A Dream of Simplicity: Scientific Computing on Turing Machines

Konrad Hinsen

Scientific Programming
Editors: Konrad Hinsen, konrad.hinsen@cnrs.fr
Matthew Turk, matthewturk@gmail.com

A Dream of Simplicity: Scientific Computing on Turing Machines

Konrad Hinsen
Centre de Biophysique Moléculaire in Orléans

Frustrated by another failed software installation? Wondering why you can't reproduce your colleagues' computations? This story will tell you why. It won't magically solve your problems, but it does point out a glimpse of hope for the future.

It's one of those days in the life of a computational scientist. Your desktop computer is busy updating its operating system and doesn't have any time left for you. The installation of your favorite simulation software on the new supercomputer doesn't advance beyond an error message extending over 500 lines. And your data analysis hasn't made much progress either: the results change when you use more or fewer processors, which doesn't look right. You already asked Siri for advice, but as so often is the case, that conversation led absolutely nowhere. Computers are complex, frustrating machines. If only we could have simple ones that just did what we tell them to do!

Maybe it's a good idea to go back to the basics. Ever heard of Turing machines? They were invented in 1937 by Alan Turing,[1] and they're as simple as computers can get. No keyboard, no screen, no hard disk—no operating system either, nor any voices pestering you. You write your instructions on a tape, put the tape into the machine, and push a button. The machine starts working, moving the tape back and forth. When it stops, you take out the tape again and read off the result. Computing couldn't be simpler!

Turing invented the machine that was later named after him as a purely theoretical concept. There wasn't much real computing hardware available back then, but people eventually built physical realizations (www.legoturingmachine.org/lego-turing-machine, http://aturingmachine.com, and http://videotheque.cnrs.fr/index.php?urlaction=doc&id_doc=3001). However, before you rush off to Amazon to shop for your own Turing machine, watch out: not all Turing machines are equal. Each of them has its own instruction set to define its capabilities. What Turing wrote about in his famous paper is the family of all imaginable Turing machines, which only shows that he was a theoretician. Fortunately, there's a subfamily called *universal* Turing machines. They aren't all equal either, but they all have the same capabilities. In fact, they can compute anything that any computer can possibly compute, so make sure you buy a universal model, even if it's a bit more expensive.

Another detail you should watch out for is the machine's symbol set. Those symbols are what you're allowed to use on the tape and what the machine can use to produce its output. The cheap models work in binary, allowing only the symbols 0 and 1 on the tape; high-end Turing machines can handle Unicode symbols. Not only does this allow you to copy your favorite poems to tape, no matter which language they're written in, but it also saves you a lot of typing.

When you order your Turing machine, be sure to order a sufficient supply of blank tape to get you started. Turing speaks of infinitely long tapes in his paper, but that's a theoretician for you. The good news is that a computation that's of any practical use will need only a finite piece of tape. Exactly how much is difficult to predict, unfortunately, but for your first experiments, a 100-meter roll should be sufficient.

For the details about operating your machine, refer to its instruction manual. In particular, the manual should tell you how to prepare your input tape, either via a special keyboard device or a plain old typewriter from the attic. What's more important is the machine's instruction set, which is explained in detail in the manual as well. The effect of each instruction tends to be very basic: move the tape one step to the right, copy the symbol at the current position to internal register number 3, and so on. It might seem laborious to express your problem in terms of such low-level operations, but that's the price you pay for simplicity. You chose asceticism to escape the frustration of complex machinery. Would you really be happier reading the 500+ pages that describe the instruction set of an ordinary modern computer?[2]

You'll definitely appreciate a Turing machine's simplicity when it comes to publishing your work. Being a *CiSE* reader, you're, of course, practicing reproducible research—a Turing machine makes it even

easier. Just publish a copy of your input tape as supplementary material to your article and describe the precise model of Turing machine that you used. If you worry that its manufacturer might go out of business, making it impossible for your readers to get an identical model for replication purposes, add the specification of the machine as an appendix to your article as well (the specification is the part of the manual that describes the instruction set, and with that information, any competent Turing machine shop can build a compatible model, at any time).

Frequently Asked Questions, Beginner Level

Q: My machine doesn't stop, it just keeps on running. What should I do?

