Redressing the Balance: A Yin-Yang Perspective on Information Technology

Konrad Hinsen
konrad.hinsen@cnrs.fr
Centre de Biophysique Moléculaire (CNRS)
Orléans, France
Synchrotron SOLEIL, Division Expériences
Saint Aubin, France

Abstract

Information is an essential aspect of how we interact with the world around us. We acquire information and then integrate it to build knowledge, understanding, and trust, which in turn serve in preparing actions. Information technology (IT) is supposed to support all these phases of information processing. But does it?

An assessment of IT through the yin-yang lens from Chinese philosophy shows that over the last decades, support for the yin processes of building knowledge, understanding, and trust has been neglected, the focus of most research and development having been on the yang processes of acting. IT shares this imbalance with other aspects of Western and globalized culture. I discuss possible directions for reestablishing a yin-yang balance in IT, as a small contribution to redressing the balance in the world at large.

CCS Concepts: • **Software and its engineering**; • **Human-centered computing**; • **Computing methodologies**; • **Applied computing**;

Keywords: Yin, Yang, Science, Engineering, Technology, Automation, Formal systems, Specifications, Type systems, Simulation, Data analysis

ACM Reference Format:
Konrad Hinsen. 2024. Redressing the Balance: A Yin-Yang Perspective on Information Technology. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 11 pages.
https://doi.org/10.1145/3689492.3689808

Onward! '24, October 23–25, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1215-9/24/10
https://doi.org/10.1145/3689492.3689808

1 Introduction

The importance of computing technology for learning and higher-level knowledge[1] acquisition processes, such as understanding and trusting, has been recognized from its early days. Turing's famous paper [37] introducing what we now call the Turing machine is about the use of automated processing for deriving mathematical knowledge. The most common path to learning from computation is via its results, but the utility of software as a medium for learning and understanding has also been clear for a long time. Fifty years ago, Donald Knuth wrote [19]:

> It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e., express it as an algorithm.

Indeed, formulating knowledge in terms of algorithms, i.e. executable rules in a formal system, makes all of this knowledge explicit and is a proof that no aspects that matter to the result of the computation have been overlooked. Furthermore, it gives access to automated consistency checks ("does it compile?", "does it typecheck?") that increase trust in the correctness of the automated procedure.

More recently, Sussman and Wisdom elaborated on this idea, discussing "the role of programming in the formulation of ideas" [34]. Their example is the formulation of an advanced theoretical framework for classical mechanics in terms of unambiguous computer code, replacing its traditional expression in a semi-formal mathematical notation that turns out to be imprecise and ambiguous on a closer look. They even wrote an advanced textbook on classical mechanics based on this computational framework [35]. Another example of formalization supporting better understanding is the *post-hoc* formalization of the semantics of the organically grown Python language [27], which explains common pain points.

Computational science is the application at scale of the idea that knowledge can be derived from models and observations through formalization and computation. It is an

[1]In this essay, I use the term "knowledge" to mean declarative knowledge, although computation can also play a role in practical knowledge.

approach to scientific research that has been developed since the 1950s, in parallel with the development of electronic computers. Today, many scientific disciplines have a well-identified computational branch, and all of today's research can be qualified as "computer-aided" in the sense that even when computation is not the main tool deployed, it is always an essential support technology.

And yet, in the course of my career in computational science, which started in the 1990s, information technology has overall become *less* suited to the process of learning that is the essence of scientific research. APL[2], which I used for the exploratory part of my PhD thesis, was a more suitable environment for doing research than today's Jupyter notebooks. The subsequent Fortran 77 implementation of the methods I had developed using APL was more suitable for learning from the results of my simulations than many of today's simulation software packages, because it could be read and understood in its entirety in a few days.

It is easy to blame the growing complexity of scientific models for the loss of understandability, but that is not the whole story. As I will try to show in this essay, there has been very little work over the last few decades on technology that supports learning, understanding, and trusting, compared to the efforts invested in technology that supports *doing* things.

The analysis tool I have chosen for this assessment, the yin-yang lens from Chinese philosophy, is likely to be unfamiliar to most readers, which is why I provide a short introduction in the following section. The yin-yang lens provides a fresh perspective on the problem and reveals connections between information technology and other social processes, which I will come back to in the final part of this essay. After this introduction, I will examine how today's information processing technology supports the yin processes of learning, understanding, and trusting, and discuss possible directions for improving this support.

2 The Yin-Yang Lens

The yin-yang lens[3] originated in Chinese philosophy as the dualist aspect of an abstract world view describing multiplicity and diversity coming out of unity. It describes two complementary kinds of processes, labeled *yin* and *yang*, which feed and control each other in a circular or alternating pattern. Yin and yang processes are often grouped in complementary pairs that are at the opposites of some characteristic, such as dark^{yin} – light^{yang}, diminishing^{yin} – growing^{yang}, retreating^{yin} – advancing^{yang}, or humid^{yin} – dry^{yang}. To give a concrete example, in *taijiquan*[4], a Chinese martial art, yin movements are backward and down, and generally serve to absorb the energy of an opponent while maintaining one's own integrity and stability. Yang movements are forward and up, transferring one's energy to the opponent with the goal of hurting and destabilizing.

A focus on yin or yang is often expressed through a choice of words. In the paragraph above, I have annotated such words with ^{yin} or ^{yang}, and I will continue to do so in the following.

Information technology is used in processes that involve acquiring^{yin} information and integrating^{yin} it to form knowledge or trust, and also in processes that involve exploiting^{yang} information for acting^{yang} and shaping^{yang} one's environment (which includes other people). A simple example at the level of an individual, taken from an illuminating case study by ink&switch [33], is preparing a trip by gathering information about the destination (yin), using tools such as search engines or note-taking software, planning the details of the trip (transition to yang), and then leaving (yang). Western cultures tend to see this as a linear chain of steps that ends with the trip itself. But the trip inevitably causes more learning to happen, the individual acquiring new information that is likely to be exploited later on. Learning and acting happen in cycles that continue over the life of an inividual. Many such cycles are ongoing at any time, as we learn about and act on different aspects of the world around us.

