Dealing With Software Collapse

Konrad Hinsen

Dealing with software collapse

Konrad Hinsen

There is a good chance that you have never heard of software collapse before, for the simple reason that it's a term I have made up myself two years ago in a blog post. However, if you have been doing computational science for a few years, there is a good chance that you have experienced software collapse, and probably it was not a pleasant experience. In this article, I will explain what software collapse is, what causes it, and how you can manage the risk of it happening to you.

What I call software *collapse* is more commonly referred to as software *rot*: the fact that software stops working eventually if is not actively maintained. The rot metaphor has a long history, the first documented reference being the 1983 edition of the Hacker's Dictionary [1]. Back then, it was used jokingly by a small community of computer experts who understood the phenomenon perfectly well, and therefore a funny but technically inaccurate metaphor was not a problem. Today, it is being discussed in much wider circles, for example in the context of reproducible research. In my opinion, it is appropriate to introduce a useful metaphor in place of the traditional humorous one, because good metaphors contribute to a better understanding of what's actually going on.

The main issue with the rot metaphor is that it puts the blame on the wrong piece of the puzzle. If software becomes unusable over time, it's not because of any alteration to that software that needs to be reversed. Rather, it's the foundation on which the software has been built, ranging from the actual hardware via the operating system to programming languages and libraries, that has changed so much that the software is no longer compatible with it. Since unstable foundations resemble how a house is destroyed by an earthquake, rather than how spoiling food is transformed by fungi, I consider *collapse* an appropriate metaphor.

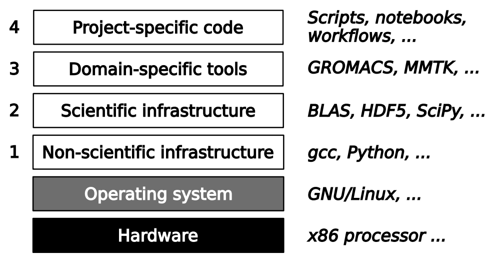

Fig. 1. A typical scientific software stack

THE LAYERS OF THE SOFTWARE STACK

Let's make this more concrete by considering the layers of a typical scientific software stack as shown in Fig. 1. Its ultimate foundation (in black) is the computer hardware that does all the computations. The gray layer right above it, the operating system, performs basic services and isolates the upper layers of the stack from hardware minutiae. So much, in fact, that most software developers consider the operating system, rather than the hardware, the foundation to keep in mind for their artefacts. What computational scientists are most concerned with are the four numbered white layers in Fig. 1, and that is what I will concentrate on. However, it happens occasionally that software collapse is caused by the two foundation layers, as in the case of the Pentium FDIV bug that made a lot of noise in 1994.

Layer 1 contains infrastructure software that is not specific to scientific computing. This includes compilers and interpreters, libraries for data management, but also higher-level tools such as text editors and Web browsers. For the typical computational scientist, this is software that they obtain from the wider non-scientific software market. They can choose which software to use, but they do not have much impact on its development.

Layer 2 is infrastructure created specifically for scientific computing, but not for any particular application domain. Here we find widely used mathematical libraries such as BLAS, LAPACK or SciPy, scientific data management tools such as HDF5, visualization libraries such as VTK, infrastructure for high-performance computing such as MPI, and many more. These software packages are often developed by software professionals as well, but in contrast to layer 1 software, computational scientists are the specific client group the software is written for, which gives them more influence on its evolution.

Layer 3 contains domain-specific research software. These are tools and libraries that implement models and methods which are developed and used by communities ranging in size from a single research lab to thousands of researchers. The examples I have quoted in Fig. 1 are from my own field of research, biomolecular simulation, so unless that is your field as well, you have probably never heard of them. Often the developers are simply a subset of the user community, i.e. scientists who do software development on the side. Larger communities may have research software engineers who have a science background but specialize in software development. In either case, developers work in very close contact with their users, who provide essential feedback not only on the quality of the software, but also on the directions that future developments should take.

Finally, layer 4 contains the software written by scientists for a specific research project. It can take various forms including scripts, notebooks, and workflows, but also special-purpose libraries and utilities. It is becoming more and more common to publish this software in the interest of transparency and reproducibility, but for inspection rather than for reuse by other scientists.

Software in any of these four layers depends on software from the same or lower layers, and is therefore at risk of collapse if one of its direct or indirect dependencies introduces changes that break backwards compatibility. There are three main causes for such changes. What comes to mind first is a development decision to give up backwards compatibility in exchange for the freedom to improve the software's interface. But accidental breakage is probably more common in practice, in particular when interfaces are defined ambiguously and/or insufficiently documented. Bugs can also cause accidental breakage, but since they will usually be fixed upon discovery, they are less of an issue in the long run. Finally, a dependency may simply disappear together with the server that hosted it. That risk is highest for software distributed via a lab's home page, rather than via code hosting sites such as GitHub, but code hosting sites aren't eternal either - anyone remember Google Code?

