Staged computation: the technique you didn't know you were using

Konrad Hinsen

Staged computation: the technique you didn't know you were using

Konrad Hinsen

A quick Web search for "staged computation" will convince you that it refers to an exotic technique of interest mainly to programming language designers and implementers. Nothing could be further from the truth. It's a technique that everybody is using all the time, and it's one of the main reasons why reproducible results are so difficult to achieve. In other words, it's something that every computational scientist should know about.

By definition, a staged computation is a computation that proceeds as a sequence of multiple stages, in which each stage produces the *code* for the next stage, except for the last stage that produces the final result. Most of the references you will find in a Web search are about so-called meta-programming techniques, in which a program's source code is transformed before actually being compiled into an executable. The most widely used meta-programming technique in scientific computing is the use of macros in the C and C++ languages. These macros are rules for rewriting the source code before compilation that are used for specializing the code, or for adapting it different platforms. Other languages have more elaborate metaprogramming tools, in particular the languages of the Lisp family.

However, the kind of staged computation that I will discuss here is something quite different: it is the very use of compilers. A compiler transforms a program from one notation ("source code") to another notation ("executable binary"). Executing a piece of source code thus requires two stages: the first stage, compilation, produces the code that is run in the second stage, execution. But that is not the end of the story. The compiler you run is typically an executable binary stored somewhere in your computer's file system, and so are the libraries that a program makes use of. These binaries have been produced by someone else, on another computer, using yet another compiler. So our two-stage computation is really a many-stage computation, with the results of the initial stages stored on some server for downloading by people like you. Package managers, such as `apt` used by Linux distributions such as Debian or Ubuntu, or `Homebrew` for macOS, make this approach straightforward in practice.

1 MULTI-STAGE REPRODUCIBILITY

Fig. 1 provides a visual illustration of a staged computation. Each box in this diagram corresponds to data stored in a file, but there are three distinct categories of data. The first category, shown in blue, is input authored by humans, i.e. mostly program source code. The second category, shown in magenta, is observational input, typically coming from experimental equipment. The third and dominant category, shown in green, is results of computations. The computations that produce them are indicated by arrows that link inputs to outputs.

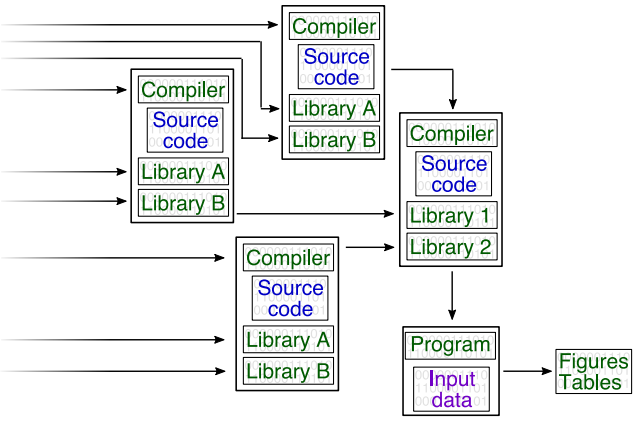

Fig. 1. The final stages of a typical staged computation. Items shown in blue are human input. Items shown in magenta are other input, typically from experiments. Items shown in green are the output of a computation.

What we would like to be reproducible is the box in the bottom right, i.e. the figures and tables we put into our publications. The important message of Fig. 1 is that *reproducing these results requires all the other items in the diagram to be precisely identified and either archived or themselves reproducible.* And since even a modest computation can accumulate hundreds of little boxes, this is not a trivial requirement.

Let's look at what this means for our three categories of items. Human input is the easiest case: it cannot be reproduced, so it must be archived. Note that "archiving" means more than just storing a copy in a safe place. That would be a backup, not an archival copy. Archiving requires producing a safely guarded copy plus a handle via which this copy can be retrieved unambiguously. That handle could be a file name or a URL (both very fragile), a Digital Object Identifier (DOI), which is already more robust, or ideally a handle computed from the content itself [1], such as the identifiers used by Software Heritage [2]. Since human

input tends to evolve in the course of a research project, it is also advisable to keep it under version control [3]. Both Zenodo (https://zenodo.org/) and Software Heritage (https://www.softwareheritage.org/) provide facilities for easily archiving version-controlled human input.

