while, I wasn't scared any more by a debugger window popping up. Often it isn't important: close the debugger and move on. Or it is something I can explore and fix myself, since all the source code is there and debugging support is excellent. And for a really critical issue, I can e-mail the stack trace to the developer and ask for help, which developers are more likely to provide because they can just load the stack trace into their debugger and analyze the exact problem I had.

From that perspective, the problem with software is often not that it has bugs, but that the people affected by the bugs have to way to deal with them in a satisfactory way. And that's usually not a technical, but a social issue.

It's about communication between software architects and developers on one hand and software users on the other hand. It's also about the gradient in technical competence between these two parties. Convivial software can be buggy because it is debuggable. At the other end of the scale, software controlling safety-critical machines (such as airplanes) in real time needs to be bug-free because their users don't even have the time to deal with bugs before it is too late, even if they had the competence.

In computational science, debuggability enters into considerations of trust. An individual scientist, or even a typical team, cannot check all the software they depend on for bugs, simply because there is too much of it. They can check the most domain-specific layers, and they should do so because nobody else can do it, for lack of domain knowledge. For everything else, they need to trust others to find and fix critical bugs. This is a lot more likely to happen if the software is debuggable.

Building a Web of Trust for Open Science

The goal of science is to construct collectively a corpus of reliable knowledge. This requires strong quality control, which is in fact what makes the difference between science and other ways of acquiring knowledge. In the following, I will explain why I see Open Science as an essential ingredient to keep quality control in science effective.

In the early days of science, quality control was informal. Scientists could read and understand all publications in their field and form their own judgment, because both scientists and publications were few in number. With the enormous growth of science starting in the 1950s, peer review of publications became the cornerstone of quality control. Scientists trusted work that they had not examined themselves because they trusted the peer review system and the people who supervised it, in particular journal editors. Indirect trust, through the reputation acquired over time by journals, research institutions, and also individual researchers, became a very important aspect of figuring out which results to consider reliable. However, reputation is a reliable source of trust only if it is ultimately grounded in real expertise, i.e. scientists judging the work of others based on their own understanding of it.

Today, this traditional chain of trust does not work any more. The reproducibility crisis is perhaps the most visible symptom of scientists losing trust in peer review. The public debate on climate change illustrates that the general public is losing trust in science. Something has gone wrong.

The peer review system was set up at a time when a typical publication had one to three authors, from a single discipline. Collecting reports from two or three experts in the same discipline was then a reasonable approach to evaluating the quality of the work. All reviewers could be expected to fully

understand the work, and having several reviewers would reduce the risk of problematic aspects going unnoticed. In some disciplines, e.g. theoretical physics, nothing much has changed since the 1950s, and peer review continues to work rather well.

In other disciplines, such as the life sciences or climate research, a typical publication summarizes the work of a large multi-disciplinary collaboration. A proper evaluation of such work cannot be done by any individual in a week. It takes another multi-disciplinary team, and it takes a lot more time. Moreover, it takes access to much more than a few-page summary of the work. This is something the Open Science movement has recognized and improved. Publication of data and code is still not universal, and many details remain to be worked out, but it seems uncontroversial to me that this is where we need to go, and the transition has started.

However, publishing data and code is not enough to build trust in large-scale multidisciplinary research. It's not even enough to build trust in datasets and code. For each form of research output, we need techniques for acquiring expert judgment and then summarizing and relaying it to a wider audience. And perhaps even "research output" is the wrong unit of evaluation. Perhaps the only practicable way to proceed is to have independent experts follow along as a research project, or a software development project, progresses through its various phases.

Ultimately, what we need to construct is a Web of Trust, much like the concept of the same name in cryptography. For a start, we could ask authors of a scientific paper to indicate, for each artifact that they rely on (paper, software, dataset, …), to what degree and for what reason they consider it reliable. We could also collect such judgments outside of the publication process, for example on social media. At the very least, it would force everyone to think about the question. After a while, we'd have an interesting graph of trust relation to analyze. I expect some surprises, in particular "trust bubbles": artifacts that people trust because everybody else seems to trust them, without any grounding in qualified expert judgment.

Now is a good time to think about this. Elon Musk buying Twitter has led scientists to discover and adopt Mastodon, a social network based on an open protocol. We now have a social network that scientists know and use, and which is open to extensions. Developing the infrastructure for building a trust graph is within reach.

If you have ideas or suggestions about this, please comment on Mastodon!

Cellular automaton

This page is empty

Cheap complexity

In a 2018 talk at CyCon, a conference on cyber conflict, Thomas Dullien identifies one of the root causes of increasing security issues with computing technology as "the anomaly of cheap complexity." A quote from page 8:

```
For most of human history, a more complex device
was more expensive to build than a simpler device.

This is not the case in modern computing. It is
often more cost-effective to take a very
complicated device, and make it simulate simplicity,
than to make a simpler device.
```

What causes this anomaly is economies of scale driving the development of complex general-purpose computing hardware, which can then simulate any simpler machine using software, whose cost at scale is much lower. Unfortunately, the complexity of the general-purpose hardware tends to create unexpected behavior, which evil-minded adversaries can exploit in attacks.

What I would like to point out in the following is that (1) cheap complexity is also an issue in science and (2) it occurs in software for much of the same reasons as in hardware. I will start with point 2, because it is the main contribution to point 1.

Software may be cheaper at scale (i.e. as the number of its users grows) than hardware, requiring no material resources, but producing software is nevertheless very expensive. It thus becomes advantageous to develop complex general-purpose software, whose development cost is shared by a larger number of users, than simpler situated software serving only few users. Assuming, of course, that the complexity incurs no excessive cost in development. Every software developer knows that complexity does come at

a price, in the form of technical debt. However, the software industry has so far escaped from paying the cost because it is not held responsible for the problems caused by bad software. As an illustration, computer viruses have caused enormous economic damage, and yet the companies selling the software that viruses attack are not held responsible for this damage, which is attributed to inadequate vigilance by its users instead. It's thus users who pay the price for the complexity of software.

So why does all this matter for science? Security is rarely a concern in computer-aided research, at least not beyond issues such as viruses, which affect all users of computers. The enemy of science is not an evil-minded hacker, but good old human nature, well-known to be prone to making mistakes. Science is all about the acquisition of reliable knowledge. If the knowledge is encoded in computational media, using computational tools, then those tools had better be reliable as well. Complexity is the enemy of reliability. And we all know from first-hand experience that our computers and the software they run are not reliable. How often do you reboot your computer to fix a problem? How often do you install security updates? Is it reasonable to believe that *scientific* software is exempt from these complexity-related reliability issues, just because its failures are less spectacular?

So what do complexity-related failures in computer-aided research look like? The first ones that come to mind are wrong results. Unfortunately, we most often don't know the correct result when we compute something in research, so we never know if a result is correct. Perhaps the most visible failures that we can easily observe are reproducibility failures. If a computed result is not reproducible, we don't really know what has been computed (because computation, being deterministic, *is* reproducible). In most cases, we don't know what has been computed because we don't know which precise software stack we have been running. And we don't know the precise software stack because the software stack is too complex to be easily described and reconstructed. As I explain under Computational reproducibility, that problem is solved in principle, and increasingly also solved in practice, by delegating the management of software stacks to computers. Except that… the management software for software stacks is pretty complex as well!

The ultimate solution to the problem of cheap complexity is, of course, paying the higher price for simpler technology. In the not-so-distant past when complexity wasn't cheap, innovation, whether in science or technology, was followed by a phase of consolidation. New scientific knowledge, once it

became reliable, was reformulated in ever simpler and more compact ways. Compare, for example, Isaac Newton's "Principia Mathematica" to a modern explanation of the same theory in a textbook for physics students. The modern version is more compact and much easier to understand. The same process happens in technology, for example when breadboard prototypes for electronic circuits get redesigned into printed circuit boards and then integrated circuit chips for industrial mass production. Could this approach work for software and digital scientific knowledge as well? We won't know before we try!

Code over data

A computation is, from a bird's eye view, the application of code to input data, producing output data. Usually the code is a tool to manipulate or transform data. Computer users tend to care more about their data than their tools. They want to write a letter, not use Microsoft Word. They want to watch a movie, not start VLC. They want to simulate the behavior of a protein, not run GROMACS. Computing *should* be data-centric, for most use cases.

Reality is quite the opposite. It's code over data everywhere you look. Your phone shows "apps", meaning tools. They work on data that is handled opaquely. You don't really know where it is, what it represents, who can access it. All you get is the view on the data that the app shows you.

On the desktop, it's very similar. You probably know the name of your word processor, but not the name of the file format it uses to store the data. You probably cannot name other software that could use that same file format. Contrary to phones, desktop systems let you see and manipulate the files that contain your data, but offer you only very generic operations (copy, delete), or running "the app" that effectively owns the data.

In the university classroom, we see the same predominance of code over data. Whether you look at the titles of computer science classes or textbooks, you see a lot more on software or algorithms than on data models.

Computing is obsessed with tools, which seem more important than the tasks they are designed to perform and the information that they process. Computer-aided research is inheriting this obsession. More and more papers cite software (which is good!) but don't describe in sufficient detail what the software does (which is bad), nor how the input and output data are structured, making it difficult for readers to examine the work in detail, possibly using different software.

Another illustration is the use of computational notebooks in data science. Data science is about data, but notebooks are about code. If you want to show and explain data in a notebook, you have to write the code for this yourself. A computational medium for data science would put the data in the primary focus of attention, not the code that loads and processes the data.

Computational disempowerment

Many scientists today find themselves depending on computational tools that they do not fully understand and that they are unable to modify to fit their needs. Their research is thus constrained by their software tools, and sometimes that constraint only leaves the choice of doing research badly or not at all.

The paper Between Scripts and Applications: Computational Media for the Frontier of Nanoscience.md describes this situation with the very fitting term "computational disempowerment", which I am happy to borrow here.

My current view of a way out of this situation consists of two major ingredients:

1. The introduction and use of computational media as the main human-computer interface in science.
2. The creation of a digital infrastructure for science, consisting of software components and services that are designed, developed, and maintained explicitly for scientific research, with appropriate governance mechanisms.

Computational environment

Although now frequently used in the context of computational reproducibility, the term "computational environment" is rather recent. It usually refers to the software infrastructure required to run a specific computation: operating system, compilers, interpreters, support libraries, etc. Sometimes it is used in a wider sense that includes the actual hardware on which a computation runs.

Computational environments are the main focus of research and development around computational reproducibility because now that the actual preservation of code is a solved problem (see Software Heritage), the documentation and preservation of computational environments is the biggest unsolved one.

A computation, as defined by Alan Turing in his famous 1937 paper that introduced the Turing machine, is a transformation of a a string of input symbols into a string of output symbols via the application of well-defined rules. In today's computers, the symbols are bits (0 or 1), so a computation is the transformation of an input bit sequence to an output bit sequence, the rules being roughly the processor's instruction set.

Human computer users like to divide the input bit sequence into *code* and *input data*, the idea being that "active" code operates on "inert" data. While this is very useful distinction in terms of computer applications, it belongs to the realm of interpretation. For the computer, it's all bits. One way to see that code vs. data is a matter of interpretation is to consider an alternative one: the format of the input data can be seen as a formal language, for which the code reading the data is an interpreter. In this interpretation, everything is code.

Next, human users like to divide the code into a *program* that is run and the *environment* that supports the program. This is an even more arbitrary dividing line. It comes down to a distinction between "the code that I

care about" (the program) and "the code that I don't care about" (the environment). The computer cares equally about each bit in its input.

A computation is reproducible if the full input bit sequence is preserved and can be replayed: input data, the code that the authors of the computation care about, but also the code they don't care about. Is it really surprising that all the trouble comes from the code we don't care about? Starting to care about one's computational environments is the key step to improving computational reproducibility.

Today, preserving and replaying computational environments bit by bit has become relatively easy. Container images, via management tools such as Docker or Singularity, do the trick. Unfortunately, that has proven insufficient for making computer-aided research reproducible. Container images ensure reproducibility for computers (bits, it's all just bits!), but not for humans. For a human user, a container image proves exactly one fact: that there exists a computer program that produces a given result. Which, for a digital result, is a rather trivial fact. Human users need to *understand* the computation at the level of their *interpretation* of inputs, code, and outputs. Human users need *source code*.

Computational environments being digital artifacts, created via computation (see "The dual nature of software"), they always have source code, but we don't yet care enough about environments to (1) preserve this source code (it's often a few lines of shell commands typed in by hand) and (2) make it reproducible. The good news is that this is possible, and even supported by existing tools such as Guix. As a simple example, the following two source code files fully define a computational environment, bit by bit:

File "manifest.scm":

```
(specifications->manifest
  (list "python"
        "python-matplotlib"
        "python-numpy"))
```

File "channels.scm":

```
(list (channel
        (name 'guix)
        (url "https://git.savannah.gnu.org/git/guix.git")
        (branch "master")
        (commit
```

```
"35b176daf1a466f136f0b77c03de78f482a30702")))
```

Given those two files, and a computer with an installation of Guix, the environment can be re-created using the command

```
guix time-machine -C channels.scm -- guix shell -m manifest.scm
```

On two computers that have compatible processor instruction sets, this will create environments that are identical, bit for bit.

Why does the source code of an environment consist of two files? Because the creators of Guix decided to separate the list of the software building blocks in the environment (`manifest.scm`) from the list of their precise versions (`channels.scm`), in order to make it easier to vary the versions and thus test for the robustness of the computation. It's a minor technical design decision.

Are you ready to start caring about your computational environments? Then take a serious look at Guix.

Recommended reading: - Dealing with software collapse (preprint) - A Dream of Simplicity: Scientific Computing on Turing Machines (preprint)

Computational media

Media are substrates for encoding information. They can serve many purposes, the most common ones being communication, archival, or interfacing with tools. Printed paper is a medium. An abacus is a medium. The telephone is a medium. Television is a medium.

Digital media are media defined by software, amenable to processing with a computer. MP3 audio files are digital media, as are Word documents, PNG images, and many others.

Computational media are digital media that can encode computation among other information. Spreadsheets (e.g. Excel and its many clones) are probably the most well-known example. Game engines are another example, well known as well though most people are probably unaware of their capacity to encode computation.

Programming languages can be seen as a degenerate form of computational media, which can encode computation but nothing else. I consider the predominance of programming languages in today's computing technology, and in particular the widespread idea of "general purpose" programming languages, a sign of the immaturity of this technology. It goes along with an exaggerated focus on code over data.

Computational science suffers from this exaggerated focus as well. The core entities of science are observations, models, and the relations between them. Computational media for science should encode these entities, and let scientists explore and refine them. Tools, such as computers and software, are merely means to this end. Astronomy is about stars and galaxies, not about telescopes. Particle physics is about elementary particles, not about particle colliders. Biology is about living organisms, not about microscopes or test tubes. The computational branches of these disciplines should also be about entities in nature and the models we make of them, not about

computers and software.

What could a computational medium for science look like? I'll stick to what I know: physics and chemistry, and in particular biophysics. My current idea of a computational medium for these disciplines takes the shape of a Wiki, a collection of cross-references pages, much like Wikipedia. Some pages in this Wiki describe entities, such as proteins. Other pages describe *models* for entities in nature, such as the elastic network model for proteins. Yet other pages describe *observations* on these entities, i.e. typically the outcomes of experimental studies. Below this surface of human-readable narratives, the pages contain machine-readable representations of everything, made explorable and refineable by suitable tools.

Much of the technology required for such a computational medium already exists. We have Wikis, and we have the semantic Web as a backbone for encoding relations in a machine-readable way. The Nanopublications project (and others!) illustrates how the semantic Web can be used to encode relations between observations and models, as well as statements *about* observations and models, such as claims or evidence as part of a discourse graph. We also have good digital representations for observations, though the multitude of data formats makes them hard to manage. What's lacking do far is a suitable representation of models - that's what I hope to achieve with digital scientific notations.

Computational tools will still exist, of course. But instead of the scientist going to the tool and set it in motion, loading data from files etc., the tools will be an extension to the computational medium. One possibility is to run tools from code cells inside a Wiki page, which then functions much like a computational notebook does today.

Recommended reading:

- Beyond programming languages, by Terry Winograd. A 1979 paper whose vision has not yet been realized.
- The computer revolution hasn't happened yet, by Alan Kay. A recorded talk from 1997, but it hasn't happened in the following 25 years either.
- Software as Computational Media, by Clemens Nylandsted Klokmose (video, recorded keynote speech at the conference LIVE'21)
- Computational science: shifting the focus from tools to models, by yours truly.

Computational notebook

An electronic document embedding a computation into a narrative.

