# Technical Debt in Computational Science

Konrad Hinsen

# Technical debt in computational science

Konrad Hinsen

Technical debt is a recent metaphor that has been rapidly adopted by the software industry. It was first used by Ward Cunningham in 1992 in a report on a software development project (http://c2.com/doc/oopsla92.html). The term refers to future obligations that are the consequence of technical choices made for a short-term benefit. The standard example is writing sub-optimal code under time pressure, knowing that the code will have to be refactored or rewritten later in order to make the software maintainable. This effort, which does not improve the software's utility for its users and therefore does not add market value to it, serves to pay back the debt.

The word "debt" emphasizes the analogies to monetary debt. Both are future obligations incurred in exchange for a short-term benefit. But the analogy goes further: both generate interest. In the example of the hastily written code, any work done on it before refactoring or rewriting, be it for fixing bugs or for quickly adding features, will require more effort than it would for well-written code. It is also probable that much of this work will have to be repeated after paying back the debt. The additional effort is the equivalent of paying interest on a debt. Another useful analogy is debt default: defaulting on a technical debt is lowering one's quality standards, accepting that an objective cannot be met because of a bad technical choice in the past. For a company, that can mean the end of a product line, or in the worst case the end of the company itself.

Just like a financial debt, a technical debt is not necessarily a bad thing. There can be good reasons for cutting corners and fixing the resulting problems later. For a company, being the first to propose a product on the market is a competitive advantage that can procure long-term benefits. Similarly, a scientist can derive a significant benefit from being the first to publish an important new result. The point of the technical debt metaphor is not to reprehend such choices, but to remind of the long-term consequences.

Like all analogies, the debt metaphor has its limits. A financial debt is the result of a contract between a borrower and a lender that describes the exact conditions of the debt. Unless you carelessly take a loan without reading the contract, you know what your future obligations are, and what short-term benefits you get in return. Technical debt results from a contract with your future self, and its terms are usually not written down anywhere. An experienced engineer will recognize having incurred a technical debt, but may not be able to give a precise

estimate of the interest and the final payback. An inexperienced person can even incur technical debt without being aware of it at all, seeing the short-term benefit but not the long-term obligations.

A simple Web search will quickly yield many examples of and discussions about technical debt in the context of commercial software development. Much of this applies to scientific software as well, in particular to larger and long-lived software projects with multiple developers and some form of project management. However, both the nature of the software projects and of the organizations behind them is much more diverse in scientific computing. In particular, much software development happens in relatively small research groups that have informal collaborations with other such groups, either on a common software package or on distinct but interdependent software packages. In such an organization, anyone's technical debt has an impact on everyone else.

I will illustrate this with examples from the scientific Python ecosystem, the term commonly used to describe the large set of scientific libraries written in the Python language. It has an onion-like structure, with the Python language itself at the core. The next layer contains a small number of scientific infrastructure libraries such as NumPy (array computations) and matplotlib (plotting). The third layer consists of domain-specific libraries which tend to depend on libraries in the infrastructure layer, and sometimes depend on other items in the domain-specific layer as well. Outside of these three layers, we have "client code": scripts and workflows that are specific to a research project, but also highly domain-specific software tools with graphical user interfaces.

An important event that currently receives much attention in this ecosystem is the transition from Python 2 to Python 3. This transition is a nice example of paying back technical debt with a partial default. The Python language had continuously evolved over the years, acquiring both new language features and new modules in its standard library. The desire to keep each version backwards compatible with earlier versions led to redundant features that made the language needlessly complicated. For example, there were "old-style" and "new-style" classes with subtly different behavior. Everyone agreed that new-style was better, but old-style was there before and much existing code relied on it. Similarly, the standard library had acquired redundant modules, whereas other modules had become obsolete in the sense that they relied on no longer maintained libraries or were specific to computing platforms that have long since been transferred to museums.

The reason why the transition to Python 3 is partly a repayment and partly a default is that it preserves one objective while violating another one. Python started out with the goal of being a simple and easy to learn language, and that objective was preserved with the general cleanup that led to Python 3. But publishing a programming language and encouraging people to use it implies the promise of not breaking all those people's code in the future. This tacit promise was broken with Python 3, which is incompatible in many details with earlier

versions. The two objectives being contradictory, the only way to maintain both would have been to stop any future evolution of the language. Most programming languages face this choice at some time, but most designers choose continuously accumulating complexity rather than cleaning up the mess. In other words, they default on the technical debt by giving up simplicity.