Similar processes happen at the level of families, teams, associations, companies, cities, or states. All human social structures learn and act. At the scale of a culture, the phases stretch over long time periods, up to centuries. In Western culture, the most prominent yin process of the last centuries is called *science* and feeds yang processes such as *engineering* or *medicine*.

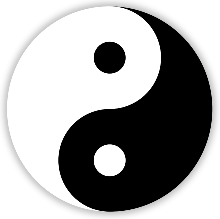

Figure 1. Yin-yang symbol (from Wikimedia)

Fig. 1 shows the most common contemporary graphical representation of yin-yang. The dark areas represent yin, the light areas yang. Imagine the hand of a clock superposed on this image. As it turns, the proportions of dark and light under it go up and down in a cycle. The yin and yang processes

[2]The acronym stands for "A Programing Language", which is the title of the book that presented this language [16]. APL was initially developed as a notation for purely human use, in developing and communicating algorithms. The book does not refer to the computer implementation of APL, which was developed later.

[3]The qualification as a *lens* is mine. In Chinese philosophy, yin-yang dualism is considered a principle of the universe that expresses itself everywhere. But if you ignore that belief and consider only how people use yin-yang in practice, it becomes a perspective taken, or a lens being applied.

[4]Traditionally called Tai-Chi in English.

have no clearly delimited beginnings and endings, they wax and wane.

The two opposite-colored dots in the black and white areas stand for the presence of *yin in the yang* and *yang in the yin*. They are a reminder that if you zoom in on a process, you will always see aspects of the opposite kind. In the trip example I mentioned above, learning about the destination requires actions, such as taking notes, that draw on earlier learning. That is yang in the yin. During the trip, the plan prepared in advance will be completed or even modified based on last-minute information. That is yin in the yang. In science, constructing an experimental setup is yang in the yin, whereas in engineering, learning from a prototype is yin in the yang.

3 Tools and Technology

By definition, technology is used for shaping the world, and therefore any use of technology implies a yang process. When I use pencil and paper to take notes during a lecture, the immediate role of the pencil is to leave marks on the paper, and thus change the world, even though note taking is part of the yin process of me remembering the subject of the lecture. When using the yin-yang lens, it is important not to mix multiple levels of granularity. There's always yin in the yang and yang in the yin, and it's easy to get lost in them. In this essay, I do not look at technology itself, nor at its development, but at the roles it plays in the processes of learning and doing by humans and human social structures. Pencil and paper are tools that *support* yin processes such as learning and understanding, just like they support yang processes such as writing laws.

Another relevant feature of technology is that its use requires prior learning and continuous adaptation by the user. Pencil and paper support me learning philosophy only after I have learned how to write using pencil and paper. Moreover, while using them to take notes, I have to adapt my note-taking habits (speed, amount, etc.) to the characteristics of these tools. We tend to be aware of the learning effort, but much less of the continuous adaptation effort required by our tools. A famous quote attributed to Marshal McLuhan (but actually from an article by John Culkin *about* McLuhan's work [4]) says that "We shape our tools and thereafter they shape us". Tools shaping us is, from the human tool user perspective, a yin process, but one that is imposed rather than chosen. There is also an imposed yang aspect, which consists of the unintended and potentially undesirable side effects of using a tool, including the use of material resources and the generation of waste.

4 Automation

The key aspect in my analysis is automated information processing, via algorithms and their implementations, or via machine learning techniques. Low-automation software, such as basic word processing or drawing tools, are not very different from pencil and paper in supporting yin and yang processes equally well. Their main function is to implement a new medium for information storage. Automation is what makes the difference between a *tool* and a *machine*. It introduces *epistemic opacity*, i.e. the impossibility for a user of the machine to fully know and understand the machine's behavior in detail, which is a topic I will come back to in section 7.

The construction and deployment of machines, be they physical machines or software, are yang processes, but as I explained for pencil and paper, this is not the aspect I am interested in. My focus is on how these machines or their construction support yin and yang processes in other domains. An example of a machine supporting a yin process is a DNA sequencer, which automates the chemical procedures required to obtain information about the genome of an organism. An example of a machine supporting a yang process in a different domain is a photocopier used to automate the distribution of teaching material to a class of students, the supported yang process being teaching.

The foundation of automation in information processing is formalization, i.e. encoding knowledge as expressions and rules of a formal system. Often there are aspects that cannot be formalized but encoded as variables in the formal system, for which values are then obtained by data-driven inference techniques. For a small number of variables, this is called parameter fitting, whereas for a very generic formal system with many variables, it is called machine learning. The variables are usually numbers but more general variants are possible, an example being genetic programming where the automatically adjusted values are algorithms.

There are two ways in which automation can support yin processes: computational modeling and the deployment of computational models in simulation and data analysis. Computational modeling is yin feeding the yang of automated processing: it consists of combining available information fragments into a formal system, potentially containing variables. Automated evaluation and analysis provide feedback about the model while it is built. Simulation and data analysis exploit computational models in the exploration of larger systems of which these models describe parts. This is a yin fed by the yang of automated processing.

While I am using a scientific vocabulary here, these concepts apply equally well to small-scale learning processes in everday life. Trip planning, an example I referred to earlier, is computational modeling (usually partial) of the trip. Embark, the trip planning tool described in [33], supports the modeling process by facilitating formalization, e.g. by identifying dates and geographical locations on which computations and database lookups can be performed. An everyday example for applying well-known computational models is consulting a weather forecast, which is the outcome of a very sophisticated simulation of the Earth's atmosphere.

5 Computational Modeling: Formalization

Formal systems, consisting of (1) a formal language that defines what a valid expression is, and (2) a set of rules for the transformation of expressions in that language, are a medium for representing information and knowledge in automated information processing systems. Formalization of knowledge, i.e. encoding knowledge in a formal system, is a major yin process in computing. Applying a formal system, e.g. by running software, is the yang process fed by formalization.