Computational notebooks differ from literate programming in documenting a *computation*, i.e. code with all required input data, whereas literate programming documents *programs*, i.e. code designed to accept varying input data. It is the focus on fully specified computations that makes it possible to include intermediate and final results. On the other hand, this same focus means that notebooks can only deal with the surface layer of a computation. The library code called from that surface layer remains inaccessible to the reader of a notebook.

Computational replicability

This page is empty

Computational reproducibility

In the context of computer-aided research, reproducibility refers to the possibility to re-execute a computation and check that the results are identical. It differs from computational replicability, which is about the robustness of results under minor changes in the software. Unfortunately, terminology hasn't settled yet and some authors use these two terms in exactly the opposite way.

Computational reproducibility became a subject of debate because its practical impossibility came as a surprise. Computations are supposed to be deterministic. $2 + 2$ is 4 today, as it has been for centuries, and we have little doubt that the result will be the same 100 years from now. Computations done by a computer usually perform a huge number of such steps, but that shouldn't make a difference: 1 million deterministic steps still make for a deterministic result. The practical experience of scientists using computers is quite the opposite: it is the rule rather than the exception that re-running someone else's computation leads to a slightly different result.

This apparent mystery has a simple explanation. If you re-do a computation twice in succession on your computer, you will get the same answer (ignoring special cases such as random number generators or parallel computing). If you re-do a computation a day later on the same computer, you will also get the same answer, most of the time. In fact, if you get a different result, then something has changed on your computer in between. Most probably, you have updated some software, possibly without being aware of it. And when you re-do someone else's computation on your computer, you are actually transferring a small component of one software system into a different software environment - yours. In other words, when a reproduction

attempt for a computation yields a different result, the by far most frequent explanation is that the computations were subtly different.

So how can it happen that two people who are convinced of doing the same computation are actually doing different ones? The two main culprits are the complexity and the opacity of today's software stacks. What you think of as "the software" you are running is really just the tip of the iceberg. Between that code and the processor that is doing the work inside your computer, there are many layers of software that have an impact on the results you will get. Obtaining a full description of those layers is very difficult to impossible on most of today's computing platforms. Transferring all of them to another machine is even more difficult, and often impossible.

A case study

An interesting case study from chemistry was published in 2019 by Neupane *et al.*. It starts from a 2014 publication of a computational protocol for obtaining molecular structures from chemical shifts measured by NMR (don't worry if you don't understand what this means). The supplementary material for that publication contains two Python scripts that are essential parts of the protocol. What Neupane *et al.* discovered is that these scripts access the data files they process in a way that tacitly assumes a behavior specific to the Windows operating system. When run under Linux, the scripts can read the data files in a wrong order, depending on circumstances that are outside of the scripts' control. As Neupane *et al.* note:

> This simple glitch in the original script calls into question the conclusions of a significant number of papers on a wide range of topics in a way that cannot be easily resolved from published information because the operating system is rarely mentioned.

Yes, your operating system is part of the software that you are running. As are, in the case of this specific example, the Python interpreter, the Python libraries it depends on, and a much larger number of nearly invisible libraries that Python itself depends on. All of these software components are regularly updated by their authors, with the goal of fixing bugs, adding features, or improving performance. This explains why the software environment on your computer changes all the time, and why two different computers are highly unlikely to have the same software environment.

The two Python scripts that are the focus of this case study have been fixed

in the meantime, but I suspect that many scientists still have and use the original ones.

Is there a way out?

Yes. Computational reproducibility is, in principle, a solved problem. There are well-understood techniques to document a software assembly completely and precisely, in such a way that it can be transferred to a different computer. Not just any computer though, it has to be sufficiently similar to the original one, and in particular use the same type of processor (which, in a way, is also part of your software stack). Better yet, there are freely available tools that manage software (and computations) reproducibly for you: Nix and Guix. A key insight behind these two tools is that every computation on a modern computing system is actually a staged computation, with reproducibility of the last stage (the one we most care about) requires the reproducibility of all prior stages.

This isn't the end of the story though. The existence of support tools that guarantee computational reproducibility is only the first step. In terms of user-friendliness, these tools still leave a lot to be desired. And most research software has not yet been integrated into their management scheme, and for some software this is nearly impossible. In particular, only Open Source software can be managed reproducibly, because controlled compilation of the source code is a crucial step. And that also means that the only operating system that can be supported is Linux.

A few years ago, a frequently discussed question was "is computational reproducibility possible?". Today it is clear that the answer is "yes". Now the question is how much reproducibility is worth to researchers. Enough to support the development of Nix and Guix? Enough to invest into learning how to use them? Enough to abandon proprietary software, including the popular operating systems Windows and macOS? Time will tell. Computational reproducibility is no longer a technical issue, it's a social one.

Further reading: - Is reproducibility practical? by Ludovic Courtès

Computational science

This page is empty

Computer-aided research

Scientific research in which computers and software are essential parts of the research workflow. The term computational science is usually reserved for research in which computation is the dominant tool. Computer-aided research is a much wider category, including most of today's experimental research that relies on computational data processing in various stages of the overall workflow.

Content-addressable storage

This page is empty

Convivial software

🌱

In 1973, Ivan Illich published a book entitled "Tools for conviviality", in which he argues for *convivial* technology, which he defines as technology that empowers people to do what they want, rather than constraining their choices. An important aspect of convivial technology is that people must be able to adapt it to their needs and wishes.

Convivial software is thus software that its users are not only allowed to modify (that would be Open Source), but which is designed and written with the explicit goal of facilitating modification by its users. For now, it is vision rather than an established practice. Modifying software always requires a higher technical competence than merely using it. Convivial software should both reduce the barrier between user and developer and provide guidance for the user motivated to overcome it. Ideas for reaching this ideal include explainable systems, malleable systems, re-editable software, and the general quest for simplicity.

Further reading: - Convivial Computing in the Damaged Earth Catalog - Convivial design heuristics for software systems, by Stephen Kell

Conway's law {#Conway's-law}

Melvin E. Conway (1967):

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Data science

According to Wikipedia,

> Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from noisy, structured and unstructured data, and apply knowledge and actionable insights from data across a broad range of application domains.

This sounds very modern, but it's really only the label that is recent. Researchers such as Apollonius, Hipparchus, and Ptolemy, practiced data science about 2000 years ago.

The focus of interest of these early researchers was a topic that had kept humanity busy for quite a while already, all over the world: the motion of heavenly bodies. The main motivation was making predictions for the near future. The configuration of the stars and planets was widely believed to have an impact on human affairs (a belief we call astrology today), so knowing them in advance was of obvious interest. They had astronomical observations at their disposal, but numbers alone are not sufficient to make predictions. You also need a model for extrapolating the numbers to the future.

The tool that Apollonius, Hipparchus, Ptolemy, and probably others, developed and improved to near perfection was epicycles: a model for the orbit of a heavenly body consisting of a superposition of circles, with each circle's center moving along a bigger circle's circumference. Epicycles are similar in spirit to Fourier series. Any periodic orbit can be described as a superposition of circular motions. Given enough data, one can fit an epicycle model and make predictions. But since the epicycle model does not contain any physics, it doesn't come with any safeguards against mistakes. Epicycles can equally well describe real and completely unrealistic orbits, and therefore

the quality of the data is very important.

Today's data science works much the same. Very general models, such as neural networks, are fitted to large datasets via machine learning techniques, and then used to make predictions. Again the models contain very few assumptions about underlying laws of nature. They are by design very general (see e.g this visual proof that neural networks can compute any function) in order to capture any kind of regularity in the input datasets. As for epicycles, data quality is important, which is why data scientist invest a significant effort into cleaning up the raw data they work on.

Aside from the obvious technological aspects and the associated change of scale in the size of datasets, the main improvement of today's data science on epicyle models for orbits is even more generality. Early astronomers had periodicity baked into their models from the start. Neural networks (and other models used in data science) could predict the motion of heavenly bodies with even less theoretical input. However, it is important to realize that every model imposes *some* a priori assumptions, even if, as in the case of neural networks, these assumptions are not fully understood and therefore not formalized. Seen in this light, the improvement of modern data science over epicycles is gradual rather than fundamental. It is also interesting to note that neural network research has (re-)discovered the benefits of more specialized models, as e.g. in convolutional neural networks.

Adopting an historical perspective, data science turns out to mark the *beginning* of scientific disciplines rather than their refinement. It permits the very first step from raw observations to a description of regularities in the form of empirical models. Connecting these regularities to more fundamental principles that are already known, or even discovering *new* fundamental principles as in the case of Newton's laws for celestial mechanics, can only happen afterwards, via the construction of explanatory models.

Decentralized science

Under the label of "decentralized science", several communities are exploring the use of blockchains and related cryptographic technology for structuring scientific research. One focus is on funding, using ideas from crypto-currencies. Another focus is publication with better provenance tracking.

While I am optimistic about the long-time potential of these efforts, I don't expect any real progress to be made as long as these communities remain attached to blockchains and their crypto-currencies. A big practical issue with blockchains is that they are, by design, largely decoupled from the real world. The interface between people, institutions, and blockchains are anonymous accounts, of which each actor can create an arbitrary number. Anonymity and cheap accounts means that nothing prevents people from reviewing their own publications, using a second account, or voting multiple times on some issue. Anonymity also means that actions on the blockchain are unrelated to a person's scientific reputation.

Further reading:

- A Guide to DeSci, the Latest Web3Movement, by Sarah Hamburg

Digital Garden

A digital garden is a small ecosystem of interrelated documents that its curator tends to with regular updates and revisions. It differs from a blog, which is a stream of finished-then-published documents. A digital garden can be considered a special case of a Wiki that is curated by a single person or a small team, in contrast to open-to-all collaborative works such as Wikipedia.

Some recent entries start with an icon indicating its level of maturity, an idea taken from Maggie Appleton's Digital Garden:

- 🌱 Seedling: New ideas and concepts that may or may not turn out to be useful.
- 🪴 Plant: Entries that have proven useful and are linked to from other parts of the garden. The text has room for improvement and is likely to evolve.
- 🌲 Evergreen: Notes or essays that are fully worked out. They may receive minor updates or correction from time to time but are otherwise stable.

These categories are very much 🌱 as I just started to use them, so they may well evolve. Note that many entries do not have a maturity indicator, because they mostly serve as keywords for cross-referencing, or as anchors that provide links to external resources.

Recommended reading on digital gardens:

- The Garden and the Stream: A Technopastoral by Mike Caulfield
- A Brief History & Ethos of the Digital Garden by Maggie Appleton

Digital infrastructure

Hardware, software, and services that allow scientists to do computer-aided research focusing on the science rather than on computing technology.

Today's digital infrastructure for science is 1. insufficient, in leaving many needs unsatisfied 2. mostly borrowed from domains of activity whose needs are very different from research (such as enterprise software) 3. not under control of the research community

Recommended reading:

- Decentralized Infrastructure for (Neuro)science, by Jonny Saunders

Digital scientific notation

A scientific notation is a convention for encoding scientific information using symbols. The best known example is mathematical notation. The goal of a scientific notation is to represent scientific knowledge in a way that humans can easily comprehend and manipulate. While in principle a mathematical equation could be replaced by an equivalent statement in plain language, the more concise equation is faster to read (assuming a trained reader) and allows manipulation by formal rules (such as "add the same term to both sides").

A *digital* scientific notation is a scientific notation that can be processed by both humans and computers. A machine readable notation is necessarily a formal language and thus has a well-defined unambiguous syntax in addition to some useful level of well-defined semantics.

There are many formal languages designed for representing scientific information. An example is the Systems Biology Markup Language (SBML). Most of them do not qualify as digital scientific notations, because they are designed to be used by software but not for communication between humans.

There are also formal languages that are designed to be read and written by humans, in addition to computers. Programming languages are the most prominent examples. In scientific computing, programming languages are routinely used to represent scientific knowledge as program code. In particular, computational notebooks embed code written in high-level programming languages such as Python or R into a narrative, much like mathematical notation is used in traditional scientific publications. However, programming languages fill the role of scientific notations rather poorly, in particular because they cannot express anything other than executable algorithms.

Digital scientific notations are *not* computational tools, but parts of the communication interfaces between scientists and their computational tools.

In particular, they permit scientists engaged in computer-aided research to discuss computational models and methods in a way that ensures conformity between the human narratives and the computations.

Further reading: - Scientific notations for the digital era (on arXiv) and a comment on it by Mark Buchanan in *Nature Physics* - Scientific communication in the digital age (in *Physics Today*) - Leibniz in four minutes, a short video demo of Leibniz that illustrates the role of digital scientific notations.

Discourse graph

See the Discourse Graph Starter Kit

Donald Knuth

This page is empty

Elastic network model

This page is empty

Emacs

This page is empty

Empirical model

The first type of scientific model that people construct when figuring out a new phenomenon is the *empirical* or *descriptive* model. Its role is to capture observed regularities, and to separate them from noise, the latter being small deviations from the regular behavior that are, at least provisionally, attributed to imprecisions in the observations, or to perturbations to be left for later study. Whenever you fit a straight line to a set of points, for example, you are constructing an empirical model that captures the linear relation between two observables. Empirical models almost always have parameters that must be fitted to observations. Once the parameters have been fitted, the model can be used to *predict* future observations, which is a great way to test its generality. Usually, empirical models are constructed from generic building blocks: polynomials and sine waves for constructing mathematical functions, circles, spheres, and triangles for geometric figures, etc.

The use of empirical models goes back a few thousand years. As I have described in in a blog post, the astronomers of antiquity who constructed a model for the observed motion of the Sun and the planets used the same principles that we still use today. Their generic building blocks were circles, combined in the form of epicycles. The very latest variant of empirical models is machine learning models, popular in data science, where the generic building blocks are, for example, artificial neurons. Impressive success stories of these models have led some enthusiasts to proclaim the end of theory, but empirical models of any kind and size are really the beginning, not the end, of constructing scientific theories around explanatory models

The main problem with empirical models is that they are not that powerful. They can predict future observations from past observations, but that's all. In particular, they cannot answer what-if questions, i.e. make predictions for systems that have never been observed in the past. The epicycles of

Ptolemy's model describing the motion celestial bodies cannot answer the question how the orbit of Mars would be changed by the impact of a huge asteroid, for example.

Today's machine learning models are no different. A major recent success story is AlphaFold predicting protein structures from their sequences. This is indeed a huge step forward, as it opens the door to completely new ways of studying the folding mechanisms of proteins. It has also already become a powerful tool in structural biology. But it is not, as DeepMind's blog post claims, "a solution to a 50-year-old grand challenge in biology". We still do not know what the fundamental mechanisms of protein folding are, nor how they play together for each specific protein structure. And that means that we cannot answer what-if questions such as "How do changes in a protein's environment influence its fold?", because the only variation in its inputs that AlphaFold has been trained on is the protein's amino acid sequence.

Empty page

There are many empty pages in this collection, and you may wonder why.

One reason is that this digital garden is work in progress. When I work on a page, I often insert links to pages that I intend to write, but haven't written yet. So you see an empty page. If you come back later, you may find some real content there. So… come back often.

The second reason is that empty pages are useful link targets, due to the backlink feature in my digital garden. At the end of each page, you see a list of other pages that link to the current one. Empty pages thus fulfill the role of a subject index in a traditional book: they help you find where some topic is discussed.

Epistemic diversity

Epistemic diversity refers to the coexistence of multiple perspectives, research methodologies, theories, and models in a scientific discipline. Ideally, these different points of view participate in all debates, enriching and critiquing each other. From an evolutionary perspective, epistemic diversity is important to prevent scientific inquiry from getting stuck, with one point of view dominating a discipline and criticism of this point of view becoming impossible to get heard. History has shown that even the most successful scientific theories lose their status of "best known description" one day. Newtonian mechanics was considered an absolute Truth for two centuries, before relativity and then quantum mechanics revealed its limitations and degraded it to a practically very useful, but no longer universal reasoning framework about our universe.

Up to here, I expect that most scientifically educated readers would nod in agreement. And yet, epistemic diversity is threatened by two foundational ideas of the industrial age: automation and standardization. Both are important for scaling up production processes, increasing productivity and lowering costs. Since the 1950s, science has increasingly be considered a support for economic and thus industrial development, in addition to its traditional role of improving our understanding of the world. Perhaps the most visible symptom is the now common practice of evaluating scientists by productivity criteria. One way to be more productive is to adopt standard practices, and to apply one's acquired competences mechanically to as many specific situations as possible.