A: First, it's possible that your computation contains an infinite loop, which means it's erroneous and you should correct it. Otherwise, the problem you want to solve is probably bigger than your level of patience. Buy a faster machine or start practicing yoga.

Q: How do I know whether my computation is just long or contains an infinite loop?

A: Mathematics can't help you, sorry[1]. Maybe your yoga teacher can offer advice.

Q: What I write on the tape is a program, but how do I get my data into the machine?

A: I don't agree. I'd say the program is embodied by the machine, and the tape contains only data. In fact, the distinction between program and data is part of our *interpretation* of a computation. From the machine's point of view, it just manipulates a sequence of symbols. You need to develop a more holistic view of computation.

Q: It's a real pain to write floating-point operations in terms of single-symbol manipulations all the time. Isn't there an easier way to do this?

A: You could try contacting a renowned Turing machine manufacturer and asking for a machine designed specifically for number crunching, with predefined floating-point operations in its instruction set. Let me know if you succeed! In the meantime, read on for other ideas.

Q: I do many computations that share common elements, in particular, mathematical functions. Is there a way to write them only once?

A: This is a more general version of the last question, since floating-point operations are nothing but specific common elements. The answer is the same as well. Convince a manufacturer that it should design them into its machines... or read on for other ideas.

Q: I need to run the same computation on many different input datasets. Every time I have to design an input tape from scratch. Isn't there an easier way to do this?

A: Let me rephrase your question: you have many computations that share a common element, which is the algorithm that you wish to apply to your different datasets. So this is really the same question as the previous one. If your algorithm is important and you plan to use it intensively for many years, consider having a Turing machine designed and built specifically for it. Your tapes then contain nothing but your datasets. If a custom-built machine sounds impractical, read on for other ideas.

Building Blocks to the Rescue

If you've started playing around with a Turing machine and perhaps encountered some of the beginner-level questions highlighted in the sidebar, I suspect that some of you are considering a remorseful return to your desktop computers. Well, simplicity isn't for cowards. I never said it would be *easy*, but stay with me for a while, and maybe we'll find a way out of this situation.

What's probably bothering you is having to figure out how to do what you consider basic operations, such as adding two floating-point numbers, in terms of even more basic symbol manipulations. Worse, you have to do it all over again, for every single addition! And what happens when you lose track of which part of your very long tape does what?

As I mentioned, one solution is to convince a Turing machine manufacturer to produce a machine that's better suited to your needs. Unfortunately, this can rapidly become rather expensive, plus it has an additional disadvantage: every new instruction in your Turing machine makes its specification more complex. If your manufacturer disappears, the next one will have to work much harder to make a

compatible machine in the future. That's not only going to cost a lot of money, but a complex specification also entails a higher risk of mistakes in the design of a new machine. And even if those arguments don't convince *you*, they might convince the editorial boards of your favorite journals, who worry about the reproducibility of your work. They might actually reject your next paper for relying on an obscure and expensive custom-built machine rather than on cheap and well-understood commodity equipment.

Here's another idea. Imagine that you could write symbol sequences for common simple jobs, such as adding two floating-point numbers, and then just copy and paste these sequences into your tape where needed. Rather than compose your tape symbol by symbol, you could compose it from bigger building blocks. Or, better yet, make a collection of such reusable tape segments, paste the whole collection into your tape, and then somehow refer to its items when designing your computation. We could call such pieces subroutines and our collection a subroutine library.

We have a few administrative details to consider to make this work. To tell our machine to run a subroutine, we must tell it precisely which one. We could give each subroutine a name, and say "run subroutine FLOAT-ADD," or we could use a numbering scheme, saying, "run the third subroutine in the library," but neither of these approaches is perfect. Imagine that you delegate writing subroutines to your graduate students and then combine their efforts into a single library that everybody can be proud of. What if two of your students have used the name "FLOAT-ADD" for somewhat different subroutines? What if all of them have referred to their subroutine number 2 somewhere in their code, but once the library is put together, the numbers have all shifted? In fact, you don't just want to compose subroutines into libraries, you also want to compose libraries into bigger libraries. Otherwise, what's the point of having graduate students? So it's worth thinking harder about this problem.