Strategies for risk management

There are basically four strategies to adopt when facing potential software collapse, and they are quite similar to the strategies that architects can adopt when facing the risk of earthquakes:

1) Avoid risk at all cost. Build on stable foundations only. Don't build a house in a zone with seismic activity, don't accept software dependencies unless they have a very good track record for stability.
2) Make your construction robust. Make your house solid enough to withstand typical seismic activity in your area, with a safety margin.
3) When shaking foundations cause damage, do repair work before collapse happens.
4) Accept that your construction is short-lived. In case of collapse, start from scratch.

For architects, the fourth strategy is unacceptable, and the third one is not that much better because it requires constant attention to prevent a catastrophe. Number one, on the other hand, is very restrictive. It's what you choose for exceptional stability needs, such as for storing nuclear waste, but for most residential and business needs, the second strategy represents a sweet spot between risks and constraints.

For software developers, strategy one is also limiting, but to highly variable degree. It is adopted by necessity when the stakes are particularly high. Banks relying on proven decades-old Cobol programs are a good example. In computational science, a minimal-dependency strategy is usually adopted by choice to escape the collapse-management treadmill when the cost is not too high. As an example, number-crunching on simple data structures can be done in C or Fortran with no other dependencies than the compiler.

When the complexity of data structures or algorithms is higher, a reasonable development productivity requires building on a wider range of infrastructure software, and thus dependencies. This is particularly true for visualization or graphical user interfaces that require extensive interfacing with the operating system. But perhaps the most frequent motivation for adding dependencies in layer-3 software is the desire to write less code oneself, avoiding both vertical integration [2] and reinventing the wheel. The risk of collapse incurred is then a classical case of technical debt [3]. For layer-4 software, this almost becomes an imperative as it is rarely possible to develop dependency-free software for a single project within the typical constraints on time and resources.

Strategy two, the sweet spot for buildings, is not really applicable to software in the current state of the art. The reason is that we do not know how to make software robust against small incompatible changes in their dependencies. In the physical world, the importance of a change can be quantified, the impact of quantified changes can be estimated, and sufficient safety margins can then be found. In the world of bits, there is no such thing as a small change. A library API either is or is not a subset of another library API. One can imagine more tolerant APIs that include, for example, a negotiation phase between library and client, but I am not aware of any developments in this direction.

The third strategy is the most popular one for scientific software development in layers 2 and 3. A team adapts its code to breaking changes in its dependencies as part of an ongoing development process that also includes fixing bugs and implementing new functionality. This tends to work rather well as long as adaptation and bug fixing represent a minor effort compared to new developments. In that scenario, volunteer developers are attracted to the project by the interesting work on new functionality, and accept the drudgery of maintenance as the price to pay. When maintenance takes a larger share of the developers' attention, this strategy can still be made to work by actually paying people to do the job, but funding for software maintenance remains very difficult to find.

The fourth strategy is acceptable only for software that has no long-term importance. The only category I can think of is research prototypes, i.e. software that is developed in order to gain a better understanding of a problem, but not to be used as a tool. In today's practice, the first part of this strategy, "accept that your construction is short-lived", is also adopted by the majority of layer-4 software, not because it is a good choice but because it is the only possible one in a context of limited resources. The "starting from scratch" part never happens. This is one of the causes for the all to common non-reproducibility of computational results.

Evaluating the risk of collapse

Let's move on to more practical considerations. You are developing scientific software that has been or will be published. How do you deal with the possibility of one of your dependencies causing your work to collapse? And what are you going to do to prevent a problem with your product to cause the collapse of someone else's software that depends on yours?

As a first step, consider the time scale of change in your own project. Do you develop software that implements well-known and trusted methods for use by a large number of researchers? In that case, your software will evolve very slowly, fulfilling the same role for decades. At the other extreme, if your software is developed as part of research in a fast-moving field like machine learning or bioinformatics, it will evolve rapidly, and last year's release may be of interest only for the history of science. As a rule of thumb, the time scale of layer-4 software is the duration of the project it serves plus the length of time you expect your computations to remain reproducible. For layer-3 software, it's the time scale of methodological advance in its research domain that matters. Check for example How old are the methodological papers that you tend to cite. Infrastructure software, i.e. layers 1 and 2, can fulfill its role only if it is more conservative than anything that depends on it, so its time scale of change is defined by its intended application domains.

Next, you must estimate the time scale of change of your dependencies. For layer-3 dependencies, that should be rather straightforward, as they are likely to evolve in the same research community as yourself, and thus evolve on similar time scales as your own work. For infrastructure software, the task is more difficult. The fact that you are considering to adopt package X as a dependency does not mean that the developers of X have your needs in mind. So you have to look at the past evolution of X, and perhaps at the time scales of the major clients of X, to get an idea of what to expect for the future. For young projects, there isn't much past to study, so you should estimate their time scale by their age.

Once you have all these time scale estimates, you can identify the most risky dependencies: those whose time scales of change are faster than your own. If you go for strategy number 3, i.e. adapting your code rapidly to changes in the dependencies, then you might have to invest a lot of effort into catching up with those fast-moving projects.