This quest for productivity has left its marks in the Open Science movement, in particular in the FAIR principles. Whereas Findability and Accessibility are uncritical, Interoperability and Reuse are about standardization and productivity. Making data interoperable with existing software and conventions often requires cutting it down, removing information that doesn't fit the

imposed storage formats or protocols. Reusing data implies accepting the values and motivations behind their collection.

For software, reuse implies adopting a potentially large set of explicit and tacit hypotheses made by its developers, and living with its mostly unknown but inevitable bugs. Bugs and the difficulty of fixing them is usually cited as a reason *for* reusing software as much as possible, mutualizing the effort of maintenance among a larger community. That is fine if the goal is to eliminate as many bugs as possible. But in science, what really matters is to reduce the *impact* of bugs on results, and from that point of view, it's bugs going unnoticed that are the most serious problem. With diverse software and thus diverse bugs, the chance of them going unnoticed is smaller. Provided, of course, that the results from diversely buggy software packages are confronted, and the causes of any differences explored, which is not commonly done today. Computational replicability matters.

In a world of finite resources, there is no obvious solution to the tension between doing the best possible job on one project or tool on one hand, and exploring multiple directions in the interest of epistemic diversity on the other hand. But we should at least remain aware of the tension.

Further reading: - Open Science and Epistemic Diversity: Friends or Foes? by Sabina Leonelli

Epistemic opacity

Epistemic opacity is philosophers' jargon for describing processes and mechanisms that are much easier to use than to understand. If you do use such a process, you don't really know what you are doing.

Consider a somewhat complex computation, but one which is still doable by hand. Computing the correlation of two 100-point discrete signals, for example. Now consider the following ways of getting to the result:

1. You do the computation by hand, yourself.
2. You ask one of your students to do the computation for you (assuming you are an academic, of course!)
3. You write a computer program do to the computation, then run it.
4. You run a computer program written by someone else.

From top to bottom, epistemic opacity increases, making a huge jump between number 3 and 4. If you do everything yourself, by hand, you will likely insert checks to catch mistake, because you know that everybody makes mistakes in lengthy computations. Probably you also make a drawing of the result as you go on computing points. And since you have some intuitive notion (assuming you are familiar with correlation functions of course) of what the result will look like. The computation is under control.

Delegating the job to a student makes it less transparent, but you can still ask the student questions, and look at the student's worksheet. And since the student learned the methods from you, the worksheet has a chance of making sense to you.

Writing a program for the job is similar. You write, proof-read, and most of all test the program, performing checks similar in spirit (though different in details) from the checks in the manual computation. But it's much easier to be superficial about testing: you will get a result even if you don't.

Running someone else's program is a very different story. You can do that even if you don't know what a correlation function is! The program is an opaque machine into which you stuff data and then take new data out at the other end. If you do understand the program's task, you will still spot significant mistakes in the result. But in the manual computation, you would also spot mistakes in the intermediate results, which in the automatic computation never become visible.

This is an important and not sufficiently discussed problem with reusable scientific software. It's of course *efficient*, in the sense of productivity, to re-use someone else's software to get a job done. But it severely limits your understanding of the result, and your capacity to verify that the result corresponds to the scientific method you wish to implement.

Evergreen

See Digital Garden

Experimental reproducibility

This page is empty

Explanatory model

In contrast to empirical models, explanatory models describe the underlying mechanisms that determine the values of observed quantities, rather than extrapolating the quantities themselves. They describe the systems being studied at a more fundamental level, allowing for a wide range of generalizations.

A simple explanatory model is given by the Lotka-Volterra equations, also called predator-prey equations. This is a model for the time evolution of the populations of two species in a preditor-prey relation. An example is shown in this plot (Lamiot, CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0, via Wikimedia Commons):

An empirical model would capture the oscillations of the two curves and their correlations, for example by describing the populations as superpositions of sine waves. The Lotka-Volterra equations instead describe the interactions between the population numbers: predators and prey are born and die, but in addition predators eat prey, which reduces the number of prey in proportion to the number of predators, and contributes to a future increase in the number of predators because they can better feed their young. With that type of description, one can ask what-if questions: What if hunters shoot lots of predators? What if prey are hit by a famine, i.e. a decrease in their own source of food? In fact, the significant deviations from regular periodic change in the above plot suggests that such influences from the environment (everything not explicitly represented in the model) are quite important in practice.

One of the biggest success stories in the history of science is the shift from the empirical models for celestial mechanics (Ptolemy's geocentric epicycles, Kepler's heliocentric ellipses) to Issac Newton's explanatory differential equations. Newton's laws of motion and gravitation fully explained Kepler's

elliptical orbits and improved on them. More importantly, they showed that the fundamental laws of physics are the same on Earth and in space, a fact that may seem obvious to us today but wasn't in the 17th century. Finally, Newton's laws have permitted the elaboration of a rich theory, today called "classical mechanics", that provides several alternative forms of the basic equations (in particular Lagrangian and Hamiltonian mechanics), plus derived principles such as the conservation of energy. As for what-if questions, Newton's laws have made it possible to send artefacts to the moon and to the other planets of the solar system, something which would have been unimaginable on the basis of Ptolemy's epicycles.

In the past, almost all explanatory models took the form of mathematical equations, and in particular differential equations. This is likely to change in the digital era. Agent-based models are an example of "digital native" explanatory models. There is, however, a formal characteristic that is shared by all explanatory models that I am aware of, and that distinguishes them from empirical models: they take the form of specifications.

FAIR

Findability, **A**ccessibility, **I**nteroperability, and **R**euse of digital assets

See the FAIR principles home page

FLOSS

Free/Libre and Open Source Software

The two terms refer to different perspectives on why software source code should be made public. Open Source focusses on mutualizing the development effort, whereas Free Software adds an emphasis on granting specific freedoms to *users* rather than developers.

Further reading:

- FLOSS and FOSS, by Richard Stallman

Floating-point arithmetic

This page is empty

Force field

See Wikipedia

Formal language

This page is empty

Formal system

Wikipedia defines a formal system as:

> A formal system is an abstract structure used for inferring theorems from axioms according to a set of rules.

This is a rather narrow definition in the context of mathematics. I use the term in a wider sense as any abstract structure used for deducing outputs from given inputs using precise rules. For example, I consider Newton's laws of motion a formal system for computing the trajectories of point masses from their initial positions and a description of their interactions. In this wider sense, formal systems are the symbolic equivalent of machines, and computers have turned this metaphor into a physical reality. Indeed, every computer program implements the rules of some formal system.

Formal systems are usually constructed with a specific intended meaning for its rules, inputs, and outputs. However, the meaning comes from the embedding context, e.g. the scientific model in which the formal system is used. A formal system by itself is just symbols and rules for manipulating them. It is up to the user of the formal system to verify that the rules conform to their intended meanings. Stated in the jargon of computer programming: it is up to the user to verify that a program does what it is expected to do.

This sets a limit to the usefulness of large formal systems in scientific models: a formal system (and in particular a computer program or a trained neural network) that is so complex that examining and verifying it becomes infeasible has a rather limited utility in science. While it can be used to make testable predictions, it remains at the level of an empirical model. To become the foundation of more powerful explanatory models, formal systems must be verifiable to the point that we can be reasonably certain to know their limits of validity. Unfortunately, today's software technology makes it easier to build large and complex formal systems (programs) than small and simple

ones that scientists can explore in detail and thus understand.

Formal vs. informal

What characterizes formal systems (as well as formal languages, methods, etc.) is the existence of a closed symbolic universe in which precise rules apply. Formal systems require no reference to a context for interpreting their rules or making judgments of any kind. The symbols that are processed by a formal systems are of course interpreted in some application context, but this happens outside of the formal universe.

Informal methods, in contrast, use less precise references and rules, whose exact interpretation is context-dependent. The words of a (human) language are a good example. They usually take different meanings in different contexts, but those meanings are nevertheless somehow related.

A computation, as defined e.g. by a Turing machine, is a perfectly closed formal system. The input symbols fully determine the output symbols. An algorithm, and its implementation as a computer program, are somewhat open in having an interface to the outside world, from which they accept input data. However, the format and computational semantics of the input data is defined by the formal system, not the outside world. The more input the program takes from the outside world, the more its output is shaped by the outside world, making the computation less formal. An extreme case is a machine learning system, whose output is determined much more by the input data used in the training phase than by its formalized aspects (such as artificial neurons), which are intentionally chosen to constrain the output only weakly. This is why machine learning has been so successful in describing many aspects of the world, but it is also the reason why it creates very little explanatory power.

Scientific models are inherently context-dependent and therefore informal. Formal systems are tools for reasoning that are always embedded into a larger informal model that links the formal system's symbols to concepts

about the physical world. However, only the formalized part of the model can make it explanatory.

Formalization

Formalization is a process in which concepts and their relations are made more precise by the introduction of formal systems. It can be seen a specific technique of conceptual engineering.

Formalization is widely applied in the construction of scientific models, but its importance varies widely between scientific disciplines. The most heavily formalized discipline is physics, to the point that one could almost *define* physics as the study of nature using formalized models. Other disciplines are less attached to formalization, but more formalized models are generally considered superior to less formalized models, and in particular the special case of quantification is almost universally seen as desirable in science today.

The prestige associated with formalized models creates the risk of premature formalization, i.e. the introduction of formal systems that do not faithfully implement the original informal model and/or the available observations, but leave a superficial impression of precision.

Even though formal systems are often presented as the central part of a scientific model, in particular in physics textbooks, the model is always more than its formal system(s). At the very least, each model has an informal part that describes how the formal expressions relate to observations. Newton's laws of motion, for example, require a definition of concepts such as time and force in terms of observable properties to make a complete scientific model.

In the past, formalization was limited to simple formal systems that could be constructed and verified by humans without machine support. This was a laborious task that typically involved entire communities for many years. Formal systems in scientific models thus tended to be few, simple, and well examined. In the digital era, formalization happens, often without much thought, whenever a scientist writes a program to predict or process observations. Since computer programs are notoriously difficult to understand, if

only due to the complexity of today's software stacks, we see the opposite phenomenon of numerous complex formal systems that are only superficially examined and verified.

Can we have both the level of verification and transparency of the good old days *and* today's ease of constructing new formal systems using computers? I believe we can. The two key ingredients that I see are:

1. Notations for formal systems that are much more lightweight than software source code, and integrate well with the narratives that define the informal aspects of scientific models. I call them Digital scientific notations.

2. Support tools for managing the formalization process, both at the level of individual scientists focusing on a single aspect, and at the level of research communities working towards consensual models. My ideas for this part remain vague, but I suspect that computational media for science will be an important ingredient. And maybe also static type systems.

Free software

This page is empty

Git

This page is empty

Glamorous Toolkit

A toolkit for making software systems explainable. Watch that talk, and then see the Glamorous Toolkit Web site for further information, and for downloading a copy for your own exploration.

Guix

See the Guix Web site

Human-computer interaction

This page is empty

Implementation details

This digital garden is a TiddlyWiki extended with the Markdown and Krystal plugins.

The pages are written as Markdown files (plus optional metadata), which I write and edit using GNU Emacs. They are stored in a repository on Codeberg, which also contains a Python script and a bash script that generate the TiddlyWiki by adding the pages to a template file (which is also in the repository).

In search of simplicity

There are various ongoing attempts to create supporting technology for making simple software that empowers individuals and small groups who are not professional software developers. To the best of my knowledge, there are no commercial or large-scale activities in this sector today (although there were in the past). All of the projects I am aware of are made by people scratching their own itches. As a consequence, they are small projects by small communities, and not very visible.

None of the projects listed below of is about scientific software. Moreover, many of them take an extremist position on some aspect, which may well be incompatible with the requirements of scientific computing. I do not mention these projects here because I think scientists should adopt them. I see them as (1) proof that simplicity in software is possible if you make it a high enough priority and (2) a source of ideas for how to make software more accessible. The only discussion of related issues in the context of academic research that I am aware of is an issue of "digital humanities quarterly" dedicated to "Minimal Computing".

The following alphabetically ordered list makes no claim to being exhaustive. It contains projects that I somehow discovered and which looked interesting enough to spend a few hours investigating them.

Freewheeling apps is a small software stack for personal and easy-to-modify applications with graphical user interfaces that are portable across devices and platforms. It is built on top of LÖVE, a game development framework for the programming language Lua. There are a few examples to get started. Lua is an interesting project in its own right from the simplicity point of view, because in spite of being 30 years old (just two years younger than Python), and used in various application domains, it has largely escaped the fate of creeping complexification.

Gemini is a minimalist Web-like technology stack, replacing both the HTTP protocol and the HTML markup language by much simpler equivalents. The goal of Gemini is twofold: create a simple technology stack that is easy to implement, and eliminate features seen as undesirable by its community, such as tracking or advertising, by making them technically impossible.

The programming language Hare, currently in an early stage of development, has the design goal to be simple, stable, and robust. Its plan for avoiding complexification is radical: once the specification reaches version 1.0, it will be frozen forever. The stated goal is to become a 100-year programming language.

The IndieWeb is a community dedicated to simple Web sites based on simple standards and simple technology, outside of corporate control. It provides support for people wishing to build and host their own Web sites, using a manageable subset of the official Web standards complemented by conventions such as microformat.

KolibriOS is an operating system for old and/or low-resource computers. It does not seem to have an explicit focus on simplicity, but requires such a focus indirectly though resource restrictions.

Minimacy is a small functional programming language, resembling OCaml with its own virtual machine and standard library. It runs inside a host operating system, but is meant to be usable without any operating system in the long run. The goal is to have a technology stack that a single software engineer can understand in its entirety, and build applications on top of it without fearing bad surprises such as software collapse or security issues.

Smalltalk 80 is a programming system designed in the 1970s and 1980 at Xerox' Palo Alto Research Center. Its design principles include the goal of "personal mastery", defined as "a system being entirely comprehensible to a single individual." Several descendants, such as Squeak and Pharo, are still actively developed and used, but only Cuis has maintained the focus on simplicity.

UVM and Uxn are small and portable virtual machines that support graphical user interfaces. The motivation behind UVM is simplicity and resilience, to which uxn adds the desire to use low-power computing hardware in order to reduce the material and energy footprint of computing technology.

Julia

https://julialang.org/

Legally open vs. effectively open

Open Science is very much in fashion today, but it remains a rather fuzzy concept. In addition to divergent opinions about what exactly should be made open, there are multiple degrees of openness. Most of the discussion around Open Science is about what I call "legally open": allowing everyone to access the outputs of research legally and at reasonable cost (which in the digital era means at no cost). Open access is all about being legally open, for example. The Open Source movement in software started similarly, concentrating on licenses. What I will argue for in the following is that Open Science should be about making research outputs *effectively* open, by which I mean that research outputs are made *easily* accessible for as large an audience as reasonably possible.

To make the difference between the two concepts clearer, here is a useful analogy. Legally open means I am telling people: "All the results of my project are in the papers on my desk. Go have a look, the door is not locked." Effectively open is "Here is a summary of my project, with links to all the details you may need and references to textbooks you may have to read if you come from a different domain."

Effectively open corresponds to the tradition in scientific publishing. We don't submit photocopies of our lab notebooks to journals for reviewing and publication. We write articles *explaining* our work to readers, starting by describing the context and motivation for the work. And if an article is not sufficiently well written to be understandable, good journals will reject it, no matter how good or novel the underlying research work could possibly be.

There is a good reason for this tradition. Science is about increasing our knowledge of the world collectively. A contribution that nobody but the

author can understand is not really a contribution. Moreover, science is built around an error correction protocol. To err is human, as is having biases that influence our reasoning. Such errors and biases are eliminated in the long run because individual contributions are critically examined by other scientists. They make errors and have biases as well, but, most probably, different ones. A particular strategy of critical examination, peer review of submissions to journals, has become the gold standard for quality control in science, even though there this is more of a historical accident than a conscious methodological choice.

If we want code and data to become research outputs of equal value to papers, we need to subject them to critical examination as well. It isn't obvious *how* best to do this, and I expect that experimentation with different techniques will be required. But it seems obvious to me that we have to introduce a requirement equivalent to the "well written" criterion for articles. It has to become the authors' responsibility to convince their colleagues of the quality of their code, their data, and the computational environments that they have chosen to base their work on. That means that code, data, and computational environments have to become effectively open. They must be amenable to critical examination, with reasonable effort, by independent experts, i.e. experts that were not involved in their creation.