We also need to figure out a way to tell our machine to run a subroutine and then come back to the "main" program—as well as a way to store the data to be processed by our subroutines in a place where the subroutine's instructions can pick them up. How to do this depends on your machine's instruction set, so you have to figure it out for yourself. Definitely do *not* delegate this to your students: each of them will come up with a different scheme, and they'll quarrel forever about whose scheme is best. Have you seen the endless online discussions about the relative merits of different programming languages? You don't want that happening in your research group. Setting up conventions requires firm authority. It's a job for a real expert, like you.

Whatever scheme you come up with, it will consist of conventions to be followed by both authors and users of subroutine libraries and support code to make the scheme work on your Turing machine. Let's call these conventions an API, which will impress your friends in software engineering. While we're at it, let's call the support code an operating system. When everything ends up working, you can proudly say that you've written an operating system that implements a powerful API. The mundane work of using it to prepare useful computations can then be left to your students. It's much more interesting to be the author of an operating system—after all, everyone who writes or uses a subroutine will have to refer to the API and the operating system. You'll get lots of citations!

Data Management

I hope that by now you're feeling a bit more optimistic about your future with the Turing machine. You can compose large programs using references to reusable subroutines. You can also compose these subroutines into libraries. And if you've figured out a clever subroutine reference scheme, in which one library's references don't interfere with those of another library, you can even compose libraries into bigger libraries.

But remember that a program is really just another reusable building block for computations: you still need to compose it with an input dataset to make a full computation that you can set your Turing machine to work on. Or maybe several input datasets. Come to think of it, you'll want datasets to be building blocks as well, which means you need a way to refer to data items in your computation. How about names or a numbering scheme? If you want to impress your friends in software engineering, maybe you should call a data item a file and a collection of data items a filesystem. You prefer to call it a data item library? That's fine with me, but be warned that if you don't put up a smokescreen of jargon, people will belittle your work as overly simple!

The management of data access requires some support code as well, but just a couple of subroutines.

This is a good occasion to complement your operating system with its very own subroutine library. You'll need a great idea for a composable subroutine reference schema, otherwise your operating system's library will occupy names such as "READ-DATASET" or particularly convenient numbers such as 1, meaning that nobody else's subroutines can use them. If you didn't yet come up with a great idea, don't worry: as the operating system supplier, you're the boss, so everyone else will have to comply with your rules!

There's a lot we could do to make programming our little Turing machine even more pleasant, but let's stop here and profit a bit from our investments. Finish your subroutine library and use it to get some research done. When you publish your work, don't forget that for reproducibility you still have to publish your complete input tape, including the operating system and your subroutine library. And because your tape is probably a bit long by now, be nice to your readers and write some documentation so they can figure out how everything works.

Opening Up to the World

Congratulations on your first publication based on Turing machine computations! I'm sure it'll become a citation classic. In the meantime, as you might have expected, other *CiSE* readers have followed the same path you did, and their papers will be out at about the same time as yours. Like you, they're attracted by simplicity, so wouldn't those people be ideal partners for future collaborations? How about organizing a conference to bring them all together?

Unfortunately, you'll likely have a few technical problems when you start collaborating. First, not everybody bought the same model of Turing machine. Second, each group has its own operating system with its distinct API and standard library. Even data is organized differently on each group's tapes. Before you can start sharing the fruits of your hard work, you have to think about how to put it all together.

Believe it or not, Turing has once more done the hard work for you.[1] He showed that every universal Turing machine can run software for any other Turing machine by simply prefixing that machine's tape with a specific symbol sequence that acts as an adapter. Call it a virtual Turing machine if you want. Producing such an adapter should be no more than an exercise for you now that you're an experienced Turing machine programmer. And if you can virtualize your colleagues' entire Turing machines, virtualizing their operating systems and APIs is just a little extra work. Not only can you run everyone's tapes, you can even compose a new computation from subroutines written by different people for different instruction sets and APIs. Isn't that cool?

Frequently Asked Questions, Expert Level

Q: I studied one of my tapes over the weekend and noticed that only 10 percent of the symbols were about my science—the other 90 percent were operating system, adapters, and subroutines for data access. Isn't that a lot of overhead?

A: Perhaps, but would you rather go back to composing your tapes symbol by symbol as you did in the dark ages?