Formalization of knowledge comes in many varieties, not all of which are easy to spot as such. Unicode is a formal system, with rules for operations such as capitalizing letters. The simple act of typing text into a computer is thus already formalization. Numbers with associated arithmetic operations are a formal system as well, meaning that quantification is formalization. Spreadsheets are among the most popular formalization tools today. Adding machine-readable markup (e.g. HTML tags) to a text is also formalization, as is data wrangling resulting in a table stored in a CSV file. These are all examples for lightweight formalization, in which the formalized information is not very different from the corresponding informal expressions in plain written language, but more amenable to automated processing. At the other end of the scale, we have program source code, formal specifications for large software systems, or mathematical statements formalized for use with a proof assistant. In these cases, the formalized version is hard to recognize even for domain experts if they are unfamiliar with the formalization tools.

Computational formalization never starts from scratch. It happens in a software environment that has a strong imposed yin aspect: users need to learn the environment and adapt their way of thinking to it, in particular by adopting its notations. During the formalization process, a good support system should provide a rapid feedback loop. The user should be able to construct the formal system in small steps, and receive immediate feedback on the consequences of each change. This feedback includes the application of the transformation rules of the formal system, but also verification of conditions, consistency checks, visualizations, and connections to repositories of established knowledge, such as databases.

In the following, I will discuss various formalization tools and techniques with respect to their suitability for supporting yin processes. I will limit myself to techniques that can integrate automation. This means that I exclude most formalization tools actually used for yin processes today, such as note taking software or semantic Web tools. These tools implement media that are limited to inert data, whereas the automated processing happens in software external to the media that users cannot easily change as part of their yin process. The fact that such restricted media dominate the yin support of information technology today is a symptom of the problem I wish to expose.

5.1 Programming Languages

Programming languages are the formal systems that have received by far the most attention, both by academic researchers and by software developers. They have a strong yang focus on the automated transformation (algorithms, programs) of formal expressions (data), whereas statements *about* data, which are not executable, are neglected. The one exception is statements used in a particular form of consistency checks known as static type checking. Even assertions, which superficially look like statements about data, are really instructions for execution. They say "ring a bell if this property doesn't hold", which is much less valuable than a queryable database for such properties.

The stated goal of programming languages is to support the creation^{yang} of programs that are executed^{yang} in order to do^{yang} something. Creating programs requires the prior formalization^{yin} of the information being processed and of the processing rules, but neither programming languages nor the development environments proposed for them provide much support for this formalization phase.

5.2 Programming Systems

Programming systems, whose differences from languages have been discussed in detail by Gabriel [8], support the iterative construction of a formal subsystem in a rapid feedback loop. This is a huge advantage for the human user performing formalization work.

Programming systems have been recognized for a long time as well adapted to exploratory programming [31], i.e. the construction and evaluation of prototypes as part of the design phase of software systems. Design lies at the transition from yin to yang: the goal of building an artefact is already there, but many of its detailed characteristics remain to be defined based on experiments with prototypes. A recent testimonial [23] expresses the advantage of programming systems in this phase concisely:

> Those who follow me know that GNV is written in Common Lisp. A great learning exercise for me. It's incredibly suited for this task because I could modify the code for numerous edge cases _while_ the crawler is running, without starting from scratch every time an exception is thrown. Sure, when the project is finished and the problem is modeled out properly, one could say: that would be easy to write in lang X.Y.Z! Sure, but it's not the end product where Common Lisp shines, it's the journey.

As Gabriel notes [8], research on programming systems has almost stopped in the 1990s. The three main families, Lisp, APL, and Smalltalk, have their origins in the 1950s to

1970s. They all have actively developed and used descendants, but mainstream research and development has shifted to programming languages.

5.3 Specification Languages

Specification languages are formal languages that are designed specifically to support the formalization process. Specifications have two properties that make them more suitable for yin processes than programs:

1. They can be written as a set of small independent fragments. A specification does not have to be complete in the sense of fully defining an algorithm or a program. Partial specifications can be analyzed and explored, e.g. using a model finder. Any number of specifications can be lumped together to make a larger specification, facilitating exploration.
2. There is no requirement of executability. A specification can encode constraints on and metadata about any kind of formal expression. More generally, a specification does not have a built-in purpose, contrary to a program that is considered defective if it is not executable. However, nothing prevents the inclusion of a ready-to-run algorithm in a specification, meaning that specifications are strictly more general than programs.

Specifications are to algorithms as mathematical equations are to mathematical functions: a more general, less constraining framework to express properties of formal systems, at the price of not being immediately exploitable for computing values [12]. In science, the introduction of equations has led to an enormous increase in the power of scientific models. For example, early astronomers were concerned with predicting the future positions of heavenly bodies from past observations, by fitting generic geometrical shapes (circles, epicycles) to these observations in what can be qualified as an early form of data science. For the yang purpose of making predictions, this was good enough. Newton's equations [24] described the same orbits as solutions of a set of equations that express more fundamental physical laws. These laws turned out to be valid in very different settings as well. For two centuries, they were considered the foundation of all science, and the idea of deterministic laws underlying the whole universe, pioneered by Newton, remains a widespread (though tacit) metaphysical belief among scientists and technologists.

Fortunately, the mathematicians and scientists of past centuries were less blindly focused on solving equations than today's information technologists are on turning specifications into programs. Reasoning *about* Newton's equations lead to significant fundamental insight into the nature of physical processes, such as the discovery of the principle of energy conservation. Specifications hold the promise of enabling similar progress in the space of computational models. I have been pursuing a line of research in this space, rebranding specification languages as *digital scientific notations* [11] to emphasize the different intent. In fact, a specification is always *for* something and that something is usually a software system. In contrast, a digital scientific notation encodes information and knowledge independently of any specific goal that authors or readers might pursue.