The main obstacle to making code and computational environments effectively open is the complexity of today's software stacks. Even an apparently perfectly documented data analysis presented as a computational notebook is only superficially well documented. The code in the notebook typically makes use of many libraries, which in turn rely on an even larger number of lower-level libraries, with the total dependency graph easily containing hundreds of software packages. No reviewer can reasonably be expected to critically examine the whole software stack. On the other hand, most of the code in those hundreds of software packages is not relevant for the top-level data analysis. It may thus be possible to make the data analysis effectively open by organizing and packaging the code differently, or by providing inspection tools that guide its examination. This should become a research question in scientific software engineering.

A completely different approach is to refactor scientific software with the goal of moving as much as possible into a small number of very widely usable packages. The generality of such packages would then justify significant efforts invested into developing and independently examining them. A typical researcher could then trust these packages on the basis of expert

stamps of approval, rather than on personal examination. This approach is analogous to how we develop trust in industrial products, such as drugs or refrigerators, via technical norms, regulatory oversight, quality labels, and other sources of trust by delegation. The "industrial" core software would then be complemented by domain-specific and project-specific artefacts, which might be higher-level software layers or scientific models written in digital scientific notations.

Leibniz

A research project aiming at developing a digital scientific notation for computational physics and chemistry. Such a notation should be suitable as well for other domains using predominantly mathematical models, but my focus is on the domains that I know best.

Leibniz is named after Gottfried Wilhelm Leibniz, who made important contributions to science, mathematics, formal logic, and computation, topics that are all relevant to this project. He invented a widely used notation for calculus, laid the foundation of equational logic by his definition of equality, and anticipated formal logic with his "calculus ratiocinator".

An embeddable specification language

A first iteration of Leibniz focused on developing a formal language for embedding specifications and algorithms into a narrative written principally for human readers. It is the subject of a publication and of a (recorded) presentation at RacketCon 2020. The latter is the best introduction to Leibniz at this time. You can then move on to studying a pedagogical example and other, more technical examples.

The focus on a formal language embeddable into a narrative motivated the choice of the Racket ecosystem and in particular its documentation language Scribble. Leibniz is implemented as an extension to Scribble. It is an algebraic specification language, based on equational logic and term rewriting. Its design is strongly inspired by Maude and its predecessors from the OBJ family. The first iteration of Leibniz is in fact equivalent to a subset of Maude (providing only Maude's *functional modules*), but with a very different syntax in view of its intended use. A nice feature of algebraic specifications is that they consist of small elements whose order

rarely matters. This makes it easier to insert these elements into the flow of a narrative, much like mathematical notation.

The goal I have set myself for a usable version of Leibniz is the possibility to write a readable specification for a molecular mechanics force field such as the AMBER family. The first iteration is clearly not good enough for that. Most of all, it lacks built-in support for collections, such as "all atoms in a molecule". You can define collections such as lists explicitly, of course, as it is done in Maude. Another mathematical concept that is not easy to represent in Maude or Leibniz 1 is the function. Maude is an intentionally minimalistic language, which I think a digital scientific notation should not be. This sets the agenda for the next iteration: improving expressiveness.

An interactive authoring system

At this stage of the project, the edit-compile-run/view cycle of Racket and Scribble became more and more cumbersome. Modifying and debugging both the implementation of a new language and test code written in this language at the same time led to feedback loop of unacceptable duration, due to the two nested edit-compile-run cycles of Racket and Leibniz itself. I had just discovered, through fortuitous circumstances, the Pharo live programming system, which is a descendant of Smalltalk. Implementing the second iteration of Leibniz in Pharo looked like a good opportunity to evaluate live programming in general, and Pharo in particular, for a project that could benefit a lot from this improved interactivity.

Shortly after starting the second iteration, Glamorous Toolkit, a new user interface and development environment for Pharo focusing on moldable development was made available for adventurous explorers (it has since advanced to beta status). I rapidly adopted it for my work on Leibniz (and other projects), because moldable development turned out to be a very good fit for my work. Another major step was the introduction of Lepiter, a computational notebook on steroids integrated into Glamorous Toolkit. It turned the implementation of an interactive authoring system for Leibniz from a over-ambitious idea into something that looked doable. My current prototype (shown in this short demo and in this longer demo) has a lot of rough edges, but it is good enough for me to experiment with language features.

Another promising discovery that became an experimental feature of Leibniz is e-graphs and their use in equality saturation, inspired by the Metatheory

package for Julia. For Leibniz, a modification will be required to make it useful since Leibniz has both symmetric equality axioms and asymmetric rewrite rules, whereas e-graphs handle only symmetric equivalence relations.

Expressiveness has significantly improved in the second iteration. Sorts can now be composite terms, which allows for sorts such as "an array of 5 non-negative integers". Array terms are a language features, though only one-dimensional arrays are fully implemented for now.

Links to the future

Link rot is a well-known problem on the Web. We have all been frustrated when clicking on links that just don't work any more. Most often, the server that the link points to has disappeared, or its contents have been reorganized.

This site contains links that show the opposite behavior: they may not work now, but they will work in the future. These "links to the future" are the links labelled "archive copy" at the end of each page.

How does this work? The two main ingredients to the answer are content-addressable storage and the Software Heritage archive.

All the pages on this site are backed by a Git repository hosted on Codeberg. The "permanent links" at the bottom of each page point to this repository. It keeps a complete history for each page, which ensures that the permanent links indeed point to the same version of the page that you are looking at, forever.

That's for some reasonable definition of "forever", of course, given that nothing is eternal in our universe. The more precise promise made for these links, by Codeberg, is that they will either point to the correct version of the page, or fail to resolve. If I delete my repository, for example, the links will fail to resolve from then on. Also if Codeberg closes down, or removes my repository for whatever reason, for example because it decides that its contents are in violation of its rules. Note that the promise is made by Codeberg the association. If Codeberg gets hacked, or bought by some evil entity, it is conceivable that my permanent links will work but resolve to something else in some unlikely but not impossible future.

The Software Heritage archive adds another layer of promises of longevity. Again these are promises made by Software Heritage the organization, which may well disappear one day, or get hacked. But assuming its continued

existence and well-being, Software Heritage promises that links into its archive, based on their permanent identifiers, will forever resolve to the exact same file contents. This is possible because the identifiers are derived from the contents of the file, by computing a cryptographic hash. Unless two files have the same hash (this is known as a hash collision, and it's not impossible but highly improbable by accident and very costly to provoke intentionally), the link cannot point to anything else than the original file.

Software Heritage archives the public repositories on some forges, such as GitHub, automatically, scanning the site from time to time to add new versions and new repositories. Codeberg is not (yet?) on the list of automatically scanned forges, so I save this repository myself from time to time, using this script. This is not instantaneous, so whenever I publish a new version, the archival links won't work for a while, because Software Heritage hasn't incorporated the new version yet.

This leaves one final question: how can I know the link to the Software Heritage archive before it actually works? Well, that's the nice part of content-addressable storage: since the identifier is computed from the file, I can compute it myself, from my own copy of the file, knowing that Software Heritage will obtain the same hash when it does its computation later. That computation is most easily delegated to Git, the magic incantation being `git hash-object <file>`. And that means that links to the future are actually very easy to implement.

Linux

This page is empty

Literate programming

An approach to program design and documentation that embeds software source code into an explanatory narrative, structuring the code to follow the narrative for convenience of a human reader.

Further reading: - Literate programming by D. Knuth. The paper proposing the idea and describing the first supporting tools.

Machine learning

This page is empty

Making Systems Explainable

A recorded talk by Oscar Niestrasz at the VISSOFT 2022 conference, demoing Glamorous Toolkit used for explaining software systems.

Associated paperL Making Systems Explainable, by O. Nierstrasz and T. Gîrba

The talk is about software systems from the perspective of software developers. But computer-aided research has very much the same problems, for the same reasons, and could benefit from the same solutions. The talk starts with a demonstration of how a simple board game can be implemented in a way that makes it inspectable from a multitude of perspectives. It doesn't take much imagination to replace that board game by a computational scientific model.

Consider the debate around CovidSim, the epidemiological model for COVID-19 used in the United Kingdom to inform public policies in 2020. It was heavily criticized, first for the code not being publicly available for inspection, and then for bad software engineering practices casting doubts on its reliability. Both of these criticisms are very valid. But if CovidSim had been developed openly and following best practices of software engineering for C++ code, would this have made it trustworthy in the eyes of (1) fellow epidemiologists, (2) public health officials and (3) the general public? I suspect the answer is no. While the model would be straightforward to run, understanding how the various modeling choices made impact the result remains very difficult because it requires modifying the code, And that requires an in-depth understanding of how the code works. But even a perfect C++ simulation code takes an immense effort to understand for anyone but its authors.

Now imagine the models of CovidSim implemented in Glamorous Toolkit, with lots of views into the model, examples for how to run it with various parameters, and explanatory narratives linking right to these examples and

views. Like the Ludo game in this talk. Quite a different scenario.

Open Science requires more than papers, data, and code being publicly available. They must be made accessible for inspection and modification, at a reasonable level of effort and technical competence. We require scientific papers to be well-written. Researchers cannot just publish copies of their hand-written notes. Why should we accept them publishing software that nobody else can understand?

Malleable Systems

See the Malleable Systems Collective.

Minimalism

The idea of reaching a goal with as little resources as possible. For an overview of minimalism in computing, see Wikipedia.

Model

This term has many different but similar meanings in various domains of science and engineering. In the digital era, it is not uncommon to work at the intersection of multiple disciplines that use the term somewhat differently.

The most relevant uses of "model" in this context are:

- Scientific models
- Models in formal logic
- Model-driven engineering in software development

Modeling scientific discourse

On November 12/13, I participated in the Workshop on knowledge synthesis infrastructure at CSCW 2022, and more specifically in a working group on discourse modeling. This is my synthesis of the two sessions, which is obviously colored by my prior work, my limited knowledge of the field, and my expectations. I hope that other participants will summarize their points of view as well.

The overall idea of discourse modeling is to make scientific discourse (i.e. the publicly visible part of communication between scientists, which today consists mainly of publications and conference contributions) semi-formal and thus more accessible to machine processing (searching, summarizing, etc.). The central concept is the discourse graph, which identifies common elements of discourse (questions, hypotheses, evidence, etc.) and their relations. A discourse graph does not replace the actual discourse (which is more complete and more nuanced), but complements it with a machine-readable summary. Discourse graphs can also be seen as a first step towards integration of new knowledge into knowledge graphs, which model the state of accumulated scientific knowledge in a field, rather than the individual contributions.

While some people have been creating discourse graphs as part of their reading of the literature for a while, these graphs remain personal for now. They are rarely published, and not processed collaboratively. The questions addressed by the working group at the workshop focused on making discourse graphs more accessible, both from the authors' and from the consumers' points of view. Ideally, we wanted to be able to create discourse graphs collaboratively, using well-known tools (we chose Semantic Media Wiki for our experiments, because we had an instance available for the workshop), and make them available for others in some form of database that could store multiple discourse graphs. As was to be expected, we did not reach this goal in two sessions, but we learned a lot in the process.

The object of our experiment was a transcript of our own oral discussion on the topic, which happened in a Zoom breakout room. That's a bit different from scientific discourse, of course, but the overall principles should still apply.

One of the first problems we encountered is the ubiquitous tension in formalization efforts between top-down and bottom-up construction of the formal system to be used. In the case of discourse modeling, the formal system consists of the definitions of the nodes and edges in the discourse graphs. The nodes are the elements of discourse that can be identified (questions, evidence, sources, etc.) and the identifiers used for them. The edges are the relations between elements, such as "evidence X supports claim A". In a top-down approach, the formal system is defined at the start and then guides the formalization process. In a bottom-up approach, the concepts in the formal system are identified during the formalization process, requiring regular revisions of the annotations. A top-down approach is usually more efficient, and produces a predictable result. On the other hand, it fails to capture aspects of the discourse that were not considered in the design of the formal system. Considering that discourse graphs are a rather recent idea, and that none of us had much experience with them, a bottom-up approach seemed more appropriate. We used the node types "question" and "claim" quite frequently, plus "source" which, for our oral discussion, referred simply to a speaker. There wasn't much "evidence", because we weren't doing science. I ended up putting "proposal" on a few items as well. In real scientific discourse, additional domain-specific node types (e.g. "algorithm" in computer science) would be required. As for edges, we didn't make it to that stage, realizing that we would first have to put unique identifiers on our nodes, a task for which there was no obvious simple solution in Semantic Media Wiki.

An issue that we discussed a lot and kept in mind during our practice was the possibility of the collaborative use of discourse graphs. This includes collaboratively constructing such a graph, but also the use of existing graphs in a collaborative setting, be it close collaboration in a team, or loose collaboration in a scientific discipline. In a bottom-up approach to building a formal system, there is a regular need to establish consensus on some definitions, or establish and document the absence of consensus, in the case that there are good reasons for different people having different definitions. We discussed in particular the possibility of using federations, as implemented in various forms in various tools in the knowledge processing universe. Examples we looked at (much too briefly!) are Federated Wiki, Agora, and everything2.

Another issue we discussed was the importance of the user experience in working with discourse graphs, and the different roles that users can take. Much of the discussion in the Semantic Web space focuses on the technology, with much less emphasis on the needs of users. Personally, I found Semantic Media Wiki to be a rather bad user interface for the semantic annotation of discourse. The "ground truth" of a Wiki page is marked-up plain text, which in the presence of a lot of markup (formatting plus semantic annotations) becomes hard to read. The rendered version, however, de-emphasizes the semantic annotation too much. Finally, the transition from one to the other is too cumbersome and slow. Of course, I am spoiled by spending much of my workday in fast-feedback live systems: Emacs and Glamorous Toolkit.

The two main user roles around discourse graphs are "author" and "analyst", with "author" subdivided into "author of a narrative accompanied by a discourse graph" and "author of a discourse graph for a pre-existing narrative". Authors need authoring tools, in which they can build the discourse graph, possibly creating or extending as they go the formal system that frames it. Analysts need tools for searching and browsing databases of discourse graphs, plus good visualization tools. We probably don't have good tools for either role, but it seems to me that they can be built by combining existing and well-known ingredients. The main risk I see for the future of discourse graphs is a wealth of tools for analysts (which is the role most people in the Semantic Web universe care about), but few or no good authoring tools, and thus no data for analysts to analyze.

Finally, a few words on how this topic relates to my own recent work on Leibniz, my digital scientific notation. The common aspect is the goal of supporting the formalization of scientific discourse, though at different levels: discourse graphs are about formalizing the narratives, whereas digital scientific notations are about embedding complete formal systems, as parts of scientific models, inside a narrative. A formal system defined by a Leibniz context would be referred to by a node in a discourse graph.

Moldable development

Quoting the Moldable Development Web site:

> Moldable development is a way of programming through which you construct custom tools for each problem.

This implies erasing the boundary between development tools and code under development. Instead of writing a piece of software to perform some task, you extend a software environment by tools for working in some problem domain. The added tools do not just perform tasks, but also provide feedback and insight concerning the problem domain to the user of the software system.

In computational science, this is a big step towards shifting the focus from tools and tasks to problems, models, and methods, something I have been advocating since 2015.

Today's reference environment in supporting moldable development is the Glamorous Toolkit, which is based on Pharo but also supports other languages to varying degrees.

Molecular mechanics

A technique of molecular simulation, based on the idea of treating atoms as classical point masses.

See Wikipedia for more information.

Molecular simulation

This page is empty

Nix

See the Nix Web site.

Observation

This page is empty

Open Science

As the ongoing public debates about global challenges such as climate change or the Covid pandemic have shown, people are losing trust in science. In part this is due to societal changes that extend far beyond scientific practices: there is a general loss of trust in institutions, and in particular governments. Taking one more step back, there is a loss of trust in the hierarchical decision structures of Western societies, with its authorities that rely on the opinions of experts.

But there are also changes inside the scientific community that have contributed to this loss of trust, which is shared by many researchers themselves. The pressure towards bibliometric productivity and the information technology revolution have encouraged and enabled scientists to do more research but this comes sometimes at the price of a lower level of rigor. Moreover, increasing specialization makes it more difficult for scientists to judge the reliability of the work of their colleagues that they build on. The reproducibility crisis is a good illustration: a large number of studies that were widely expected to be reproducible turned out not to be.

The Open Science movement is a reaction to this development, aiming for more trustworthy processes at all levels with a focus on increasing transparency. This push towards transparency is in conflict with notions of intellectual property that have been put in place to ensure a competitive advantage, both to scientists and to the institutions funding their research. Reducing competition is thus a key to successfully implementing Open Science, but it is limited by the fact than in a context of scarce resources (mainly jobs and project funding), some level of competition is inevitable.