It is more difficult to give general recommendations for observational input, because that is a very diverse category. Like human-authored input, observational input is not reproducible, and must therefore be archived. For choosing an adequate archiving technique and platform, the nature and size of the data matters, but also legal criteria such as ownership or privacy.

For the computed results, in particular compiled software, we have the choice between archiving and reproducing. Unfortunately, the tools we have been using for decades to manage software support neither option satisfactorily. They have been designed for installing, updating, and deploying software, but not for tracking provenance or reproducing earlier states. These tasks must therefore be assumed by humans, who are not very good at managing hundreds of items. This is why reproducible computations remain such a tough challenge. The good news is that computers are very good at dealing with large problems, meaning that we can delegate the management of staged computations to software tools, as I will show later.

2 THE STATE OF THE ART

Consider the very simple program shown in Fig. 2, which does a few computations and prints the results. On a typical Linux system, you would run it using

```
gcc pi.c -o pi
./pi
```

assuming you are using the popular GNU Compiler Collection. The few lines of text printed by the program are the final result. That's what goes into the bottom-right box in Fig. 1. The executable binary `pi` is the "Program" in the box on the left end of the arrow. There is no "Input data" in this case. The source code file `pi.c` is the "Source code" in the box above. The "Compiler" is `gcc`. And it looks like we have no libraries, so we have properly identified everything in the rightmost three boxes of Fig. 1. That's a good start!

The bad news is that appearances are deceptive: we do have libraries, the compiler is merely hiding them from us. Under the hood, the compiler runs additional programs such as `as` or `collect2`, and adds libraries from the C language runtime system. We need to add this hidden stuff, with version numbers, to the "Compiler" and "Library" fields of the box. And with compilers playing hide-and-seek, we need a more reliable way to figure out all of our dependencies!

I suspect that many readers think that I am exaggerating. We are talking about a minuscule C program. All it takes to run it is a toolchain for compiling C programs. It doesn't matter if I use version 7 or version 9 of `gcc`. It matters even less what the version number of `collect2` is, whatever it may do. And all the stages before the compilation of `pi.c` shouldn't matter at all. If the `gcc` that I am running has passed its test suite, all should be fine.

This reasoning is, in fact, correct most of the time, when applied to C compilers and other stable tools and libraries. But most of the time is not all of the time. One particular subtle point is floating-point arithmetic, which has the reputation of being fundamentally irreproducible [4]. And yet, at the level of the operations defined by the standard IEEE-754 (which all processors and compilers today respect), floating-point arithmetic is perfectly deterministic. The problem is that programmers don't write their code in terms of IEEE-754 operations. The C language doesn't even give access to that level. It's the compiler that generates those low-level instructions, and it assumes that the programmer doesn't care about the differences due to round-off. So if you want reproducible floating-point results, their full specification is your code plus the C compiler plus the compilation options.

More importantly, the "details don't matter" reasoning fails for large software assemblies, in which pieces you have never heard about but which have an impact on the final results may change because of bugs or voluntary decisions to break backwards compatibility. Under the Guix package manager, about which I will say more below, running the C program in Fig. 2 requires four packages (gcc, binutils, glibc, ld-wrapper, the last one being a Guix-specific package). That's what takes the green-colored slots in the rightmost "Compiler" box in Fig. 1. For a Python script using the popular NumPy and SciPy extensions, that's already 24 packages. If you include all the packages from the earlier stages of the computation, you get 89 packages for a C program, but 501 for Python + NumPy + SciPy. Figuring out which of these packages should be irrelevant details becomes a serious challenge.

Let's move on for now, assuming that we can somehow keep track of hundreds of dependencies. We must then either archive the compiled code, or make it reproducible. But we can immediately eliminate "archiving" because there is no practically usable infrastructure for this. Sure, package managers download compiled code from servers, but these servers are *caches*, designed for increasing the efficiency of software distribution. They are not archives from which binaries could be retrieved at arbitrary later times via an unambiguous handle. That also applies to services such as DockerHub that hold container images.

The situation looks more promising for making all compiled code reproducible. Package managers are based on build recipes, which is the code that is run to re-build the package. Likewise, containers are usually built from such recipes, e.g. the well-known Dockerfiles. Unfortunately, a closer look reveals that most of these build recipes are not reproducible. They say something like "download the current Python source code and compile it with the current version of gcc." That's great for regularly updating software, which is after all what package and container managers were designed for, but it's not reproducible.