Looking at the same process from the point of view of the creators of the scientific libraries written in Python, we see that the technical debt in the development of the Python language has a direct impact on their work. With the Python development community moving on to Python 3, it is foreseeable that it will abandon Python 2 in the long run. Library authors thus have to choose: either they migrate to Python 3 as well, or they keep the Python 2 platform alive by taking over its maintenance. Both choices involve an additional effort. Doing nothing seems like a third option, but given the fast rate of change in computing platforms, it is probable that today's Python 2 will become effectively unusable within a few years. Moreover, hardly any scientific library is useful in isolation, so everyone's choice depends on the expected behavior of the authors of related libraries. At this time, the core infrastructure libraries and many of the bigger domain-specific libraries have initiated or even completed the transition to Python 3, while maintaining some level of compatibility with Python 2. On the other hand, many libraries with a smaller developer base remain in the Python 2 universe, lacking either the means or the motivation to move on.

In terms of the technical debt metaphor, we can say that choosing the Python language, or in fact choosing to base one's work on any dependency or tool controlled by someone else, creates technical debt. The short-term benefit is the immediate availability of a useful software component. The interest is the work required to adapt one's own code to changes in the dependencies, or alternatively to take on the responsibility for maintaining a version of those dependencies that remains compatible with one's own code. Paying back the debt would mean replacing the dependency by one's own code, but this is rarely done in practice. The technical debt resulting from dependencies is, in most cases, perpetual. Moreover, such debts are practically inevitable because not depending on other people's work, i.e. writing everything oneself, is not a realistic option. After all, even the computer's operating system is a dependency. One can, however, try to minimize "risky" dependencies as part of a strategy for managing technical debt. Matthew Turk has recently written about this question in this department [1].

The kind of technical debt involved here is perhaps the most frequent one in computing, even before the standard example of cutting corners to terminate a project as early as possible. It can be summarized as relying on immature technology. When you choose a programming language that's just a few years old, you should expect that nobody, not even its creator, has sufficient practical experience with it to have made all the right choices. Either the language will remain static nevertheless, and then probably fade from popularity quickly, or it will change, and become either messy or incompatible. In all these situation,

you have a maintenance problem with your code that relies on it. If you want to avoid this, you should choose a programming language that has been around for decades. Indeed, stability is one reason often cited for choosing the Fortran language. Of course, the same principle applies to other dependencies such as libraries. It's probably safe to bet on BLAS being around for many more years without incompatible changes, but the same cannot be expected of a recent implementation of the hottest algorithms of the day. This well-known problem of software becoming unusable because of changes in its dependencies is sometimes called "software rot". This is not a good metaphor, however. Software doesn't degrade in time. It's the foundations on which the software is built that change, and even they don't change by decaying, but as a side effect of improving. The software rot metaphor has led to the equally misleading term "software maintenance" for keeping software usable by adapting it to evolving environments.

In a fast-moving field such as computing, immature technology is the norm rather than the exception. We all work with immature technology every day, and we know it. My computer crashes about once per month, requiring a reboot. It asks me to install software updates, often labeled as security-critical, at least once a week. Broken Web links are a daily experience. It's safe to assume that scientific software is of no better quality, even though the symptoms of bugs are usually more subtle and can go unnoticed. For scientists, who by definition work at the frontiers of knowledge and technology, there is probably no way to avoid immature dependencies. We can, however, be aware of this and try to anticipate the consequences, or at the very least not pretend that there are none.

The technical debt metaphor is most frequently applied in software development, but it applies equally well elsewhere. An interesting example is a recent exploration of the impact of data dependencies in applications of machine learning techniques [2]. Such a systems-level view of technical debt is also useful in the context of scientific research. In the following, I will apply this point of view for the specific situation of computational science.