So far, Open Science consists of three levels of changes. The first level is Open Access: anyone should be able to consult the original publications of scientific findings, rather than having to rely on summaries provided by

expert committees or journalists. Of course, few people can actually read and understand those publications. But the circle of people who can extract information reliably from these articles is still much larger than the circle of people who could afford to consult the original literature before Open Access, in particular in the less prosperous countries of our planet.

The second level is the publication of the datasets and software that underly most of today's scientific research. In the era of ubiquitous computer-aided research, a published summary of a study and its outcome is simply no longer sufficient. Many details of the applied methods are documented only in the code (this story about a seven-year battle between two teams that didn't share their code is a nice illustration), and access to the data and code permits other scientists to study the same phenomena from different perspectives. Publication of data and code thus makes science more verifiable, and by enabling complementary work, it supports the construction of the web of interrelated findings that permits consensus formation and ultimately trust.

The third level of Open Science is about laying open the decision procedures in scientific research. Which topics get explored, and by whom? Who decides which results get published? How does quality control actually work? Which biases affect any of these decisions? How do political and economic interests intervene? The ultimate goal is ensuring that scientific research, in particular when it is funded from public money, benefits society as a whole.

At this time, only the first level, Open Access, has made significant progress. Its necessity is accepted by all stakeholders, but the details of implementation remain a subject of debate. The main problem is the enormous power held by the traditional scientific publishers. In the digital era, the value they contribute to the publication process has become negligible, but they still control the names of the well-respected traditional journals. These journal names are a key element in defining scientific reputation, a tendency that has become much stronger with the introduction of bibliometry into the evaluation processes of scientific institutions. The publishers are doing their best to monetize this control, by extracting hefty publication fees from authors of Open Access publications. There are clear signs that scientific institutions are willing to move away from bibliometry-based evaluation (see e.g. the San Francisco Declaration on Research Assessment), though for now it is not clear by what it is going to be replaced.

The implementation of the second level is still in its early days. Non-publication of data and code is still the norm, and even when these crucial

elements are made public, they are not included in the critical examination that a paper undergoes during peer review. For data, the FAIR principles have established criteria that are increasingly accepted, though not yet massively applied. For code, only a handful of journals perform elementary tests upon submission of a paper, such as checking for computational reproducibility.

There is still a long way to go towards trustworthy software. So far, no effort at all is made to check if the computations conform to the scientific methods as described in the paper. There is not even any requirement for authors to provide readable code - we are still accepting legally open code rather than insisting on effectively open code. Lack of incentives is one reason but probably not even the major one: the state of the art in scientific software makes it nearly impossible to write code that a human reader other than the authors can understand in sufficient detail. And if a reviewer cannot be expected to understand the code, there is no hope for the peer review process to inspect the code for correctness. My work on digital scientific notations aims at addressing this issue, but many other changes need to happen, in particular to avoid falling into the trap of cheap complexity all the time.

Another open problem is the reviewing process itself. Today, individual reviewers are expected to comment on all aspects of a submitted study. This is unrealistic when dealing with publications that have a dozen authors from several disciplines and deploy millions of lines of code to analyze gigabytes of input data. Such submissions must be reviewed by teams combining different specializations, one of which needs to be in scientific software engineering. Moreover, such complex reviews should happen in parallel to the research itself, rather than as a single huge task at the end.

The third level is just beginning to be explored. The most concrete experimentation is open peer review, of which many varieties have been implemented by journals (e.g. F1000Research, which was one of the pioneers) and independent reviewing networks (of which PubPeer is perhaps the best known). The main difficulty faced by open peer review is being the only decision process being opened so far. This puts reviewers at the risk of retaliation by the colleagues they criticize in their reviews, e.g. in hiring or funding decisions. In contrast, open peer review works very well in less competitive contexts, such as journals that define the role of peer review as helping authors to improve their work, rather than decide on acceptance or rejection. Examples are the Journal of Open Source Software and ReScience. This attitude can be seen as a step towards separating dissemination and quality control in scientific research from the decision processes about resource allocation (jobs,

funding, …), which is very much in the spirit of Open Science.

Another aspect of level three is the keywords "diversity" and "inclusion" appearing on the lists of criteria for composing committees, in an attempt to make these committees more representative of society at large. For now, diversity and inclusion efforts only address the most egregious and outwardly visible misrepresentations, such as gender disparities. More subtle though numerically very important discriminations, such as the *de facto* exclusion from science of everyone who doesn't speak English, are not even discussed yet as possible objectives.

The most important decision processes in science are those concerning resource allocation: hiring and funding. They ultimately determine which research topics are being explored, and by whom. The only initiatives that I am aware of for profoundly reforming these processes happen in the so far marginal "decentralized science" communities. In my opinion, we need more ideas to improve resource allocation, and we must actually test them in practice, rather than limit ourselves to debating their expected benefits and downsides.

Open Source

This page is empty

Pace layers in science and technology

The concept of pace layers has been introduced by Stewart Brand in 1999. If you haven't read his article Pace Layering: How Complex Systems Learn and Keep Learning, do so right now (it's not very long) and then come back here.

The pace layers of human civilization that Stewart Brand identifies are, from fast to slow: - Fashion/art - Commerce - Infrastructure - Governance - Culture - Nature

Brand puts science into the infrastructure layer. Quote:

```
Education is intellectual infrastructure. So is science.
They have very high yield, but delayed payback.
Hasty societies that can't span those delays will
lose out over time to societies that can. On the
other hand, cultures too hidebound to allow education
to advance at infrastructural pace also lose out.
```

From the bird's-eye perspective of human civilization, infrastructure is clearly the appropriate layer for science. But we can also zoom in on science and technology, the two being closely intertwined, and identify pace layers in the advancement of this particular aspect of human civilization.

The fastest layer is *observation* (science) and *tinkering* (technology). Being curious about one's environment, without any long-term plans in mind. Playing with ideas, watching what happens. Moving fast and breaking things.

Next comes *empirical research*, which consists of formulating hypotheses

based on observations, and then testing them on new observations obtained from experiments, which are designed specifically for testing a hypothesis. On the technological side, we have the *design and fabrication of artifacts* for a specific purpose.

The design and refinement of *models* occupies the next layer in science. Models are more general than hypotheses, requiring a wider range of observational input and a wider range of experiments for testing. In technology, the corresponding layer is about *techniques* for designing and fabricating artifacts.

The fourth pace layer of science contains *theories* and *paradigms*. Paradigms define the phenomena that a discipline considers relevant, and the methods used to study them. Theories are the consolidated foundation of research based on a paradigm. Examples are quantum mechanics in physics, and evolution in biology. The technology analogue is, unfortunately, called *technology* as well. Examples from the discipline of electrical engineering are electric generators and integrated circuits.

The slowest pace layer contains *disciplines* in both science and engineering. In the bigger picture of human civilization, they are part of the governance layer. Disciplines are about the values of a community. What is good science? What is relevant science? Different disciplines have different answers. These values matter more than the original domain of study of a discipline. Consider physics, which has evolved to include social phenomena into its domain of interest, as for example in econophysics. On the other hand, many molecular phenomena are chemistry, and thus outside of physics, because they are described in terms of patterns rather than numbers. What really defines physics is the focus on mathematical models and quantified observations.

In the digital era, the technology dominating much of scientific progress is computing. Computing technology has the same pace layers as other technologies, but they all move faster than for the older branches of engineering. This makes it challenging to use computing technology in the slower pace layers of science. For example, techniques for implementing scientific models should be stable at the time scale of evolution of scientific models, which is several decades. While there is computing technology that is sufficiently stable (Fortran, Unix, HTML, …), it is not particularly well suited for representing scientific models.

Pharo

An Open Source dialect of the Smalltalk programming system. See the Pharo Web site for more information.

Plant

See Digital Garden

Premature formalization

Formalization comes with an aura of mathematical precision, which is not limited to the sciences. As a consequence, there has always been a tendency to use formal systems prematurely, i.e. without having verified that the formal system satisfies all the requirements of its role (scientific model, prediction machine, control system, etc.). Before computers, this was not much of a problem because developing and applying formal systems was very laborious. Today, writing a few lines of code is sufficient to create a formal system and use it in practice, without the critical examination that should have happened before.

One example, the use of bibliometry to measure the impact of scientific publication, is described in the article on quantification. Premature quantification is a frequent special case of premature formalization, as is premature automation via computers. Another example I discuss elsewhere is the use of static type systems in a not yet fully understood application domain. Human-computer interaction is a rich source of examples of premature formalization: rigid entry forms, unforgiving (and often incomprehensible) error handling, overly constraining computational media for text and graphics, and many more. See the talk "Programmable ink", which calls this the "tyranny of formalisms".

Premature formalization becomes dangerous when formal systems are used in a normative ("this is how the world ought to be") rather than descriptive ("this is how we think the world works") way. Most people living in the Western world have at some time encountered administrative forms (paper or Web) that they had to fill in but that were lacking some option or entry field that mattered for their case. Those are forms based on prematurely constructed formal systems, i.e. formal system that are insufficient to deal with the complexity of real life and yet are imposed on its users.

Today's Web3 movement is particularly prone to premature formalization. The type of formal system that it focuses on is the blockchain, in particular its fancier varieties that permit smart contracts and Digital Autonomous Organizations (DAOs). As its name suggests, a smart contract is intended to be a formalization of the legal concept of a contract, adapted for use by computers. However, legal contracts work very differently from a computer program. They are used in contexts where the possibility of non-respect, for various reasons, is always present and dealt with by various social institutions (courts etc.). Smart contracts, in contrast, are computer programs that are run under clear predefined conditions and change the state of the blockchain in a fully automated fashion. This can of course be a useful way to proceed, but it is in no way a faithful formalization of a legal contract. Some people in the Web3 movement are aware of this and think about the wise use of smart contracts as a new kind of tool. Others, however, see smart contracts as strictly superior replacements of traditional law, because of its perceived objectivity and reliability. The aura of mathematical precision strikes again.

Recommended reading: - Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems (by FM Shipman and CC Marshall)

Programmable ink

A talk/demo at StrangeLoop 2022 by Szymon Kaliski of Ink&Switch that explores two aspects of human-computer interaction: visual vs. symbolic, and formal/rigid vs. informal/fuzzy.

Visual interaction is one of the main missing pieces in today's scientific computing systems. We have tools for turning symbolic data into visual data (e.g. simple plotting or more elaborate data visualization tools), and also a few tools that permit interaction with the visual form, but nothing that goes in the other direction: from visual to symbolic. The talk shows two examples that illustrate this point very well: the derivative plot generated on the fly from a hand-sketched curve, and the computation of a correlation between two hand-sketched curves. Such tools are essential in constructing a computational medium for science. And also completely absent from my own explorations (see Leibniz), mostly because of my lack of competence in this space, but also because days are still limited to 24 hours.

Visual representations are also great for working at a less formal level, as all the demos illustrate very nicely. For symbolic representations, the equivalent is the extraction of formal relations from plain language, a task that is addressed by some work on language models. But so far, there is little work on supporting the transition from informal to formal. Most work on computing technology either stays exclusively in formal universes (logic, programming languages, …) or emphasizes informal processing (machine learning, …). Even little formalization aids would be of great benefit to scientists. I'd love to have a tool where I can sketch a plot and have the axes straightened (and then more cleanup…), as shown in one of the demos!

Finally, I love the expression "tyranny of formalisms" used a few times in this talk, referring to the fact to most of today's computational tools are examples of premature formalization, in particular in their user interfaces. I

suspect this is because computers, being dynamical formal systems, attract people who love formalisms, and aren't aware of the fact that most people work differently.

Programming language

This page is empty

Programming system

This page is empty

Publishing

Why do scientists publish their work? Leaving aside the incentives of today's publish-or-perish culture, they do so to enable their peers (and sometimes others) to learn about and engage with their work. It's worth to take a closer look at this, because it matters for the Open Science movement's efforts of publishing more than papers, in particular data and code.

Readers of a scientific paper can adopt one of the four following levels of engagement:

1. **Take note** of the work. Read the abstract, take a look at the figures and their captions, read the introduction and conclusions. If it looks potentially relevant for their own work, file it away, mentally, electronically, or physically.
2. **Understand** in detail the scientific questions asked, the hypotheses made, the methods and tools applied, the data obtained and/or processed, and the conclusions drawn.
3. **Judge** the well-foundedness of the hypotheses, the adequacy of the methods and tools, the quality of execution, and the validity of the conclusions.
4. **Make their own** the ideas, the methods and tools, and the data, in view of basing their own future work on them.

These four levels of engagement can only happen in the order cited. You cannot judge something that you do not understand, nor make your own something you are unable to judge. Increasing engagement requires increasing levels of competence and experience, and of course a more and more serious investment of time.

When it comes to data or code, a frequently stated goal for Open Science

is "reuse", the R in the FAIR principles. It corresponds to the highest level of engagement. You have make the data or code your own if you want to reuse it, because you are responsible for ensuring that they are adequate for the purposes of your own research project. Reusing a dataset or a piece of scientific software is very different from reusing yesterday's dishes after washing them. Scientific data and code are not commodities. They are highly context-dependent. Responsible reuse requires understanding both the original and the new context.

It then follows that reuse is possible only if the data and code is published with sufficient documentation to permit understanding and judging them. This is particularly difficult for code. A popular quote from the preface of the famous textbook Structure and Interpretation of Computer Programs says that "Programs must be written for people to read, and only incidentally for machines to execute." Unfortunately, this is not today's reality. At best, software developers make an effort to keep their code readable for themselves and their peers, but not for their users. This is the main motivation behind my research project on Digital Scientific Notations.

Python

This page is empty

Quantification

Quantification is a specific kind of formalization, and probably the kind most prominent in science. Quantification in scientific models goes hand in hand with measurement in observations. Together, quantification and measurement are often presented as the hallmark of science, as the turning point at which the exploration of a phenomenon becomes scientific (see e.g. the Wikipedia page).

Being a special kind of formalization, quantification can also happen prematurely, and in fact it often does. An example that academics are well familiar with is bibliometry. The informal concept of impact, applied to a study or to the publication(s) resulting from it, is easy to grasp and apply in evident situations. I doubt anyone would question my claim that Albert Einstein's 1905 paper introducing special relativity had more impact on our collective scientific knowledge than my 1998 paper on elastic network models for proteins (which is probably the highest-impact publication I have so far). Formalizing this concept to the point of making impact a measurable magnitude is, however, a highly non-trivial matter. Such a magnitude allows to compare any paper to any other paper, impact-wise, which is not an obvious operation. Was Einstein's paper on relativity more or less impactful than his contemporary paper on Brownian motion? That's not a question I'd be willing to answer. I have read and understood both papers and am quite familiar with the theories that were later developed on the basis of these two works. Both papers had a very high impact, but in very different respects and in different sub-fields of physics. How could I compare them?

The mismatch here is that, in mathematical terms, the concept of impact has only partial order (you can rank some works relative to each other, but not all), whereas numbers have total order (for any pair of non-equal numbers, one is larger than the other). Numbers also have other properties that are not obviously valid for scientific impact. For example, the average of a set

of numbers is well-defined, but the same cannot be said about the average impact of Albert Einstein's publications.

Bibliometry took the approach of "any number is better than no number", putting the label "impact" on an easily measurable quantity for which some relation to impact can be justified: the number of citations to a paper in the later scientific literature. This principle of "better any number than no number" is perhaps the most frequent cause of premature quantification. It allows moving on with building superficially precise models and theories that however fail to describe the phenomenon that they were supposed to describe.

Reliable knowledge

"Reliable knowledge" is the first part of the title of a book by John Ziman, a physicist whose interest later shifted to the philosophy of science. It's an excellent book, whose full title is "Reliable Knowledge: An Exploration of the Grounds for Belief in Science". I am less enthusiastic about the term "belief in science", which I prefer to replace by "trust in science", but that's minor quibble.

I see "reliable knowledge" as the best term to summarize the goal of scientific research, which is why it got a page of its own in my digital garden. It also describes the two directions in which scientific knowledge advances: more knowledge, and increased reliability for existing knowledge. The reproducibility crisis has shown the importance of the second direction, which we have collectively neglected during several decades of emphasizing novelty as the main criterion for judging scientific publications.

Reproducibility crisis

Starting around 2010, more and more cases were reported of scientific findings published in peer-reviewed articles that other scientists were unable to reproduce. Sometimes they reached different results or conclusions, sometimes they had to give up because of missing information. This sudden increase in results known to be irreproducible is often called the reproducibility or replication crisis.

The sudden explosion of the number of these cases is probably just a domino effect: the more people discuss the issue, the more others are inclined to check for reproducibility, and thus discover failure. But the reproducibility failures are real and cast a shadow of doubt on the reliability of today's scientific research.