At this time, specifications and associated formal methods are a respected topic of academic research, but have a very limited domain of practical application. They are seen as a powerful but also difficult to use tool whose deployment cost is justified only in the construction^{yang} of particularly large or safety-critical software systems. As Krishnamurthi and Nelson point out [22], there is a lot of potential for wider use of specifications and formal methods if tools and techniques are adapted to the needs of humans rather than mathematicians and computer scientists[5].

5.4 Static Type Systems and Type Checkers

Both programming and specification languages have type systems, which are often static, meaning that the types of expressions can be derived from the source code without having to perform any of the automated transformations. Type systems are formal systems used to describe properties of programs or specifications and relations between those properties, with the goal of permitting the automated validation of type relations.

Like programming languages, today's type systems are extremely yang-focused. Their two main roles are detecting potential mistakes in the composition of expressions and facilitating code optimizations. One characteristic following from this objective is that every expression must have exactly one type. Another characteristic is that there can only be one type system for a complete program or specification. Authors have to adapt their code to the type system, to the point of not being allowed to write correct code unless the type system can be used to *prove* its correctness. There are of course good reasons for this, but they are valid only in the yang context of constructing large software systems.

Gradual typing is an approach that aims at bridging formalization work and software construction. It removes one of the totalitarian aspects of static type checkers by allowing authors to opt out of their services for parts of the code. But the typed part of the code is subject to the same rigid rules as in traditional full-program type checking. I suspect that this is a major reason why gradual typing has not been widely adopted so far.

However, static type systems could be helpful in the yin process of formalization, by guiding authors towards well-formed expressions and supporting exploration tools such

[5]It is often assumed that mathematicians and computer scientists *are* humans, but I am not aware of any peer-reviewed research that supports this hypothesis.

as model finders, and by documenting interfaces in a precise way that is both human-readable and machine-readable. We can see how yin-friendly type checking could work by considering an example that is in fact older than computers: dimensional analysis for physical quantities, which has been used to validate quantity arithmetic since the 19th century.

Dimensional analysis applies to physical quantities only, simply ignoring any other mathematical objects. It can be extended as needed, by introducing additional dimensions. It also coexists with other formal validation systems, such as the rank formalism of tensor algebra. As a simple example, in classical mechanics, angular velocity ω, angular momentum L, and inertia I are related by the equation $L = I \cdot \omega$. Dimensional analysis says that ω has dimension 1/time, I has dimension mass $\cdot$ length^2, implying that L has dimension mass $\cdot$ length^2/time. Independently, tensor algebra asserts that ω has rank 1, I has rank 2, implying that L as the contraction between them must have rank 1.

A yin-friendly type checker would manage multiple type systems and admit any number of type annotations, including none, on any datum or variable. It would also allow users to define their own compatibility and transformation rules for types. The type checker would then annotate the code with warnings, but not require fixing those problems immediately. A type inference engine would guide in the construction of valid expressions, and highlight the rules violated by any invalid expression.

6 Computational Modeling: Data-based Inference

Data-based inference has become highly fashionable with machine learning in recent years. It sounds like a yin process, because it says "learning", but if it's the machine doing the learning, it is not a yin process in the scope of my analysis. Can data-based inference support *humans* in learning, understanding, and trusting?

At small scales, i.e. fitting a few parameters in non-trivial formal systems, the answer is yes, judging from centuries' worth of experience with mathematical models in science. The parameters have been chosen carefully during formalization, and should have a meaning to anyone who is sufficiently familiar with the formalized parts. In the example of Newtonian celestial mechanics I have used in section 5.3, the fit parameters are the masses of the planets and the sun. Their physical meaning is understandable from experience with objects of everyday life.

At the machine learning end of the scale, the formal systems are intentionally constructed to be very general, in order to permit adaptation to a wide range of data. Nielsen has presented a nice visual proof showing that neural networks can compute any function [25, chapter 4]. For the yang process of computing the function, that's sufficient. For humans to learn from the AI training process, the parameters of the machine learning model need to be interpretable, individually or as a whole. For neural networks, this remains an active field of research.

7 Simulation and Data Analysis

Simulation and data analysis are yang processes that often feed yin processes of learning. In order to learn from their results, users must (1) have a mental model of the computation that permits them to apply it correctly and to interpret the results, and (2) trust the software to conform to this mental model. There are various ways in which users can gather the information for building their mental models^{yin} and build trust^{yin} into their validity: inspect program source code, run programs on well-known inputs, modify the code to observe how the outputs change, compare with results from different programs, etc.

There are two ways in which computational results can support learning processes. One is deducing consequences from models and comparing them to observations, for testing the less trusted part (model or observations) against the more trusted one. The other one is as a generator of ideas or hypotheses, as a form of externalized creativity. In that situation, computed results can be useful even if obtained from a defective model, and the accuracy of the user's mental model is less important as well. This is why generative AI has found its place in scientific research. Nevertheless, the learning value of computation increases with the accuracy of the user's mental models. The most useful generative AI models in science are those derived from well-defined and well-understood databases, using well-documented constraints in the formalized part. A good example is the widespread use of AlphaFold, a deep-learning model for protein structures, in structural biology [3]. AlphaFold is used as an engine for interpolating and extrapolating from the Protein Data Bank [30], a 50-year-old database of experimental data that structural biologists are intimately familiar with.

We do not have any systematic techniques for building and validating mental models of software, let alone of machine learning systems. We remain at the stage of tinkering. For simple systems in familiar contexts, this works well enough. Most people figure out how a pocket calculator works with some practice, and then use it reliably. But even common programs such as word processors can be problematic. Most of us have experienced the anxiety of accidentally clicking on the wrong menu entry in a large word processor and wondering what it did to our document. And all heuristic approaches to building a mental model fail for software with intentionally hidden features, such as silently transmitting personal data from a smartphone to a data warehouse.

For the highly complex software used in scientific research, it is a good bet that nobody's mental model is complete. As a consequence, we should expect that such software is frequently used incorrectly, without anybody noticing. I suspect that this is an important factor in the ongoing reproducibility crisis [39] that touches many scientific disciplines, in particular for statistical inferences. Performing^{yang} a statistical test takes only a mouse click, whereas understanding^{yin} what the results mean requires a course in statistics followed by a study of the software's source code. Much progress has been made on teaching statistics to young researchers, but software source code remains a serious obstacle.