Of course, change doesn't mean breaking change. Your dependencies may well evolve at a rapid pace by growing to implement more functionality, while being careful not to break backward compatibility. It's therefore also important to know the projects' policies, and the means at their disposal to actually implement them. But even if a project's policy assigns a high priority to preserve backward compatibility, a fast pace of change is still a warning sign because all change entails a risk of making your software collapse by accident. Again, the best estimate is obtained by looking at the project's track record.

Speaking of policies, you should also think about your own, and ideally write it down clearly as part of your documentation. You can make your clients' life even easier by adding your estimate of your project's time scale of change.

SOFTWARE COLLAPSE IN REAL LIFE

I will conclude this article with the story that started me thinking about software collapse, which is the story of my first personal encounter with the phenomenon. In 2007, I started a joint project with experimental biologists whose goal was to find the atomic-scale structure of a protein from low-resolution images combined with biochemical evidence for the proximity of certain pairs of amino acid residues. The result of this work was published in 2014, meaning that the project took seven years to completion. That's not untypical for projects that involve a back-and-forth interaction between experiments and computations.

At the start of that project, my software environment for molecular simulation was built around my very own Molecular Modelling Toolkit (layer 3), a Python library I had been developing since 1997 with Python (layer 1), Numerical Python (layer 2), and netCDF (layer 2) as its main dependencies. The major additional dependency for my project-specific layer-4 work was matplotlib (layer 2) for plotting.

I had started using Python in 1993 with version 1.3. In 2007, the current version was 2.5, but in spite of a significant evolution of the language during those 14 years, I only had to adapt my code once to change the name of a variable that had become a new keyword. Numerical Python had been around for ten years with continuous improvements, but no breaking changes and few major bugs. It had just been superseded by NumPy a year before, and everyone, including myself, was at some stage of migration, but since NumPy had a compatibility module for Numeric, that wasn't much of an issue. My own MMTK had evolved enormously but without ever introducing breaking changes. My very first simulation scripts from 1997 still worked fine. netCDF had an excellent track record of stability as well. In short, I had no reason to expect any trouble.

But then, over the seven years of the project, I ended up spending a lot of time catching up with dependencies. New releases of NumPy and matplotlib made my code collapse, and the increasing complexity of Python installations added another dose of instability. When I got a new computer in 2013 and installed the then-current versions of everything, some of my scripts no longer worked and, worse, one of them produced different results. Since then, software collapse has become an increasingly serious issue for my work. NumPy 1.9 caused the collapse of my Molecular Modelling Toolkit, and it seems hardly worth doing much about it because the upcoming end of support for Python 2 in 2020 will be the final death blow., since porting the code to Python 3 would almost be a complete rewrite.

But if I were to attempt a rewrite, would I still choose the Python ecosystem, whose time scale of change has become too fast for the kind of work that I do? I have seen some of my colleagues move back to C or Fortran in a quest for stability. Should I do the same? But then, working with complex molecules in these languages is no fun, and I'd miss the convenience of all those nice tools and libraries that exist in the Python universe. It's not an easy choice. The only conclusion I have drawn so far is to make my software much more modular, to prevent collapse from affecting all of it. Today I would definitely no longer start a "molecular modelling toolkit" project combining lots of distinct functionality in one package.

An interesting question is how it could happen that the scientific Python ecosystem, which had been remarkably stable for its first ten years, then turned into a fast-moving target. I suspect that one cause is the enormous growth of its user base associated with a shift in application domains and thus time scales. Python has been widely adopted in new fields of research, such as machine learning, that hardly existed when Numerical Python was first released in 1995. Today's contributors and maintainers of the scientific Python infrastructure come from backgrounds with a much faster time scale of change. For them, NumPy is probably a sufficiently stable infrastructure, whereas for me it isn't. The second cause I see is Python 3. The cleanup of the language at the cost of breaking backward compatibility has sent the message to the community that this is the Right Way, or at least a good way, to manage a long-running software project.

The lesson to be learned from this is that second-order effects can be important: the time scales of change are themselves subject to change. Another second-order risk factor is the uncertain survival probability of a project. Funding for scientific software development is still very difficult to obtain, and therefore projects flourishing today could well be starving in a few years. The only reasonable protection against this is software foundations that have multiple implementations, ideally based on a common written standard. C, Fortran, BLAS, and Unix are likely to be around in 20 years even if the development teams or companies backing them today disappear for whatever reason. They are the software equivalent of institutions that are too big to fail because too much depends on them.

References

[1] Guy L. Steele Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman and Geoffrey S. Goodfellow, *The Hacker's Dictionary: A Guide to the World of Computer Wizards* (1983). http://jargon-file.org/archive/jargon-1.5.0.dos.txt

[2] M. Turk, *Vertical Integration*, Computing in Science & Engineering **17**, 64 (2015)

[3] K. Hinsen, *Technical Debt in Computational Science*, Computing in Science & Engineering **17**, 103 (2015)

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans and at the Synchrotron SOLEIL in Saint Aubin. His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University. Contact him at konrad.hinsen@cnrs.fr.