Much has been written about this crisis, and in particular many hypotheses for its causes have been proposed. The Wikipedia article provides a good entry point. In the following, I will limit myself to the computational aspects that I haven't seen discussed elsewhere so far.

First of all, there are different forms of (ir)reproducibility that's worth distinguishing. The three main categories are:

1. Experimental reproducibility: repeating an experiment as described in the literature, and checking if the observations are similar enough, according to the state of the art.
2. Statistical reproducibility: re-doing a statistical inference based on fresh input data, usually obtained from a different sample, and checking for similarity of the inferred results.
3. Computational reproducibility: re-running a computer program, using the same code and input data, and checking for identical results.

I haven't seen a single case of experimental irreproducibility cited in the

context of the crisis. In fact, I can remember only a single widely discussed case of experimental irreproducibility in my whole scientific career: the 1989 cold fusion study by Fleischmann and Pons (see Wikipedia for details). And yet, in theoretical discussions about the importance of reproducibility in science, people talk almost exclusively about experimental reproducibility, probably because it is the historically earliest aspect of reproducibility.

Both statistical and computational reproducibility, which together cover all the cases I have seen cited in the context of the crisis, are phenomena of the digital age. This is rather obvious for computational reproducibility. Statistics has been around for much longer, and even today's most commonly used statistical techniques are about 100 years old. But before computers, doing statistics was extremely laborious. It was done sparingly, for important questions only, and usually by people with solid training in the techniques. Nowadays, it takes little training to load a dataset into a statistical software package and click a few menu items to perform an analysis.

It is in particular not required to understand the domain of applicability of the methods, nor the correct interpretation of the results. It should be obvious that this is a recipe for frequent mistakes. In theory such mistakes should be caught in peer review, but this requires authors to publish all their data and reviewers to take the time to carefully re-do and check the computations. That is starting to happen, but remains exceptional.

A more subtle problem is that, even if you understand the statistical techniques behind a study very well, you cannot be sure that the software used by the authors implements them correctly. Most such software is designed to be a black-box tool. Even if the source code is available (Open source software), and can thus be studied in principle, it is usually not written with readability and verifiability in mind, but for efficient execution by the computer. This is true of course of nearly all of today's scientific software, which is why I am interested in re-editable software and why I work on digital scientific notations.

In philosophy of science jargon, these issues illustrate the epistemic opacity of computations. In more down-to-earth terms, when scientists use computers to apply scientific models and methods, they don't really know what they are doing. If you don't know what you are doing, you cannot document it either. And insufficiently documented work is a major cause of irreproducibility.

While I have focused on software so far, the data that are being analyzed can contribute to statistical irreproducibility as well. When the people

analyzing the data were not closely involved with the data production (by whatever means), there is a good chance that they are unaware of some critical details that they should be aware of in order to analyze the data correctly. The current rush to data publication, in the context of the Open Science movement, happily ignores this issue. Therefore I expect more rather than fewer cases of reproducibility issues related to data in the near future, until the scientific community realizes that data are safely reusable only if they are carefully documented.

Computational irreproducibility is just another case of epistemic opacity. It is caused by the complexity of today's software stacks. Scientists not only ignore what exactly their software does, they do not even know in detail which software they are running, and therefore they cannot reproduce the computation on a different machine, or later in time.

Ending the reproducibility crisis will require, among many other changes in research practices, an increasing awareness of the pitfalls of delegating work to a machine, and of relying on software and data produced by others whose tacit knowledge may be crucial for their proper (re)use.

Reusable vs. re-editable components

Nearly all nontrivial information systems are assemblies of components, often produced independently by different people. Components meant to be used in different contexts are either designed to be *reusable* or *re-editable*.

Software libraries and datasets are the most common examples of reusable components. They are designed to be integrated into an assembly without any modification or adaptation.

Project templates (e.g. for use with Cookiecutter) and configuration templates are examples of re-editable components. The integrator must study them and then adapt them to the particularities of the system being assembled.

The term "re-editable" was coined by Donald Knuth in an interview in 2008. He expresses a clear preference for re-editable over reusable software:

> I also must confess to a strong bias against the fashion for reusable code. To me, "re-editable code" is much, much better than an untouchable black box or toolkit. I could go on and on about this. If you're totally convinced that reusable code is wonderful, I probably won't be able to sway you anyway, but you'll never convince me that reusable code isn't mostly a menace.

The mainstream view in software engineering, and also in scientific computing, is the opposite. The accepted ideal is a software library with thorough documentation and an equally thorough test suite, maintained by a stable team of competent professionals. Developers needing the functionality of such a library use it as-is and design their own client code around it. In the maintenance phase, they update libraries as quickly as possible. In case of breaking changes to the interfaces, they adapt their own code.

Both approaches have their good and bad sides. The arguments in favor of reusable components are mainstream and easy to find. But which are the advantages of re-editable components? I can't speak for Donald Knuth, who doesn't go into details in the interview, but I can offer my own thoughts.

A particularity of software in computer-aided research is its double role as a tool and as an expression of scientific models and methods. Reusable software is designed to be used as a black box, without a deep understanding of its implementation, even when this implementation is accessible (Open Source). It is also designed to be useful in a wide range of applications. Re-editable software, on the other hand, is designed to be read and understood by its users, and also more focused on the application its designer had in mind. This makes re-editable software more valuable as a readable expression of scientific models and methods. Moreover, it encourages or even forces its users to read the code and understand what it does, reducing the risk of inappropriate use of the science it embodies. Such inappropriate use is in my opinion an important but little discussed cause of the reproducibility crisis in science.

Comparing software to material artifacts, reusable software is analogous to industrial products, whereas re-editable software corresponds to bespoke artifacts made by a craftsperson. The mere fact that craftspeople still exist after two centuries of industrialization, even though their products are usually much more expensive, indicates that there is a value in non-standard artifacts based on simpler designs. They are obviously better adapted to their specific context, but they are also more repairable, and adaptable in case of evolving needs. Re-editable software shares those advantages.

Further reading: - Reusable vs. re-editable code (preprint)

Science gateway

A Web site providing access to a collection of software tools and databases for a specific scientific domain. The goal is to reduce the barrier of access to computational tools and methods via deployment as a service.

Further reading: - Science Gateways: Accelerating Research and Education — Part I - Science Gateways: Accelerating Research and Education — Part II

Science in the digital era

The broad topic of this collection of essays is the changes that scientific research is undergoing as a consequence of, or in parallel to, the information technology revolution that started in the 1960s.

One important aspect, and my main focus, is the change in how formal systems are used in scientific models. Before computers, obtaining inferences from formal systems was laborious, and limited the size and complexity that formal systems could have. With automated computation, large and complex formal systems (typically called software) are easy to create and apply. However, it is impossible to evaluate all, or even all relevant, inferences one can draw from such systems, meaning that today's computational models are far less understood than their ancestors, and only superficially tested by confrontation with observations. Moreover, the current state of software technology makes it easier to build large and complex formal systems than small and simple ones. This is the ultimate cause of computational irreproducibility, a major ingredient of the reproducibility crisis.

Another aspect of the information technology revolution is the new forms of organization and communication it enables for scientific research. In particular, they permit a level of transparency that was immediately recognized as desirable, leading to the Open Science movement that is rapidly gaining momentum.

Of course, social changes are at least as important as communication technology in the emergence of Open Science. Many topics of research, e.g. health or climate, are of increasing social and political relevance. The preceding paradigm of science, which saw research as an industrial activity producing knowledge, is no longer appropriate. In the name of productivity optimization, it restricted participation in the process of doing science to a small number of experts, who alone decided which topics were worthy of study,

and who alone could judge which practical consequences should be drawn from their findings. With trust in experts waning in parallel with trust in the governments that employ them, this leads to phenomena such as widespread climate change denial. Public policies can be science-based only of a much larger part of the population can participate in the collective learning process that we call science.

The two aspects I have outlined above create a new tension. Science cannot become more transparent and more accessible if it uses complex software as the main (or only) expression of its models. It is not enough for those models to be Open Source, they also have to be understandable and explorable. One of my personal research topics is how to encourage a return to simple and understandable formal systems, by using specifications written in a digital scientific notation rather than software in the construction of scientific models.

Sciences of the artificial

A term introduced by Herbert Simon for disciplines such as mathematics and computer science, which study neither nature nor human societies, but abstract structures created by humans.

Further reading: - The sciences of the artificial by Herbert Simon

Scientific computing

This page is empty

Scientific model

The construction, evaluation, and incremental improvement of models for observable phenomena is one of the main objectives of scientific research. From a birds' eye view, the constantly evolving output of science is a network of models plus metadata about these models: where they come from, which observations they explain, which observations they don't explain, etc.

Scientific models can be described or classified according to several criteria. An important one is the distinction between empirical or descriptive models on one hand and explanatory models on the other hand. An empirical model summarizes observations and permits predictions, via interpolation or extrapolation, along a few well-defined parametric dimensions. For example, a mathematical function fitted to a time series permits predictions at different time points. An explanatory model describes observations as the outcome of a more fundamental process or mechanism. It is much more powerful than an empirical model, because it can be transferred (extrapolated) to a much wider set of systems, beyond varying well-defined parameters.

In the digital era, both empirical and explanatory models have acquired specific computational variants that would have been impractical before commodity computing. The new empirical models are those obtained by machine learning techniques, and the new explanatory models are the models underlying simulations. Some simulations are based on older models of the pre-digital era which have been scaled up to larger or more complex systems. This is the case for molecular simulation, or for weather forecasting. Other simulations are based on new kinds of models that would lose interest if simplified to the point of being manageable without a computer. Examples are cellular automata or agent-based models.

Another criterion is the distinction between informal and formal models. An informal model, which could for example be formulated in plain English,

refers to concepts whose precise meaning depends on the context, and which are therefore malleable. A formal model, in contrast, refers to very precise and narrowly-defined concepts, often from mathematics and formal logic. These aren't distinct categories, however, and not even the extremes of a scale, as the commonly used term "semi-formal" might suggest. Most non-trivial scientific models are partly formalized, with the formalized aspects embedded into a wider informal description.

Computation, as defined e.g. by Turing machines, is the pinnacle of the development of formal reasoning so far. Its roots are an intellectual current that started in 18th century Europe with the work of Leibniz and others and became mainstream in the late 19th and early 20th century. At that time, the idea that all of mathematics and then science should be formalized was very popular (see e.g. Hilbert's problems), but became more nuanced after Gödel, Turing, and other showed that formal reasoning has inherent limitations.

Nevertheless, automated formal reasoning in the form of computation became an important technique in scientific research, and highly formalized models are still considered the most advanced ones, particularly in physics. However, formal models in science very frequently contain informal elements as well, even though they are often seen as weaknesses. The most frequent informal element is an undetermined parameter that must be fitted to observations, thus adapting the formal model to the specific context of a specific system.

In recent years, there has been a strong counter-current advocating informal models as superior to formal ones, though I have never seen this point of view stated in these terms. The counter-current I am referring to is data science, and the superiority claim is best exemplified by a 2008 article in "Wired" entitled "The End of Theory: The Data Deluge Makes the Scientific Method Obsolete". And yet, the short history of data science also illustrates the opposite move towards more formalization, for example with neural networks that are more structured, e.g. multi-layer networks or convolutional networks.

Scientific notations for the digital era

Introduces the concept of a digital scientific notation and outlines desirable characteristics for such notations.

Available (open access) at: - CoScience - arXiv

Seedling

See Digital Garden

Semantic Web

This page is empty

Semantics

This page is empty

Simulation

This page is empty

Situated software

Software designed in and for a particular social situation or context.

See also: re-editable software. Not exactly the same concept, but a similar tension between simplicity and generality.

Recommended reading:

- Situated software, by Clay Shirky

Smalltalk

See Wikipedia

Software collapse

The phenomenon of software becoming unusable due to backwards-incompatible changes in its dependencies.

Recommended reading: - Dealing with software collapse (preprint) - Surviving Software Dependencies by Russ Cox. Quote: “Developers trust more code with less justification for doing so.”

Software engineering

This page is empty

Software stack

This page is empty

Source code

This page is empty

Specification

This page is empty

Staged computation

"Staged computation" is a technical term that I suspect most readers of these pages have never seen before. And yet, it refers to a very common technique, one that all of us are using every day. More importantly, understanding this technique matters for understanding computational reproducibility.

A staged computation is defined as a computation that proceeds as a sequence of multiple stages, each stage producing the *code* (not the input data!) of the following stage. The last stage produced the final output. In the academic literature, staged computation is mostly discussed in the context of code generators or compilers. However, it's most frequent use case is running a compiled program. Compilation is indeed a computation, and it produces the code (the executable binary) for the next step, which is the execution of the compiled program.

Why does this matter for reproducibility? Consider the case of a simple Fortran program (substitute your favorite language if you wish). You start from the Fortran source code file, which you first compile, because of the dual nature of software. That's the first stage. Then you run the compiled binary, which is the second stage, and you obtain a result. What you care about is the reproducibility of the complete two-stage process: you want to make sure that the same source code file will lead to the same results.

In an ideal world, the source code file would fully define the result, and the intermediate binary executable would be a mere implementation detail. In the real world, that is unfortunately not true. First of all, the Fortran language does not fully specify the semantics of the Fortran language. Different compilers can interpret a program differently, and yet all conform to the language standard. This lack of semantic precision is intentional, because if offers compiler writers more opportunities for code optimization. Other languages, such as C or C++, made the same choice in their standards. How-

ever, even if your language has fully specified semantics, different compilers can lead to different results as a consequence of mistakes. Compilers are complex pieces of software, so it's unreasonable to expect them to be free of bugs.

Therefore, if you want to make sure that someone else can reproduce your results, you have to make the complete two-stage sequence reproducible. You thus have to document the compiler you have used, and also all compilation options. Your colleague (or your later self) trying to reproduce the result will then obtain the exact same binary executable, and by running it the exact same output.

Unfortunately, this isn't the end of the story. The compiler is a binary executable that has itself been produced by a prior compilation step. You really have a three-stage computation. Or… more. The compiler used to compile your Fortran compiler has also been compiled. Also, your program has been silently complemented with precompiled program libraries (at the very least the Fortran runtime library). It isn't even obvious how many stages your computation really has. The chain of compilers compiling compilers is of course not infinite, but hard to trace.

This is known as the *bootstrap problem* and an active topic of research in the Reproducible Builds community. It is easy to state: Given a computer that can run binary executables, how you can add a toolchain for building binary executables from source code without already having one? If you want a glimpse of the complexity of this problem, have a look at the GNU Mes project, whose goal is to provide a solution applicable to several Linux distributions. Its basic idea is to start with simple compilers for small subsets of real programming languages, and progressively build more complete ones. At the very start, it is inevitable to have some hand-written binary code, but this should be kept as small as possible to make the whole system *auditable*, i.e. understandable in all detail by a person external to the development team.

By the way, the Reproducible Builds community is not primarily about reproducible research, but about reproducible software as a key component for cybersecurity. If you want to make sure that the software you run is free of malware, it is not sufficient to use Open Source software and inspect its source code. You must also be sure that the binary executables you are running were actually derived from the public source code, using a compiler that has not been tampered with. This is why understanding staged computation matters.

Further reading: - Reflections on trusting trust. Ken Thompson's 1984 Turing Award Lecture on trusting compiled software. - Staged computation: the technique you didn't know you were using (preprint).

Static type systems

If you follow discussions about programming languages even just a bit, you have surely witnessed a heated debate about static type systems. I haven't made (nor seen) a systematic study of the question, but I'd bet that it's either the most popular topic, or number two after questions of syntax. And I couldn't stop myself from writing a few paragraphs about it here as well.

Static type systems are formal systems for reasoning about the consistent use of data types in software source code. The other main option, dynamic type systems, verifies the consistent use of data types during program execution. The obvious advantage of static type checking is that it is not necessary to run the program, which might take a long time before hitting a type error. The main disadvantage of static type checking is that it constrains what is allowed in a program. A type checker will only let pass what it can *prove* to be correct, meaning that it rejects code that may well be OK but is not *provably* OK.

What I find surprising in the frequent heated debates is that the nature of the type system is rarely even discussed. People talk about static vs. dynamic types as if there were only one static and one dynamic type system. Academic computer science research does look into the details of type systems, of course, but consumers (i.e. software developers) don't seem to be very interested in these details. Also, academic research seems to have restricted the search space to type systems in the vicinity of the ML type system, for whatever reasons (this is really not my area of expertise).

Is it reasonable to assume that there is a single best (or good enough) type system for every kind of software? The experts seem to believe it is, but I don't agree. I consider type systems to be domain-specific, and I suspect that the ML type system and its variants are simply a good choice for writing compilers and related tools, which is what researchers in this field tend to

do.