Q: My graduate student spent an entire week debugging the tape from our last collaborative endeavor. It turned out that the adapter for our colleague's data access subroutines was incorrectly using another colleague's operating system. In fact, on a more careful reading, that operating system's API specification turned out to be incomplete. Do you have any advice for avoiding such issues in the future?

A: Maybe error checking would help? All it requires is some additional support code and conventions.

Q: I have this strange feeling that if we go on improving our tape development environments, we'll get into the same mess that we have on our desktop computers. Is that right?

A: Well...you might be on to something there.

Back to the Real World

You've certainly noticed some similarities between my fictitious universe of Turing machines and the real world of computers made from silicon, but did you notice when and why the two worlds started to look

similar? At what point in the story did the Turing machines' simplicity start to evaporate? It's when we wanted to prepare a computation by composing preexisting elements designed for reuse, right? In other words, modular design makes complex computations feasible in practice, but at the same time, that's what makes our daily lives so frustrating. In the end, computers aren't really the origin of these frustrations—rather, it's the limitation of human brains that can't reason reliably about sequences of thousands of instructions. Abstraction and modularization were invented for the programmer's benefit, not the computer's.

In real life, though, a major issue is the multitude of schemes for composing units of computation. A typical programming language provides building blocks called procedure, function, and subroutine. But each language provides its own variety of this basic idea, with somewhat different conventions and different support code to make it work. Consequently, it's usually difficult and sometimes impossible to combine procedures, functions, and subroutines written in different programming languages. Moreover, languages, operating systems, and web services provide several levels of building blocks whose composition mechanisms are always similar but rarely equal to the point of being interoperable. Popular names for such building blocks are "class," "module," "package," "library," "script," "executable," "file," "directory," and "filesystem," but no two uses of these terms refer to exactly the same concept. A big part of the software agglomerate that occupies a modern computer's memory is therefore made up of glue and translation code, much like in my Turing machine story. This glue and translation code includes linkers, loaders, archive managers, configuration tools, package managers, file format converters, and a lot more, all of which solve problems that we wouldn't have without computers. More importantly, all this code creates opportunities for mistakes to sneak in.

As I already hinted, one cause for this diversity is that it's hard to make people agree on something, especially if everyone has already made investments in a particular technological choice. In the realm of physical objects, economic pressures related to economies of scale provide a strong incentive for agreement: every manufacturer can define its own sizes of screws and bolts, but people can also realize that they're better off if the whole industry agrees on a few standard sizes such that a few manufacturers produce screws and bolts for everyone at a low price. In principle, this mechanism ought to work for software as well, but the cost of deviating from a standard is much lower in the world of bits than in the world of atoms. Perhaps more importantly, computing is still in an early development phase, with several big players hoping to gain a competitive advantage by imposing their technology on everyone else.

But there's also a productive source of diversity that comes from different and evolving technological environments. One convention for composing components might favor performance, another one memory usage, and yet another one flexibility. New applications lead to new requirements that could conflict with an earlier convention's design goals. New insights permit improvements that can only be implemented by abandoning old conventions. Computing technology is young and immature, with a fast rate of technological progress creating incentives for change at all levels.

To convince you that composition really is at the heart of most of our frustrations with computers, I'll discuss one familiar problem with two different labels in more detail: software installation and nonreproducible computations. They're actually the same fundamental problem because non-reproducibility means that you can't successfully install somebody else's computation on your computer because you don't have all the required parts, they can't be composed with the software that's already available on your computer, or you don't have sufficient instructions to perform the composition.

The way software installation is handled on most of today's computational platforms is by continuous aggregation. The starting point is the hardware plus minimal firmware (called a BIOS). The first add-on is an operating system, such as GNU/Linux or Windows. Such an operating system is already a composition of many parts, including a kernel, system libraries, and utility programs; software installation simply adds to this collection. System libraries are complemented by other libraries, and system utility programs gain application programs as new neighbors. There is, in fact, no fundamental difference between the system parts and the other parts—the label "system" simply means that an item is provided by the operating system manufacturer.