The difficulty of building and validating mental models of software is a major difference to yin-supporting machines in the physical world. Scientists who use DNA sequencers do have a good mental model of how such a device works and what its limitations and failure modes are. They can validate most aspects of that model by putting well-known samples into the machine and checking the outputs. Such an approach does not work for software because it tends to be vastly more complex than most physical devices. Even given the full source code for inspection, it is very hard to understand what exactly the software does. There is some hope that Large Language Models (LLMs) will one day help with this task, given their capacity to process large code bases better than humans do. However this requires that we first develop sufficient trust in LLMs performing this task correctly. Whereas this is imaginable in principle, with software developers judging the performance of LLMs interpreting their product, it doesn't look economically feasible today.

An added difficulty is that the dominant building material for software, Turing-complete programming languages, causes software to exhibit chaotic behavior under changes to the source code [9], and thus an large diversity of failure modes. Engineers building physical yin-supporting machines (e.g. scientific instruments) take care to avoid chaotic behavior, but software engineers do not currently have that possibility.

Philosophers of science call such issues *epistemic opacity* [13, 14]: the difficulty, or even impossibility, for the user to know what exactly goes on inside the machine that processes information. If you follow a set of instructions step by step, as in a cooking recipe, you do not apply rigid rules blindly and precisely. You interpret the steps in the specific context, you apply them as best as you can, inevitably making mistakes, but you are also aware of the uncertainties, risks, and mistakes, and constantly check the state of your work for symptoms of problems. In the end, you have a good understanding of what you have done and what each step contributed to the final outcome. If you delegate work to someone else, you lose much of that feedback, but you trust the person who does the work to be are careful as you would be yourself. But if you delegate something to a machine, you lose control once you have prepared the machine and pressed the "go" button. You don't get any feedback from the process, except for the final result.

I am aware of two approaches for dealing with epistemic opacity in computation. One approach is reducing it, by providing more feedback to the user. The other approach is reasoning *about* the computation, deducing characteristics of the results without following each individual step. Both approaches are used in software development, but are not easily accessible to software users, who are mostly expected to develop both a mental model and trust in the implementation from only documentation (informal non-executable prose) and interaction with the software.

The roles of *developer* and *user* have become more and more distinct over the last decades, in parallel with the development and growth of a software *industry*^{yang}. The programming *systems* of the past (see section 5.2) were made for what would be called "power users" today: people who would write at least the topmost task-specific software layer themselves, and were familiar with common development tools. Alan Kay, the chief designer of Smalltalk, had the even more ambitious goal of a personal "Dynabook" permitting everybody, even children, to write and modify software for their own use [18]. A software industry selling products to users has no interest in providing more information to users than is strictly necessary to operate the software, out of both commercial and legal considerations. Users are not supposed to inspect or modify the software. Opacity becomes a desirable feature. This is also the major obstacle to building trust in LLMs, whose construction for now requires means that only large corporations can deploy. For these corporations, it is more profitable to develop opaque system for a large number of uncritical consumers than to develop trustworthy systems for epistemically demanding applications.

The Open Source movement has not made much of a difference: it is all about sharing code among developers. As a side effect, users can inspect the source code as well, but facilitating inspection and modification by users is not a priority. In contrast, the Free Software movement does have the explicit goal of empowering *users*, but it has focused on the legal rather than the epistemic obstacles that users are facing.

In the next two sections, I will describe some possible approaches to better supporting users in developing good-enough mental models of software, and in developing trust in the implementation's correctness with respect to this mental model.

7.1 Increasing a Computation's Cognitive Surface

In analogy with material 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 tools, or with significant effort, its *cognitive bulk*. Note that comprehensible is not the same as visible. The source code of the software that underlies a computation may be very visible, in the case of Open Source software, but most of it doesn't make sense to a user with reasonable effort, and therefore belongs to the bulk. 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.

A reduction in epistemic opacity can thus be achieved by increasing the cognitive surface of a computation, relative to the bulk. An obvious idea is to divide the computation into smaller pieces and let the user see their inputs and outputs. Interactive read-eval-print loops (REPLs) serve this purpose, for power users with sufficient programming experience. Object inspectors, as known from Lisp and Smalltalk systems, are very useful complements to a REPL. Computational notebooks go one step further and integrate code snippets and their inputs and outputs into an explanatory narrative. Whereas authoring such a notebook still requires power user level, a well-written notebook can be read and understood by less experienced users. Some notebook tools, such as Jupyter with its widgets, allow the embedding of visualization and control elements into a notebook, which invite the user to explore data items in the computation.

Notebooks draw on the earlier idea of literate programming [20], but apply it to a computation, i.e. a sequence of situated[6] code snippets processing specific data, rather than to a program, which consists of generic code applicable to different inputs. This is their strength but also their limitation. In any non-trivial computation, the situated code snippets call reusable generic software libraries, to which the reader of a notebook has no access.

In an essay on making software systems explainable [26], Nierstrasz and Girba propose a combination of notebooks and literate programming, which is implemented in a Smalltalk-based programming system called Glamorous Toolkit [7]. Instead of a single notebook outlining the steps of a computation, they propose multiple cross-linked narratives. Each such narrative can contain the steps of a computation, but also visually embed code from anywhere in the software system. Another improvement in Glamorous Toolkit is the replacement of the traditional Smalltalk object inspector by a *moldable* object inspector that can be extended with domain-specific views and user interface elements, similar in spirit to Jupyter widgets but much cheaper to create, thanks to rich support from the underlying programming system.

As these examples illustrate, softening the divide between users and developers is an important aspect of increasing a computation's cognitive surface. In such a scenario, users must invest more effort into learning computing technology, becoming power users. In parallel with the power and understanding they gain, they also take more responsibility for the software's behavior. Such power users require very different support tools than the immensely complex programming languages and build systems created by and for software professionals.