A few examples from my own experience with scientific software illustrate that the ML type system is not very useful there. My first example is dimensional analysis. It's a formal system that has been used in physics and engineering for much longer than we have had computers. It has turned out to be very effective in catching mistakes. And yet, it cannot be implemented in the popular static type systems. The F# language implements dimensional analysis, but as a special case added to its generic ML-like type system.

My second example is linear algebra. If you implement matrix algorithms, your only data types in a standard programming languages are float array of float, and integer for array indices. What you really want to catch common mistakes is something different: you want to check the compatibility of array dimensions, and the conformity of array indices with array dimensions. Again that's not something you can do in an ML-like type system.

As a side note, dependent types can handle both cases, but they are not mainstream, for good reasons.

The conclusions I draw from the these and other cases I have encountered are: (1) type systems should be considered domain-specific, (2) they should not be baked into a programming language, except if it is domain-specific as well, and (3) it would probably be useful to use multiple type systems in parallel in the same code. All that would make a type system an add-on module, rather than a central language feature. This raises the interesting question of interfacing code that uses different type systems. Which is of course already an interesting question on today's world, because large software systems are rarely written in a single language, but most language designers have so far ignored it, treating all code written in a different language as external, with type checking disabled.

The closest technology I am aware of in this space is F# type providers. They turn types, but not the whole type system, into library modules that can interface to the outside world. Caveat: I haven't used them, so I can't say how well they work in practice.

Once you consider a type system something malleable rather than rigid and imposed, the task of constructing a type system for a specific domain is very similar to the formalization of scientific models. A developer would start writing dynamically typed code, and once there is a first working prototype, think about which concepts would make good types and which properties are most amenable to static verification. This may sound similar to gradual

typing, but the latter seems to focus on the gradual transition to a single predefined type system, rather than on an emergent one.

For scientific software, this could in fact be a good approach to formalizing computational models. It is similar to what scientists have done in the past. Consider the very mature field of classical mechanics. It started with Newton's laws of motion, but grew into a complex Web of interrelated formal systems. Some of them (e.g. Lagrangian and Hamiltonian mechanics) are alternatives to Newton's formulation that serve the same purpose but are more convenient in specific situations. But others work at the meta-level, very much like a type system, e.g. the law of conservation of energy. Maybe software tools such as (malleable) type checkers can help to discover similar fundamental properties in the scientific models in the digital era.

Statistical reproducibility

This page is empty

Tacit knowledge

This page is empty

Technical debt

Recommended reading:

- [Technical debt in computational science)(https://hal.archives-ouvertes.fr/hal-02072258), by yours truly

Technological sovereignty in science

The term "technological sovereignty" is usually applied to nation states or similar political entities (see e.g. the Wikipedia page on this topic). It refers to the capacity of such entities to control the use of technology and shape its development in accordance with their values and goals. But it makes sense to apply this concept to a much wider range of people and institutions, including the actors in scientific research, from individual scientists to research institutions. That is what I will try to do in the following, concentrating on information technology in scientific research.

Sovereignty as dependency management

One way to approach this issue is by looking at dependency relations. Which resources and services does it take for someone to use a certain technology productively and sustainably?

At the highest level of sovereignty, it's only commodity resources and services, such as off-the-shelf hardware, electricity, and network access. There are functioning markets for them, which makes the dependency relation uncritical. One level down, there are dependencies on non-fungible suppliers in a contractual relation, e.g. for software. Depending on a single supplier is risky, but the risk is mitigated if the supplier has contractual obligations, and thus also takes a risk in defaulting on a contract. The lowest level of sovereignty results from resources and services that are unreliable and/or out of your control. That could be the weather if you run solar-powered computers, but the most typical situation is a dependency on a rare competence (the retired professor who is the only person who understands the software he wrote and that you depend on) or on an entity that is so much bigger than you that

it doesn't even have to listen to you (e.g. Google, or the Python developer community).

These three levels are of course only a rough outline. The reliability track record of suppliers matters a lot, as does the effort or cost of replacing their product or service by a different one.

By these criteria, most individual researchers today have a rather low level of sovereignty, which translates directly into a feeling of computational disempowerment. Few individuals have sufficient competence and time to manage their computing systems and to write their own software or modify existing software to their needs. Whereas systems management is at level 2 for academic researchers (they usually have someone in their lab who is paid to help them), software for common needs such as data analysis is usually fully outside of the control of an individual user. It is developed and maintained by large entities, corporate or Open Source communities, which may offer some level of support but in general no guarantees, in particular not for maintaining compatibility in the future. Moreover, much software is too complicated to master for someone whose focus is on difficult scientific problems, given the time and resource constraints of today's research environments.

For many researchers, the best choice in terms of sovereignty is the use of proprietary scientific software, such as Mathematica, that comes with contractual guarantees and reliable customer support. That's an issue that Open Source advocates tend to downplay. They rightly point out the importance of openness, but fail to appreciate the loss of sovereignty that comes with using software developed by a community that one is not sufficiently engaged with.

Teams of small to moderate size are only slightly better off than individuals. With more people collaborating on a project, there's a higher chance that someone has just the right competence for a technical task. But sovereignty really increases only when the size of a team or lab permits hiring specialized staff, hiring external consultants (which requires not only money, but also sufficient competence to choose the right consultant), or having sufficient weight in an Open Source development community. That level of sovereignty is limited to a small number of Big Science projects.

Large institutions, such as universities or national research labs, are large enough to develop their technological sovereignty, but their values and goals are situated at the metascience level. What they could and should support,

but usually don't, is the development of better digital infrastructure for science, which would increase the technological sovereignty of researchers.

The entities that have the highest technological sovereignty in science today are the communities that develop scientific software. However, even their sovereignty is often constrained by software dependencies, in particular since contractual relations with software component suppliers are almost unheard of in science.

In summary, the technological sovereignty of most researchers and research projects is rather low, which has a couple of undesirable consequences. Scientists often cannot do exactly the work they would like to do, for lack of control over their software. Worse, they often don't know exactly what they are doing, for lack of understanding of their software. And they run into reproducibility issues because they lack control over the evolution of their software. Perhaps the worst consequence, not yet much discussed, is the absence of critical examination of software, which remains exempt from peer review, and for which no alternative evaluation process is in sight.

Becoming more sovereign

What can researchers and research institutions do to increase their technological sovereignty? Quite a bit, but often it is a matter of finding the right compromise with respect to other criteria.

The most obvious technique is avoiding dependencies. In particular, write your own software, maybe together with a few colleagues, but not as part of such a large project that you couldn't maintain it on your own any more. This approach is of course limited by available time and competence. It's unrealistic for most research to write *all* the software they need themselves, but it may be worthwhile considering this option for the most critical research software.

Another obvious, but difficult to apply, strategy is to go for simpler software, both in one's own products and in the dependencies one relies on. Simpler software means most of all less code, and more understandable code. This usually implies less general software, maybe even situated software.

There is a long history to achieving sovereignty by simplicity. In 1981, Dan Ingalls wrote in Design Principles Behind Smalltalk:

```
If a system is to serve the creative spirit,
```

```
it must be entirely comprehensible
to a single individual.
```

Later Smalltalk systems have abandoned this principle, but there is at least one, Cuis Smalltalk, that still has simplicity among its priorities. Among programming languages and systems, Forth stands out by its emphasis on simplicity of implementation, and it is indeed very feasible for a motivated amateur to develop and maintain a practically useful Forth system from scratch. I did this in the 1980s, as a high school student. A recent project in this space is Minimacy, a functional programming language designed for simplicity of implementation. None of the systems I have cited has been designed for scientific research, and I am not suggesting that scientists should adopt them. But they can serve as inspirations for achieving simplicity by design.

Yet another strategy is to choose dependencies for which there is a market, meaning multiple potential suppliers. For software, this is practically possible only for standardized software. Standardized programming languages such as C, Fortran, or Java have multiple implementations, as do standardized libraries, e.g. OpenGL for graphics, or MPI for parallel computing. Standardization is a slow process, and therefore standardized software can seem old-fashioned or even obsolete from the point of view of today's fast-moving tech world. But if it's good enough for your research computing needs, consider what matters more to you: sovereignty or the latest technology.

Research institutions, in particular the bigger ones, can do something very different, and very important, for improving science's technological sovereignty: support the development and maintenance of digital infrastructure. This includes in particular the development and maintenance of Open Source software that is safe as a dependency because its future is backed by stakeholders in science. It differs from today's community-based development of scientific software not only by long-term financial backing, but also by institutional governance that takes into account the needs of users who do not participate in development. In other words, the needs of those users who today turn to proprietary software for the stability guarantees and the technical support that communities cannot provide.

The cognitive surface of software

🌱

I have been trying for a while to come up with a good term for the concept I will describe here, and for now I have settled on "cognitive surface". It may not be the best one. If you have an idea for a better metaphor, please get in touch!

In analogy with physical objects, I call the parts of a computation that are readily comprehensible by its users its *cognitive surface*, and the parts that are comprehensible only with specialist knowledge, specialist tools, or with significant effort, its *cognitive bulk*.

Note that comprehensible is not the same as visible. For Open Source software, the source code is visible, but most of it doesn't make sense to a user with reasonable effort, and therefore belongs to the bulk. What is and what isn't comprehensible depends on the person approaching the computation. It is not an objective property of the computation or software, but a relational property involving a person and an artifact. But then, this is true for the accessible surface of physical objects as well. A tunnel with a small opening is accessible to a child, but not to an adult.

In the context of standard command-line or GUI applications, the cognitive surface of a computation consists of the software's user interface, its documentation, and of inputs and outputs of an execution. Techniques such as literate programming, computational notebooks, or moldable development aim at increasing the cognitive surface by attaching explanations to (parts of) the code. Some notebook implementations, and Glamorous Toolkit as the primary implementation of Moldable Development, also add tools for

interactive exploration of algorithms and data, be they inputs, outputs, or intermediate results. The conference talk Making Systems Explainable gives a good illustration.

Increasing the cognitive surface is one way of reducing the epistemic opacity of a computation, increasing trust in its correctness.

Another technique for reducing epistemic opacity is decreasing the cognitive bulk. One approach is reducing the amount of code, leading to minimalism. A complementary approach is reducing the functionality of software, by removing what is not required for a specific use case. This leads to re-editable code and situated software.

The dual nature of software

One of the keys to understanding the causes of computational irreproducibility is understanding the dual nature of software. Software has a human-facing side, which we call source code, and a machine-facing side, which we call binary code. The latter is not a very adequate name, because everything in a computer is stored as binary data, including source code. But that's the jargon that has established itself. Binary code is what the computer's processor can directly execute. It's a sequence of instructions defined as bit patterns. It's not something you want to look at when dealing with software.

In the not-so-distant past, humans wrote binary code for software of moderate size. I have personally written binary code for the Z80 processor in my Colour Genie home computer in the 1980s. That was the only option for programming that computer other than using the built-in BASIC interpreter. It's fun to talk to a processor in its native language, but also very cumbersome. I quickly moved on to the Z80's assembly language, which is a textual language, meaning source code, in which each line maps to one processor instruction. I still wrote code at the level of processor registers and individual memory locations, but I wrote it as legible text.

When you write source code, something or someone has to translate that source code to binary code before the software can be run. In order to write Z80 assembly code, I wrote a translation program, called an assembler. And since I didn't have an assembler when I wrote my assembler, I had to translate it to binary code myself. That can be done, but it's not fun. I have never done it again.

The main message is that running software written as source code requires some other piece of software that translates source code to binary code. Such software goes by different names: assemblers, compilers, interpreters, etc., depending on how it works exactly. If you don't write software yourself, you

may well be unaware of this translation layer, as it is almost invisible. But it's there, and it matters.

One reason why translation software matters is that, like any other software, it usually has bugs. That's true even for old and widely used translation software, such as the GNU compiler collection. Check the paper "Finding and understanding bugs in C compilers" for some examples. Your source code can be perfectly correct, and yet produce wrong results because of a bug in your translation software. That's certainly much rarer than having a bug in your own source code, but it happens.

The second reason why translation software matters is that it's what ultimately defines the meaning (in technical terms, the semantics) of the source code. A practically important case is floating-point arithmetic. Your processor very probably uses an instruction set that implements the IEEE 754 standard for floating-point arithmetic. But none of the popular high-level languages for scientific computing (C, C++, Fortran, …) lets you program in terms of IEEE 754 operations, which include details such as rounding modes that most people don't want to have to think about. It's the compiler that decides how exactly your nice mathematically-looking formulas are translated into IEEE 754 operations. And different compilers make different decisions (because language standards don't bother to deal with such details either). Most compilers let you influence their decisions via compile-time options, which however are not part of your source code. This is why floating-point operations are so often not reproducible. They are in fact perfectly reproducible if you use the same compiler with the same options, but few people even record this information along with their numerical results.

The third reason why translation software matters is that it can intentionally do nasty things, such as adding viruses or spy code to your software. That's not much of an issue in scientific computing, but a big source of worry for software that is widely deployed and/or relevant for someone's security. Ken Thompson's Turing Award speec, Reflections on trusting trust, is the classical reference on this.

Unfortunately, most software today is distributed as binary code. Even Open Source software is no exception. Open Source means that you can download the source code somewhere, but running the translation step is often so complicated that most people are unable to do it. Instead, they download binary code prepared by somebody else, usually via package managers such as Debian's Advanced Package Tool or the multi-platform Conda. Container images, such as those you get from Docker Hub, contain binary code as well.

Using someone else's binary code means that you have to trust that someone to actually have compiled the source code that they claim to have compiled (because you cannot check that). And since you don't know which exact translation software was used, you cannot check if it has bugs, and you cannot reproduce the binary code if it ever disappears from the download server. That's by far the most frequent cause for computational irreproducibility today.

The good news is that software distribution can be done better. Guix and Nix are two package managers that track the complete translation process, recursively (i.e. they even track how a compiler was compiled itself - see Staged computation), and allow you to re-run it. You get all the source code, and the recipes that were applied for translating this source code to binary code. You can also get ready-made binary code (which is a lot more efficient), which however you can verify if you want to. Nice, isn't it?

The sustainability doughnut of scientific software

The sustainability of research software has attracted much attention over the last decade. This is the consequence of software having become an essential tool for almost any form of research today. Like other research tools, e.g. scientific instruments, software requires constant attention and effort to ensure that it continues to play its role in evolving scientific and technological environments. I won't go into the details of the various challenges involved, as there is a growing literature you can easily find. A good starting point is this 2016 report of The Knowledge Exchange.

Outside of scientific research, sustainability is also an increasingly common keyword. There are ongoing discussions on the sustainability of the global economy, but also more focused discussions on particular aspects of the economy, such as agriculture or specific branches of industry. However, there is an important difference: whereas the sustainability discussion concerning research software is about ensuring *sufficient* resources, the sustainability discussion concerning the economy is about an *excessive use* of resources that is damaging the environment in which economic activity (and life!) takes place.

Economist Kate Raworth has come up with a stunningly simple visual image of sustainability, the doughnut economy, along with the equally stunningly simple insight that "A healthy economy should be designed to thrive, not grow."

What I will try to do in the following is apply the doughnut idea to scientific software. I won't go much into the "be strong enough to thrive" aspect because that is already being discussed extensively (see the first paragraph above). It's the "not grow" part that I want to focus on, because it has not

received much attention so far.

First of all: what exactly is growth, referring to a software project? Three dimensions that immediately come to mind is more users, more contributors, and more code. But there are others: More code complexity. More functionality. More dependencies. More supported platforms. There are obvious and less obvious correlations between these dimensions.

Next, what are the environmental resources that a software project uses and could potentially overuse? Use of material and energetic resources is relatively low in software development, although increasing use of AI tools (ChatGPT, GitHub's Copilot, …) could change that. In software deployment, it can be very high, but that depends more on the problem the software is applied to than on the software itself. The main resource consumed by software projects is the time that humans invest to produce or use it. Software development requires competences that are relatively rare, and scientific software is worse because it requires specialized scientific domain knowledge as well. There's a finite pool of potential contributors for each project, so there is competition among different software projects in a given domain for this finite resource, and also competition between software and research projects in the same domain. The potential user community is limited as well.

One overuse scenario is a software project growing to the point that other software in the same space can no longer thrive. This would lead first to a technical monoculture and then to an epistemic monoculture. Each piece of software implements specific scientific assumptions and models that its users cannot avoid adopting. If all researchers use the same of software for some task, there is nobody left who can identify flaws in this software, or in the assumptions it is based on. Science needs epistemic diversity for the same reason that natural ecosystems need biodiversity: to increase resilience.