Science has long-established standards of quality, which all scientists have the moral obligation to respect. In particular, scientists should make a serious effort to verify the results they obtain, actively searching for potential mistakes, in order to overcome confirmation bias, the natural tendency of human beings to search confirmation rather than refutation of their own hypotheses. Moreover, scientists must publish detailed accounts of their work in order to permit their peers to verify it, attempt to reproduce the findings themselves, and to build on it in future research. The respect of these obligations makes the difference between a scientific result and anecdotal evidence.

Verifying one's own results and conclusions implies first of all acquiring a sufficient understanding of one's methods and tools prior to using them, and ensuring that they are adequate for the task. Computational scientists have traditionally been rather negligent about this. The few prominent cases of mistakes in scientific results due to bugs in software that have been brought into the public eye [3]

are probably just the tip of the iceberg, and suggest a widespread lack of testing. Moreover, scientific software is often applied incorrectly, due to a lack of understanding of the computational methods that the software implements [4]. This is partly the fault of scientists using software they do not understand, but partly also the fault of scientific software authors providing insufficient documentation and neglecting the readability of their source code.

The word "negligence" already suggests that basic human tendencies such as laziness are an important cause for these problems. However, there is also a technical aspect to it. Scientists increasingly treat computational methods as similar to experimental ones, and consider computers and software as the theoretician's equivalent of experimental equipment. This point of view is useful in particular for simulation techniques, which produce data that is analyzed and evaluated in much the same way as experimental measurements, with a strong emphasis on statistical approaches. There is, however, a fundamental difference between computers and instruments used in experiments. Lab instruments, like any physical devices, are subject to inevitable imperfections in manufacture. They are therefore designed in such a way that small imperfections can only cause small deviations in the results. Computers, on the other hand, are chaotic dynamical systems. Changing a single bit in a computer's memory can change the result of a computation beyond any predictable bound. Computers are practically usable devices in spite of this sensitivity because of their extreme reliability, compared to other technical artifacts. Although hardware errors can become a problem with long-running computations on very large machines, for most applications of computers in scientific practice it is safe to assume that the computer does precisely what the software tells it to do. However, errors in the software or in the input data are amplified with each computational step. Often we can (and do) ensure that small errors in the input data translate to small deviations in the results, by a judicious choice of numerical methods. But we do not yet have good techniques for limiting the impact of software errors. We should therefore add the use of chaotic devices for computation to our technical debt account, and accept the effort for carefully testing our software as an inevitable interest payment, hoping to be able to pay back the debt one day by a profound change in the way computers are used in research that limits the impact of chaotic behavior. Since most scientists are not aware of this fundamental difference between software and the physical devices used in experiments, this particular debt resembles a loan taken without reading the contract.

The reproducibility requirement of science implies the publication of a sufficiently detailed description of what was done. Computational science has been performing very badly in this respect as well. This problem has received a lot of attention recently, and CiSE has dedicated two theme issues to it, in January 2009 and in July 2012. Like for software bugs, there are both human and technical reasons, the latter ones being cases of technical debt again.

One major reason for the widespread non-reproducibility of computational re-

sults is the use of immature technology, which I have already discussed above in the context of software development. It means that software needs to be actively maintained in order to be usable in the future, making software maintenance a requirement for reproducibility. Unfortunately, active maintenance of all research software down to the tiniest script used for data munging requires more effort than the scientific community can afford to dedicate to such activities. This is not only a question of affecting the means necessary to do the work. In many cases, only the original author of a script knows what it is supposed to do exactly. If the original author is a PhD student who leaves academic research after the thesis, there is no one left who could do the maintenance. In practice, we most often prefer to default on this kind of debt, all the more since such a default is still socially acceptable today. The Reproducible Research movement works towards paying back the debt in two ways: ensuring the sustainability of widely used pieces of scientific software, and preserving more information about the computational environment of a particular research study, to be published alongside its results as essential documentation.