[6]See [32] for a discussion of situated software.

Taking this idea one step further, we should question if generic software building blocks written for a wide range of applications are always the best design to aim for. Reusable software is a concept from the software industry^{yang}, where it serves productivity^{yang}. In an interview from 2008 [21], Donald Knuth said:

> 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.

Unfortunately, he didn't develop this thought beyond this one paragraph. I interpret his proposal of "re-editable code" as code that is kept as simple and situated as possible, such that readers can understand it and then adapt it to their own needs. Much like with Alan Kay's Dynabook, and in the spirit of Ivan Illich's call for convivial tools [15]. On the other hand, reusable software is very convenient if you want a reliable off-the-shelf component but don't actually care about its inner workings. Taeumel and Hirschfeld [36] discuss the trade-offs between reusable and re-editable code (which they call *repairable*) with several examples.

7.2 Reasoning about Software

As every software developer knows, reasoning about software is hard, even if it's software you have written yourself. Reasoning about someone else's software is worse, because the source code is a very imperfect representation of a developer's thought processes. Program source code is not the right medium for humans to reason about software. Could we do better? I'll present a raw idea, not being sure that it would work out in practice. It is inspired by a popular quote from the well-known computer science textbook "Structure and Interpretation of Computer Programs" [1, preface to the first edition]:

> Programs must be written for people to read,
> and only incidentally for machines to execute.

How would a user go about reading a non-trivial program? The first contact with new software is its documentation, so let's start from there. We could embed a formal specification (see section 5.3) into this documentation, permitting readers to validate the mental models they are forming, possibly with the help of formal methods. Executable examples would help as well.

Many users could stop at that level, if they have reasons to trust the implementation. For those interested in diving in deeper, a second software layer could contain a high-level implementation of the specifications, commented for an explanation of implementation choices. The algorithms would be fully worked out, but no optimizations nor adaptations made for purely technical reasons.

A code analyzer could annotate this layer with possible optimizations, telling the reader not to worry about these points because they will be taken care of automatically. A third layer would then contain the human response to those annotations: manual optimizations and fine-tuning to technical requirements. That layer would become the "source code" for machines to execute.

The challenge is to keep all three layers coherent as they evolve. That's certainly not a trivial task, but it doesn't look impossible either. For humans, such an architecture should be well-adapted because it is how we lay out explanations of complex topics in textbooks: start with an overview and specific examples, then go progressively into technical details.

7.3 Trusting Software

Assuming that I have a good-enough mental model of what a piece of software does, how I can develop trust in the correctness of its implementation? If the mental model of its operation is simple, then I can evaluate the implementation from feedback during usage. But what if I cannot possibly test every feature myself? They might be too numerous, or exploring them might be dangerous or expensive.

Such questions are not specific to software. When I take a train, I don't ask the conductor to test the brakes at high speed in order to convince me of their proper function. Trains are embedded into a network of manufacturers, operators, maintenance technicians, laws and regulations, etc., all of which contribute to building public trust into a complex industrial product. The corresponding trust mechanisms for software are much weaker. Much of the software we use every day, and all of the software we use for science, receives little to no independent audits and is not subjected to any norms or regulations.

This is, of course, a social rather than a technical problem. We will get more trustworthy software as soon as we make it a priority and are willing to pay the cost. I have outlined possible measures for scientific software elsewhere [10], but don't expect them to be applied any time soon. For now, the scientific community, in its quest for productivity^{yang} and impact^{yang}, wants to do^{yang} ever more, but not examine^{yin} if these actions are appropriate.

There is, however, one very easy to apply measure to facilitate trust building by users: slow down development. Trust building is a slow yin process. It's impossible to build trust in a system that is modified so frequently that nobody outside the development team has the time to examine it carefully.

8 What We Have Lost

Now I can go back to my motivating examples from the introduction, and describe what exactly we have lost during thirty years of yang-focused innovation in computing technology.

APL was initially designed by Ken Iverson as a notation for humans working in applied mathematics [16, 17]. It was later implemented, initially by IBM, as a programming system for explorative work. The APL systems available in the 1990s were already quite sophisticated, with IBM's APL2 in particular excelling in support for plotting and visualization, as well as providing more flexible fundamental data structures through nested arrays.

In contrast, a Jupyter notebook as an interface to Python code is an assembly of yang-focused technologies. Python was initially developed as a scripting language for systems administration [38]. A library for numerical computations, drawing inspiration from APL and other sources, was written later as a separate entity [5]. It implemented its own efficient central data structure, the multi-dimensional array, meaning that programmers have to keep juggling between Python's native data structures and NumPy's add-on data structure all the time. The Python language itself evolved significantly over its more than thirty years of existence, becoming more and more complicated and reserving some unpleasant surprises for its users [27]. While Python's initial learning curve is attractive, compared to a superficially cryptic language like APL, the ongoing cognitive load in terms of technical details that it imposes on its users is significant.

Jupyter provides an interface to Python[7] that superficially resembles an interactive programming system, but is restricted to a linear sequence of code, much like a script. The Python libraries used in a notebook can neither be inspected nor modified interactively. Users wishing to understand the library code they deploy in detail have to switch to a different set of tools. Jupyter's two-process architecture, with a computational core in Python and an interactive layer written in JavaScript, running in a Web browser, adds another level of cognitive burden on users and discourages in particular explorative work on visualization, whose implementation requires a Python layer, a JavaScript layer, plus a communication layer between the two. On the other hand, Jupyter comes with a large range of ready-to-use^{yang} predefined visualizations, which makes it superficially attractive for visualization work – assuming that you are willing to adapt your work to someone else's visualization choices, rather than adapt the visualization environment to the needs of your work.