Another overuse scenario is software becoming too complex to use or understand, the scarce resource being the cognitive capacity of its users. Computational irreproducibility is one symptom of such overuse: it indicates that a software stack has become too complex for its users to describe and reconstruct. The worst scenario in a research setting is software that is simple to use but too complex to understand, because it encourages scientists to use software that they do *not* understand. Unfortunately, that scenario is not only common, but even actively encouraged by institutions that see the benefit of easier-to-use software but not the risk of scientists losing agency concerning their research methods. The recent popularity of science gateways is a good example.

The problems caused by code complexity are well known, and nobody actually *wants* to increase it. But practice shows that increasing code complexity is common in software projects, usually as the result of growth along other dimensions, in particular growth in functionality and growth in the number of contributors. Code complexity growing with the number of contributors is an illustration for Conway's law: the complexity of the social organization is reflected in the complexity of the artifact.

The first victims of increasing code complexity are individual users. When code is simple, they can inspect it to understand its behavior, and they can modify it to adapt it to their needs. With more complex code, they have to rely on documentation, experimentation, help from other users (or developers), and other incomplete means of understanding. Modifying the code thus requires more effort and becomes increasingly risky. Users are losing agency.

Teams can better adapt to software complexification because they have more human resources at their disposal, but the cost of using the software, in terms of time and effort spent on it, nevertheless increases. As complexity increases further, the minimum size an organizations needs to have to retain agency over the code increases as well (this is an illustration of the Law of Requisite Variety in Cybernetics). Organizations smaller than the agency threshold are reduced in status from artisans who master their tools to machine operators that can only choose from a list of actions defined by someone else.

I have experienced this dynamics first-hand in the scientific Python ecosystem. When I discovered Python in 1995, at version 1.3, the interpreter was a compact and well-written C program that was easy to understand. Today, just understanding the release notes in detail requires the level of effort that was sufficient for understanding the code itself in 1995. The key scientific libraries followed the same path. The first release of Numerical Python in 1996 was written by one person and was also understandable by a single motivated individual. Today, NumPy's build system alone is more complex than the whole codebase in 1996. As a consequence, the original goal of the Matrix-SIG, which was empowering individual scientists to write their own code to solve their scientific problems, has largely been lost. Today's Python software stack empowers Facebook to build PyTorch, but for small research teams outside of computer science, choosing Python as a platform for specialized data processing scripts is a sure recipe for suffering software collapse before the end of the research project.

There are good and bad reasons for increasing code complexity. Software en-

gineering distinguishes between *essential* complexity, which is the complexity of the problem domain, and *incidental* complexity, which comes from architectural decisions and choice of tooling and dependencies. A good example for increasing essential complexity is the transition from ASCII via ISO 8859 to Unicode for representing plain text. It has lead to an increase in code complexity, but also enlarged the range of languages that can be processed from English-only to all written languages in human history. The holy grail of software engineering is accepting essential complexity while resisting the parallel introduction of incidental complexity. So far, this hasn't been very successful. Incidental complexity always sneaks into an evolving software project, often as technical debt when short-term benefits are prioritized over long-term costs.

What this means in practice is that extending functionality, for example to cover the needs of more scientific domains, or increasing diversity in the developer community, inevitably lead to increasing code complexity and thus to a shift in the target audience of the software. Software that once was a good choice for an experimental biologist doing simple data analysis becomes a good choice for large interdisciplinary teams doing climate simulations, but at the same time it becomes unmanageable for its original audience. Suitability for new kinds of problems may is usually the goal of the developer community in such a situation, and the increased complexity may well be a consciously accepted price to pay. However, developers are typically not aware of the impact that the added complexity can have on the existing user base.

Similar processes happen in other branches of technology. Perhaps this is the only way in which complex technology can evolve. But in other technology domains, the simple versions stay around as long as there is an audience for them. Industry has developed sophisticated tools and techniques for mass-producing complex furniture, but hand saws and screws are still available to craftspeople and amateurs. For software, it's the collapse of the foundations of the software stack that inexorably takes simple technology away from the people who relied on it. Installing Python 1.5 and Numerical Python 16.1 on a 2023 Linux system is beyond such people's means. It requires patching both packages, and a good understanding of the security issues in Python 1.5 to avoid disasters in deployment. It's only the growth-oriented branch of the Python stack that was maintained to keep up with the changes in the lower-level software it depends on. And Python is not an exception here. Only highly focused software projects manage to thrive without growing in complexity.

I see two causes for this seemingly inevitable dynamics towards growth in complexity, though I suspect there are more. One cause is technical: we don't know yet how to structure software into composable components at all relevant scales. The Unix philosophy, which says that a software tool should do only one thing but do it well, with complex tasks handled by the composition of several tools, just isn't applicable to software in general with today's state of the art. The other cause is social: the idea of forking a project into a stay-simple and a grow-complex branch, or into two branches focussing on different use cases, is not part of the strategical repertoire of FLOSS communities. They see the risk of a community split, but they are blind to the risk of a community slowly but continuously alienating some of its members.

The creeping complexification of software is not limited to scientific research. It happens in all application domains, and some consider it inevitable, almost as if it were a law of nature akin to the second law of thermodynamics. There are, however, projects dedicated to simplicity, which show that simplicity is achievable if it is a high-priority goal.

So far, I have described two paths in the evolution of scientific software projects that lead to an overuse of environmental resources. An obvious question is if there is something we can do about this. Can we make software projects sustainable in the long run? I am not aware of anyone even trying, so there are clearly no empirically testable answers. My aim with this essay is to launch a collective brainstorming process on these questions.

An important observation is that both of these paths are encouraged by the values of the industrial mindset that dominated the second half of the 20th century: growth, productivity, efficiency, competition. These values still drive organizations and individuals in science and engineering, in spite of the fact that we now recognize the harm they can do when applied without limits. The unsustainability of software projects is very much part of the general unsustainability of today's economy. Like the economy at large, software engineering has to shift towards an ecological mindset, adopting values such as sustainability, resilience, and coopetition. A necessary first step is to review current practices and analyze where they are in conflict with these new values.

A simple example for such a current best practice is DRY: Don't repeat yourself. Factoring out common operations creates new dependency relations, and thus more code complexity. There's a benefit, but also a cost. A shift to an ecological mindset implies attributing a higher cost to techniques that

increase dependencies.

Social structures need to be reviewed as well, and that is likely to be more difficult than reviewing software engineering techniques. Example: the currently dominating values in scientific FLOSS projects include welcoming anyone who wishes to join the community. This attitude feeds the dynamics of increasing diversity of needs that leads to increasing code complexity. The answer is of course not to be less welcoming. The answer is being explicit about the project's scope and target audience, and in particular about limits to the complexity that the project is willing to tolerate. This should reduce the number of people who are attracted to the project because they see it as a good starting point for doing something related but different. Such people should be encouraged to start a friendly fork, rather than blow up the project's coverage. In the long run, new social structures should emerge in which friendly forks cooperate without creating interdependencies, and thus complexity. This also involves solving difficult issues that seem simple at first sight. For example names: How to name friendly forks such that references are unambiguous and yet the common heritage remains visible? Who actually owns the name (= identity) of a FLOSS project?

I will end this discussion by pointing out an important connection between ensuring a safe floor and an environmentally responsible ceiling to the sustainability doughnut of scientific software: investment into the infrastructure of scientific computing. There is a clear need for stable and reliable software written by professionals as a commodity for non-technical users. This requires long-term funding and long-term governance, both of which are almost completely absent today. You can't expect a FLOSS community composed of volunteer early-career scientists to maintain infrastructure as a free service. Not even with a regular injection of three-year funding for one or two developers. Such communities thrive by innovating, not by providing services for others. Software infrastructure requires institutions employing experienced software developers, governed by a consortium of stakeholders that includes senior scientists who represent the users' interests.

Further reading:

- Nadia Eghbal's book "Working in Public" is a must-read for understanding how FLOSS projects really work.

The three environments of scientific software

The concept of an *environment* is widely used but vaguely defined. If you check its definition in a few dictionaries, you will find different though not incompatible definitions. The general idea is that an environment consists of things around you that you may not be aware of individually, but that matter for you as a whole.

In the context of software, the term "environment" most often refers to computational environments, which are the hardware and software stacks that support the execution of a specific program. There are, however, two other kinds of environments that matter in scientific computing. Their conflicting requirements for scientific software is the cause for many difficulties that computational science is facing today. I will call them the *technological* and the *scientific* environment.

The technological environment of a piece of software consists of hardware and other software that influences its design and implementation. This may seem similar to the computational environment, but the differences are profound and important. A computational environment consists of concrete machines and concrete bit sequences in memory, as used when a program is run. A technological environment consists of hardware and software as artifacts that evolve over time. Moreover, it includes hardware and software that has *some* relation to the software under consideration, even if that relation is not a dependency relation. A good example is past hardware that is no longer relevant in practice, but was relevant when the software was initially designed. Another example is software running elsewhere that the software under consideration exchanges data with. If the software under consideration is a Web server, its technological environment includes all client

software it is supposed to work with. A good way to visualize the relation between technological and computational environments is the following: a computational environment is a *snapshot* of a *subset* of the technological environment, with the subset defined by dependency relations.

The scientific environment of a piece of software consists of the discipline in which it was designed and the research projects in which it is used. It defines the *interpretation* of the software and its behavior by its users. Like the technological environment, it evolves over time as new methods are developed, older methods become obsolete, and the community's view of best practices changes.

As an illustration of the roles of the three environments, consider the following simple Python program:

```
from datalib import Dataset

points = [(1, 1), (-1, 1), (2, 4)]

data = Dataset()
for x, y in points:
    if x > 0:
        data.add_value(y)
print(data.average())
```

I have shown this code frequently in talks about reproducible research, and asked the audience what its output is. The most common answer is `2.5`, which is the average of the `y` coordinates of the points whose `x` coordinates are positive. This is indeed the *expected* output according to the *scientific* environment, which assigns meanings to names such as `Dataset`, `add_value`, and `average`.

Experienced computational scientists who think a bit about the question before answering will come up with a different answer: the output depends on the imported module `datalib`. In fact, all the named data and operations are defined in this library module, and you can't really know how what they do until you have seen its code as well.

Consider the following implementation of `datalib`:

```
class Dataset(object):

    def __init__(self):
```

```
        self.values = []

    def add_value(self, value):
        self.values = [value]

    def average(self):
        return sum(self.values, 0)/len(self.values)
```

The implementation of `add_value` is different from the expectation defined by the scientific environment. The latter says that adding a value to a dataset increases the number of data points by one. The code says that the method `add_value` replaces the points of the dataset by a single one.

In this simple case, we call the discrepancy between expected and actual behavior a bug. Real-life cases are more complex and subtle. What the code is expected to do may not be fully consensual in a community. A good example is the average of a list of floating-point values, for which there are many reasonable definitions although many scientists are not aware of this diversity. And what the code actually does is defined by the computational environment, which is usually incompletely defined and not reproducible. It is therefore difficult to assign blame for the discrepancy to anyone or anything more specific than the state of the art of computational science.

Another source of conflict is the different expectations about the time evolution of scientific software. The scientific environment expects software to evolve with its application domain. As concepts are refined and methods improved, code should follow along. The technological environment expects software to be "maintained", which means changed to remain functional as the technological environment itself evolves.

In practice, both tasks are handled in parallel by a single team of developers and maintainers. This doesn't always work out well, because teams tend to be dominated by either technology-aware software engineers or application-aware scientists. Whichever group dominates tends to see other group's focus as a distraction, or even a nuisance. There are two situations which are generally handled satisfactoryly. The first one is a very stable scientific environment, with software tracking its technological environment. This happens for implementations of well-known and widely used algorithms, for example Fourier transforms. The second one is the scientific environment evolving faster than the technological one. In that case, the team implementing "new science" handles code maintenance as a side task. The badly handled scenario is the scientific environment evolving at a slower rate than the technological

one. This often ends in software collapse.

Let's have another look at the Python example shown above. I have already explained why the scientific environment expects its result to be `2.5`. But what's the perspective of the technological environment? It sees the code as a Python program, with no interpretation of names. The code runs without raising an exception, so there's no reason to suppose a bug. And the output is… `4`. At least for Python versions before 3.0. With Python 3, the new semantics for float division change the output to `4.0`. In fact, what the technological environment's perspective really says is that the output depends on the computational environment, even in the hypothetical complete absence of bugs. Most of today's scientific environments still have a hard time understanding and accepting this fact, which is why computational reproducibility is still an unsolved problem in practice, even though it is fully understood in theory.

Trustworthy software

Scientists have to trust their tools in order to do their work. Software is an increasingly important tool in scientific research. And it seems that scientists do trust their software. There are no outcries of despair along the lines of "my research software crashes all the time" or "my simulation software had five bug fixes this year, I wonder if any of these bugs affected my last paper". It thus seems that scientific software is in much better shape than, say, our word processors or Web browsers. David Soergel disagrees, judging that "computational results are particularly prone to misplaced trust". Harold Thimbleby disagrees as well, comparing today's use of software in science to the use of statistics before clear standards for reliable practices were developed.

The problem with computational results is that, if they look plausible, you are looking at numbers that are neither obviously wrong nor obviously correct. It isn't obvious either what "correct" actually means. And there is no obvious strategy to reduce this uncertainty. It is fundamentally due to the complexity of the computations that are performed. There are of course various checks we can do. Run the software on simple problems for which we know the correct results, for example. But correctness doesn't generalize from simple to complex problems, in particular not for computation, which is chaotic: any change to the code or to the input data can change the result of a computation without any predictable bound.

Perhaps the best check we can do is solve the same problem with different software packages and compare the results. It is unlikely that different developers make the same mistake, so if we get the same output from multiple sources, it is much more trustworthy. Unfortunately, this approach is costly, and therefore rarely applied.

The way we deal with this fundamental uncertainty is unconditional optimism:

we trust results as long as they are not obviously wrong. Probably there are also scientists who go for unconditional pessimism, and stop using computers. I met a few researchers with this attitude in the 1990s, but I doubt that anyone with this point of view could survive in today's research environments.

This problem isn't as specific to software as it may seem from my description so far. Scientists have published mistaken results well before they had computers. One technique to detect and then eliminate them is independent critical examination: ask another expert on the topic, who was not involved in the original work, to go through the paper and check everything checkable. That was the original idea of peer review, introduced in the 1950s, when specialization became so pronounced that journal editors were no longer competent to judge themselves all the submissions they received. Peer review no longer works very well today, for various reasons, but that's a different story.

For software, we have never attempted independent critical examination, with rare exceptions that are limited to small pieces of code. This is not only due to a lack of resources and incentives. Software is much more difficult to review than papers. The complete software stack behind any published result is enormous, and just listing all its components is a non-trivial task. Moreover, software packages change all the time, and since computation is chaotic, review must be in principle be re-done after every change.

But all software is not equal. Some software is more reviewable than others. I have identified five dimensions along which scientific software packages can vary, making them more or less reviewable. These five dimensions are:

1. Wide-spectrum vs. situated software. Wide-spectrum software packages a lot of functionality, trying to satisfy a large number of applications. This makes it hard to review, but it also makes reviewing a worthy investment. Situated software is written for a narrow application scenario, making it simpler and thus easier to review.

2. Mature vs. experimental software. Mature software is stable, and thus easier to review than experimental software, which is a moving target.

3. Convivial vs. proprietary software. Convivial software is written with the goal of appropriation by users. They take the code and adapt it to their needs, rather than re-using somebody else's code as a black-box tool. Code that is easy to appropriate is also easier to review. Proprietary software, whose source code is not available, is very hard to review. In between, we have Open Source software, whose source

code can be inspected but was not specifically written for inspectability by its users.

4. Transparent vs. opaque software. Transparent software produces results that are easy to check for correctness because of the simplicity of the operations. It is therefore much easier to review. Unfortunately, most scientific software is opaque, making it hard to check its output.

5. Few dependencies vs. many dependencies. All software depends on other software for its execution. At the minimum, an operating system plus a compiler or interpreter. Dependencies must be examined as well as part of a software review, as they can be more or less trustworthy. Obviously, this becomes a Herculean task for software that has thousands of dependencies.

For a more detailed discussion, and for suggestions on improving the situation, see my preprint "Establishing trust in automated reasoning".

Turing machine

This page is empty

Web3

This page is empty

Webstrates

see https://webstrates.net/

Wiki

Recommended reading:

- Wiki as a pattern language, by Ward Cunningham and Michael W. Mehaffy