Another technical reason for non-reproducibility is the sheer amount of information that is required for fully specifying a computation. In theory, any computation is defined by a single computer program. All we have to do is publish that program together with a scientific article, and anyone could re-run it to verify the results. In practice, that program is a complex assembly of a multitude of parts. Typically we have many libraries, and multiple programs that call functions from these libraries. A compiler and linker creates a single unit for each of these programs, which is specialized for a particular type of computer. We then combine several such programs with input data and an outer algorithmic layer often called a "workflow" in order to obtain the result. To make it worse, we often launch computational steps interactively, meaning that a part of the workflow exists only in our heads. Tools for managing the assembly and execution of such complex computations have been around for a long time - the well-known "make" utility for the Unix family of operating systems was published in 1977. They have been ignored by most computational scientists until very recently, partly out of ignorance and partly for not wanting to learn the use of such tools. This debt is in the category of cutting corners for advancing more rapidly. We pay interest in the form of increased manual labor, and we tend to default on the reproducibility aspect.

A final category of technical debt that is frequent in computational science results from an obsession with performance. This debt is particularly difficult to deal with, because the interest can go unnoticed and the debt is almost never paid back. Its importance has nevertheless been recognized, and is well expressed by the famous D.E. Knuth quote reminding us that "premature optimization is the root of all evil (or at least most of it) in programming" [5]. Best practices in software engineering say that one should first write a clear and simple program, and validate it by extensive testing. In a second step, performance bottlenecks

are identified by profiling, and eliminated by optimization. Computational scientists often rush for optimization, choosing low-level programming languages for performance and eliminating error checks perceived as too expensive, before even having a validated program in which they could look systematically for performance bottlenecks. The consequences are a higher software development effort and more mistakes, leading to less reliable scientific results. Both of them could be measured in principle, by comparing different software projects using different approaches, but such an evaluation is expensive and in practice almost never done.

As I already mentioned, the main utility of the technical debt metaphor is to remind scientists, science managers, and funding agencies of the long-term consequences of technical choices. Upon closer inspection, almost every technical choice is associated with some kind of debt, in particular when dealing with cutting-edge technology, which is frequent in research. It is useful to analyze major choices in terms of the debt metaphor: Is the debt perpetual, or will it be paid back? What are the interest payments? Is there a chance that we will have to default on the debt? And if so, will we get away with it? The idea is to turn the tacit contract about technical debt with one's future self into an explicit one.

Any analysis of the technical debt involved in a typical research project will make the importance of infrastructure evident. Infrastructure is everything not specifically made for one research project. In computational science, it includes shared equipment such as supercomputers, but also software made for facilitating research rather than directly for conducting research. This includes systems software (operating systems, compilers, ...), programming languages, scientific libraries, and software development tools. For scientists preparing a research project, all of these items represent debt-laden dependencies. The more stable and predictable the computational infrastructure is, the less risky these dependencies are. This ought to be sufficient motivation for science funders to invest into infrastructure. Fortunately, we see this starting to happen.

Another good investment for the prevention of debt escalation is education and training. As I have shown above, much debt is the result of uninformed choices. In the ideal world, computational scientists would be better prepared to make technical choices, either through better personal education about computing technology, or by close collaboration with experts giving advice. Reading CiSE is of course a good way to improve one's technical competence. We also see grassroots movements such as Software Carpentry (http://software-carpentry.org/) who step in for the academic institutions that have failed so far to integrate computational education into the training of young scientists. With a bit of luck, we may thus be able to avoid a scientific debt crisis.

**Konrad Hinsen** is a researcher at the Centre de Biophysique Moléculaire in Orléans (France) and at the Synchrotron Soleil in Saint Aubin (France). His re-

search interests include protein structure and dynamics and scientific computing. He has a PhD in theoretical physics from RWTH Aachen University (Germany). Contact him at konrad.hinsen@cnrs-orleans.fr.

# References


[1] Matthew Turk
"Vertical Integration"
Computing in Science and Engineering **17(1)**, 64–66 (2015)

[2] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young
"Machine Learning: The High Interest Credit Card of Technical Debt"
SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)
http://research.google.com/pubs/pub43146.html

[3] Zeeya Merali
"Computational science: ...Error"
Nature **467**, 775-777 (2010)

[4] L N Joppa, G McInerny, R Harper, L Salido, K Takeda, K O'Hara, D Gavaghan, S Emmott
"Troubling Trends in Scientific Software Use"
Science **340**, 814-815 (2013)

[5] Donald E. Knuth
"Computer Programming As an Art"
Commun. ACM **17**, 667–673 (1974)