Whereas both APL and Jupyter were designed as support technologies for yin processes, Jupyter relies on two major yang-focused building blocks: Python and the Web platform. No amount of interface polishing can make up for the inappropriateness of such a foundation. What we have lost at the most fundamental level is the goal of achieving widespread

[7]Also a few other languages, but I concentrate on the most widely used one.

computational literacy. We have abandoned not only Alan Kay's ambitious goal of computational literacy for everyone, starting at a young age, but even the more modest goal of Ken Iverson: computational literacy for highly trained professionals in knowledge-intensive domains such as scientific research.

The story of my Fortran simulation code is a bit more subtle. It is more understandable than many of today's simulation codes not because of building on more suitable technology, but mostly by virtue of being smaller. For someone familiar with the problem and the mathematical toolbox used to deal with it, studying the entire code in a few days is quite feasible. Today's more powerful computers allow us to study more complex problems, using more complex models and more complex code. There is real scientific progress in this development. But if we had been serious about supporting the yin process of science, we would have worked hard to improve understandability in parallel with implementing more complex models. We would systematically have subjected software to peer review, to ensure that it makes sense to someone else than its authors, and to build trust in it. The idea of publishing and reviewing code was first proposed in 1969 [29], but received little attention before the reproducibility crisis. What we have lost in the quest for yang is the core principle behind scientific research: a self-critical attitude that insists on transparency and external verification for every contribution to the scientific record.

9 The Wider Context

In the preceding sections, I have documented a systematic emphasis on yang processes in information technology, with a corresponding weakness of support for yin processes. This imbalance is not limited to information technology. Similar trends can be observed in other aspects of today's Western and globalized societies.

Scientific research is a quintessential yin process, whose goal at the most abstract level is to increase humanity's understanding of the world. And yet, it has been invaded by yang ideas and terminology over the last decades. Research is being re-branded as knowledge *production*^{yang}. Journal articles, whose original role was communication between researchers, are now considered to be research *outputs*^{yang}, along with other countable and measurable artifacts such as datasets or software. These outputs are expected to have *impact*^{yang}. The production of high-impact outputs is overseen by research *managers*, as in industry. Whereas in the not too distant past, science was supposed to contribute to progress, today's terminology is *innovation*. Change for change's sake, without the, admittedly vague and subjective, notion of improvement that progress implies.[8]

Taking another step back, the priority of today's governments is economic *growth*^{yang}, meaning the continuous increase of the *production*^{yang} of goods and services. It no longer matters if these goods and services are beneficial or harmful, as such judgments would require a prior understanding of each product's context, followed by a public debate. A good example is the rapid deployment at scale of social media in the 2010s and of LLMs right now. They *disrupt*^{yang} human communication, including the inherently slow yin processes of trust building. A responsible introduction of such technology would have progressed slowly, collecting feedback along the way.

In Chinese philosophy, and in particular in *daoism*[9], the yin-yang lens is used to watch over harmony. Yin and yang require each other. If either one dominates, the system is out of balance and at risk of malfunction. From that perspective, many of the problems we are facing today are the result of an excess of yang. We focus on productivity and innovation, but don't watch out for the impact of our technology and its products on the biosphere, including its human inhabitants. We burn ever more fossil fuels, but close our eyes to the resulting global temperature increase, although its basic mechanisms have been known for more than a century [2, 6]. We release ever more synthetic molecules into the environment, but refuse to see their toxicity for living organisms. The primary societal reaction to these environmental issues so far has been activist^{yang} movements, who call for emergency action^{yang} without considering the potential negative effects that this yang-against-yang might have. Daoism has always been suspicious of the unintended and undesirable consequences of human action that result from insufficient alignment with the flow of natural processes. It advocates *wuwei*, often translated as effortless action, and nicely illustrated by the example of moving with the flow of a river rather than against it. This is the opposite of "disruption" and "move quickly and break things", the Silicon Valley attitude that has contaminated much of today's information technology.

While I do not fully adhere to daoist precepts, I do believe that today's yang obsessions in technology, economy, and culture are related and interdependent, and that we should counterbalance them by strengthening our yin processes. One easy first step I have taken is to refuse the use of yang terminology for yin concepts. I am a researcher, not a knowledge producer. I seek insight, not impact. I make contributions to a knowledge commons, not research outputs. And on my modest personal scale, I also try to improve yin-supporting computing technology [11]. I hope that some of my readers will join me in this quest for balance.

[8]A quest for progress is also problematic because it is usually applied to a narrowly defined aspect of the world, ignoring side effects outside of that focus that may well be deleterious. See [28] for an in-depth discussion.

[9]The traditional English spelling is *taoism.*

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 2002. *Structure and Interpretation of Computer Programs* (2. ed., 7. [pr.] ed.). MIT Press [u.a.], Cambridge, Mass.

[2] Svante Arrhenius. 1896. On the Influence of Carbonic Acid in the Air upon the Temperature of the Ground. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 251 (April 1896), 237–276. https://doi.org/10.1080/14786449608620846

[3] Elizabeth A. Campbell, Helen Walden, Johannes C. Walter, Arun K. Shukla, Martin Beck, Lori A. Passmore, and H. Eric Xu. 2024. AlphaFold: Research Accelerator and Hypothesis Generator. *Molecular Cell* 84, 3 (Feb. 2024), 404–408. https://doi.org/10.1016/j.molcel.2023.12.035

[4] John M Culkin. 1967. A Schoolman's Guide to Marshall McLuhan. *The Saturday Review* 51–53 (March 1967), 70–72. https://webspace.royalroads.ca/llefevre/wp-content/uploads/sites/258/2017/08/A-Schoolmans-Guide-to-Marshall-McLuhan-1.pdf

[5] Paul F. Dubois, Konrad Hinsen, and James Hugunin. 1996. Numerical Python. *Computers in Physics* 10, 3 (1996), 262. https://doi.org/10.1063/1.4822400

[6] Steve M. Easterbrook. 2023. *Computing the Climate: How We Know What We Know About Climate Change* (1 ed.). Cambridge University Press, Cambridge. https://doi.org/10.1017/9781316459768

