Science in the digital era

Konrad Hinsen

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: 2022-08-31

Welcome

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

https://github.com/khinsen/science-in-the-digital-era

This archival copy corresponds to commit 0515b59b1de9d21a54334c79c8bc640e35b826f4, which was published on 2022-08-31.

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

About the author

My name is Konrad Hinsen, and I have been a research scientist at CNRS in France since 1998.

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. 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.

Agent-based model

This page is empty

Cellular automaton

This page is empty

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 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 same 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 your 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 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 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, something 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. We also have good digital representations for observations, though the multitude of data formats makes them hard to manager. What's lacking is a suitable representation of models - that's what I hope to achieve with digital scientific notations.

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

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.

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.

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 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*)

Donald Knuth

This page is empty

Elastic network model

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 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.

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.

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

This page is empty

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.

Git

This page is empty

Glamorous Toolkit

See the Glamorous Toolkit Web site.

Guix

See the Guix Web site

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 GitHub, 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).

Julia

https://julialang.org/

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 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 GitHub. 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 GitHub, 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 GitHub 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 GitHub the company. If GitHub 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 all of the public repositories on GitHub, scanning the site from time to time to add new versions and new repositories. This means I don't have to do anything to get my site archived. But it's 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.

Machine learning

This page is empty

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

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

This page is empty

Open Source

This page is empty

Pharo

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

Premature formalization

This page is empty

Programming language

This page is empty

Programming system

This page is empty

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.

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 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 irreducibility, 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

Semantic Web

This page is empty

Simulation

This page is empty

Smalltalk

See Wikipedia

Software stack

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. 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

Wiki

Recommended reading:

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