3 GUIX TO THE RESCUE

The Guix package manager (http://guix.gnu.org/) for GNU/Linux was designed from the start with reproducibility in mind, and offers the best support for reproducible research that is currently available. More specifically, it is based on two fundamental concepts:

```c
#include <math.h>
#include <stdio.h>

int main()
{
    printf( "M_PI                          : %.10lf\n", M_PI);
    printf( "4 * atan(1.)                  : %.10lf\n", 4.*atan(1.));
    printf( "Leibniz' formula (four terms): %.10lf\n", 4.*(1.-1./3.+1./5.-1./7.));
    return 0;
}
```

Fig. 2. A very simple computation in C.

- An explicit representation of the full staged computation graph of Fig. 1, for all software packages, containing all information that can potentially impact results, and referring to specific versions of all source code.
- Execution in restricted environments. Guix can run programs in environments where only explicitly listed software is available, providing a guarantee that the programs have no other dependencies. These restricted environments are used for building software packages, but are also available to users for running their own code.

To run the example from Fig. 2 in a restricted environment in Guix, I would type

```
$ guix environment --container \
                   --ad-hoc gcc-toolchain
[env]$ gcc pi.c -o pi
[env]$ ./pi
```

The first line creates a restricted environment with access to a single software package (`gcc-toolchain`), a single directory (the current one), and no network access at all. It then starts a shell in that environment, into which the following two lines are typed. The option `--container` provides the strongest possible isolation of that environment, but less restrictive versions are also available. The fact that my program works correctly in that environment proves that it has no dependencies other than `gcc-toolchain`, which is a package specifically designed for C programming and contains the `gcc` compiler plus the utilities and libraries it requires to function. To make computations in such environments reproducible, all I need to do is note the exact version of Guix that I am using:

```
$ guix describe -f recutils
name: guix
url: https://git.savannah.gnu.org/git/guix.git
commit: 769b96b62e8c09b078f73adc09fb860505920f8f
```

The Guix version is given by the `commit` field, which is a unique and persistent handle. I can then at any time in the future reproduce the environment, and thus my computation:

```
$ guix time-machine \
       --commit=769b96b62e8c09b078f73adc \
       -- \
       environment --container \
                   --ad-hoc gcc-toolchain
[env]$ gcc pi.c -o pi
[env]$ ./pi
```

If you think that the first line is rather long, consider that it replaces a whole Docker container!

Like other package managers, Guix downloads precompiled binary versions of its packages from a caching server if available. Otherwise, or upon explicit user request, Guix recompiles everything from source code. Well, almost everything: a minimal archived core package called the bootstrap seed is always downloaded in binary form. It contains a basic compiler that is used to get the staged computations started. You can't avoid having to download that bootstrap seed in binary form, but if you are particularly paranoid, you can then recompile it yourself, and verify that you get the same files, bit for bit.

Since Guix stores the complete staged computation graph, you can also explore it using Guix' command line tools for standard tasks such as querying the version numbers of the packages. For more advanced needs, you can write scripts in Guile, the dialect of Scheme that Guix is written in. This is what I did to obtain the dependency counts that I quoted earlier. For readers interested in the technical details, there is a post on the Guix blog [6].

At this time, Guix is still a tool for early adopters. Its package collection is not nearly as complete as those of well established distributions, and its tools still evolve rather rapidly. However, it shows that reproducible staged computations are possible, and it is actually already very usable in practice if all the software you need is in there. Check it out for yourself!

REFERENCES

[1] K. Hinsen, *The magic of content-addressable storage*, Computing in Science & Engineering, early access, DOI: 10.1109/MCSE.2019.2949441

[2] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli, *Identifiers for Digital Objects: The Case of Software Source Code Preservation*, iPRES 2018 - 15th International Conference on Digital Preservation, Sep 2018, Boston, United States. DOI:10.17605/OSF.IO/KDE56

[3] K. Hinsen, K. Läufer, G.K. Thiruvathukal, *Essential Tools: Version Control Systems*, Computing in Science & Engineering **11**, 81–91 (2009)

[4] K. Hinsen, *The approximation tower in computational science: why testing scientific software is difficult*, Computing in Science & Engineering **17**, 72 (2015)

[5] K. Thompson, *Reflections on trusting trust*, Communications of the ACM **27**, 761 (1984)

[6] K. Hinsen, *Reproducible computations with Guix*, https://guix.gnu.org/blog/2020/reproducible-computations-with-guix/, published January 14, 2020

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.