[7] feenk GmbH. [n. d.]. Glamorous Toolkit. https://gtoolkit.com/

[8] Richard P. Gabriel. 2012. The Structure of a Programming Language Revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)*. Association for Computing Machinery, New York, NY, USA, 195–214. https://doi.org/10.1145/2384592.2384611

[9] Konrad Hinsen. 2016. The Power to Create Chaos. *Computing in Science & Engineering* 18, 4 (July 2016), 75–79. https://doi.org/10.1109/MCSE.2016.67

[10] Konrad Hinsen. 2023. Establishing Trust in Automated Reasoning. https://doi.org/10.31222/osf.io/nt96q

[11] Konrad Hinsen. 2023. Leibniz - a Digital Scientific Notation. https://leibniz.khinsen.net/

[12] Konrad Hinsen. 2023. The Nature of Computational Models. *Computing In Science & Engineering* 25, 1 (2023), 61–66. https://doi.org/10.1109/MCSE.2023.3286250

[13] Paul Humphreys. 2004. *Extending Ourselves: Computational Science, Empiricism, and Scientific Method.* Oxford University Press, New York, US.

[14] Paul Humphreys. 2009. The Philosophical Novelty of Computer Simulation Methods. *Synthese* 169, 3 (2009), 615–626. https://doi.org/10.1007/s11229-008-9435-2

[15] Ivan Illich. 1973. *Tools for Conviviality.* Calders and Boyars, London.

[16] Kenneth E. Iverson. 1962. *A Programming Language.* Wiley, New York.

[17] Kenneth E. Iverson. 1980. Notation as a Tool of Thought. *Commun. ACM* 23, 8 (Aug. 1980), 444–465. https://doi.org/10.1145/358896.358899

[18] Alan C. Kay. 1972. A Personal Computer for Children of All Ages. In *Proceedings of the ACM Annual Conference - Volume 1 (ACM '72, Vol. 1)*. Association for Computing Machinery, New York, NY, USA. https://dl.acm.org/doi/10.1145/800193.1971922

[19] Donald E Knuth. 1974. Computer Science and Its Relation to Mathematics. *The American Mathematical Monthly* 81, 4 (1974), 323–343. https://doi.org/10.2307/2318994 jstor:2318994

[20] Donald E Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. http://comjnl.oxfordjournals.org/content/27/2/97.short

[21] Donald E Knuth and Andrew Binstock. 2010. Interview with Donald Knuth. https://web.archive.org/web/20101203111941/https://www.informit.com/articles/article.aspx?p=1193856

[22] Shriram Krishnamurthi and Tim Nelson. 2019. The Human in Formal Methods. In *Formal Methods – The Next 30 Years*, Maurice H. Ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Vol. 11800. Springer International Publishing, Cham, 3–10. https://doi.org/10.1007/978-3-030-30942-8_1

[23] @louis@emacs.ch. 2024. GNV - the Gopher Web Search Engine - Just Got an Update. https://emacs.ch/@louis/112074295465231785

[24] Isaac Newton. 1687. *Philosophiæ Naturalis Principia Mathematica.* Royal Society, London. https://la.wikisource.org/wiki/Philosophiae_Naturalis_Principia_Mathematica

[25] Michael Nielsen. 2015. *Neural Networks and Deep Learning.* Determination Press. http://neuralnetworksanddeeplearning.com/

[26] Oscar Nierstrasz and Tudor Girba. 2022. Making Systems Explainable. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, Limassol, Cyprus, 1–4. https://doi.org/10.1109/VISSOFT55257.2022.00009

[27] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications.* ACM, Indianapolis Indiana USA, 217–232. https://doi.org/10.1145/2509136.2509536

[28] Consilience Project. 2024. Development in Progress. https://consilienceproject.org/development-in-progress/

[29] K. V. Roberts. 1969. The Publication of Scientific Fortran Programs. *Computer Physics Communications* 1, 1 (July 1969), 1–9. https://doi.org/10.1016/0010-4655(69)90011-3

[30] P W Rose, A Prli, C Bi, W F Bluhm, C H Christie, S Dutta, R K Green, D S Goodsell, J D Westbrook, J Woo, J Young, C Zardecki, H M Berman, P E Bourne, and S K Burley. 2014. The RCSB Protein Data Bank: Views of Structural Biology for Basic and Applied Research and Education. *Nucleic Acids Research* (2014). https://doi.org/10.1093/nar/gku1214

[31] D. W. Sandberg. 1988. Smalltalk and Exploratory Programming. *ACM SIGPLAN Notices* 23, 10 (Oct. 1988), 85–92. https://doi.org/10.1145/51607.51614

[32] Clay Shirky. 2004. Situated Software. https://web.archive.org/web/20040411202042/http://www.shirky.com/writings/situated_software.html

[33] Paul Sonntag, Alexander Obenauer, and Geoffrey Litt. 2023. Embark: Dynamic Documents as Personal Software. In *LIVE 2023: The Ninth Workshop on Live Programming*. Cascais, Portugal. https://www.inkandswitch.com/embark/

[34] Gerald Jay Sussman and Jack Wisdom. 2002. *The Role of Programming in the Formulation of Ideas.* Technical Report. MIT Artificial Intelligence Laboratory. 1–19 pages. http://hdl.handle.net/1721.1/6707

[35] Gerald Jay Sussman and Jack Wisdom. 2014. *Structure and Interpretation of Classical Mechanics* (2. ed ed.). MIT Press, Cambridge, Mass.

[36] Marcel Taeumel and Robert Hirschfeld. 2022. Relentless Repairability or Reckless Reuse: Whether or Not to Rebuild a Concern with Your Familiar Tools and Materials. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2022)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/3563835.3568733

[37] A M Turing. 1937. On Computable Numbers, with an Application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society* 42, 2 (1937), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[38] Guido van Rossum. [n. d.]. Why Was Python Created in the First Place? https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place

[39] Wikipedia contributors. 21 April 2024 20:38 UTC. Replication Crisis. *Wikipedia* (21 April 2024 20:38 UTC). https://en.wikipedia.org/w/index.php?title=Replication_crisis&oldid=1220102316