At each step of a software installation procedure, you thus compose an aggregate of items with an additional item to make a more powerful aggregate. The new items—libraries and applications—must be compatible with the aggregate you start with. But that aggregate is nearly unique; it's the preexisting installation on your computer, the result of a particular sequence of software installation steps that's unlike anybody else's. The developers of the additional items you want to add can't possibly have tested their code on the aggregate that you'll install it on, which is a big reason why software installation so often ends

in frustration. It's also a reason why computations are so difficult to reproduce: in addition to the software and data that your colleagues ideally provide as supplementary material for their papers, you would need the exact same software aggregate that they used. However, describing one's software aggregate in a paper is difficult, lengthy, and ultimately of limited use because there's no straightforward way to reproduce it elsewhere.

You might argue that this is no different from other technologies: your car is quite different from your friend's car, yet you don't have to include a detailed description of your car down to the last screw when you explain to your friend how to drive from point A to point B. But computation is unlikely any other technology in its dependence on small details.[3] Most technologies use interfaces that are robust against small changes in their implementations, but we don't yet have such interfaces in computing technology.

The difficulties of software installation have led to the development of a category of support tools called package managers, of which the best-known examples are Debian's apt and RedHat's rpm. A package manager defines its own scheme for composing software building blocks, now called packages, which has stricter rules than the older schemes defined by operating systems. In particular, the composition scheme requires each package to contain a specification of the aggregate that it can be composed with. These specifications are rather laboriously prepared by a team of packagers, but they're intentionally imprecise, as otherwise they would be obsolete after a few days, which is just another way of saying that the packagers don't really promise that a package will work on every system that conforms to the specification, because there's still a lot of unpredictable diversity in the set of all such systems. Package managers don't solve the fundamental problems of composing software items—rather, they simplify the instructions you have to follow to install software. They don't eliminate the causes and thus the risk of failure.

One particularly important problem that most package managers don't solve is the one I've referred to repeatedly: name conflicts. Every package is identified by an arbitrarily assigned name, and you can't have two packages with the same name installed on your computer. In particular, you can't have two different versions of the same software package. This might not sound so bad at first—after all, why would you want to use two different versions of a package in parallel? There are actually some good reasons, related to bugs and compatibility-breaking changes, but the main problem is elsewhere. Name conflicts make it impossible in practice to compose two software aggregates into a new, more powerful aggregate. Imagine simply downloading an archived software aggregate published by some other researcher and putting it onto your own computer without destroying anything in the process. Reproducibility would become much less of an issue in computational science.

One approach to eliminating name conflicts in software aggregates is compartmentalization. Instead of having a single software aggregate per computer, we would create multiple encapsulated aggregates that are isolated from each other—examples of such encapsulated aggregates include Java's jar files, MacOS application bundles, Docker containers, and Python's virtual environments. Unfortunately, compartmentalization comes at a price. In the one-aggregate universe, existing software building blocks could be freely reused. In the compartmentalized world, reuse across aggregate boundaries is impossible, and aggregates become black boxes. This is particularly inconvenient in a research setting, where experimentation with building blocks ("let's see what happens if I change only this piece") is common.

Doing Better in the Future

Once we understand the causes of a problem, we can figure out how to solve it. A nice example is a recent approach to avoiding name conflicts, which is based on the idea of content-addressable storage. Instead of referring to building blocks by arbitrarily assigned names, a unique short reference handle is computed from the contents of that building block, that is, from the contents of the files that represent the building block on your computer. This can be achieved by using a cryptographic hash function such as SHA-1 (for a plain data file, this means that you use the SHA-1 hash of the data as the filename). Name conflicts can then only occur in the case of a hash collision, whose theoretical probability is vanishingly small and that to the best of my knowledge has never been observed in practice. With content-addressable storage, large amounts of data can be stored with unique short reference IDs and without wasting any storage by allowing duplicates. This is why it's used by the popular version control system Git and also by the Software Heritage project (https://www.softwareheritage.org), whose ambitious goal is to create a permanent archive of all publicly available software source code on the planet.

Two recent package managers, Nix (https://nixos.org) and its offshoot Guix (https://guixsd.org), use content-addressable storage to solve the name conflict problem in software installation. Entire software aggregates thus become composable: you can take the full aggregate of computer A and merge it with the full aggregate on computer B, creating a composite aggregate that has all the functionalities of A and B. Each of these aggregates can contain three different versions of some library without causing any trouble. The price to pay is a more laborious preparation of package descriptions: unlike the first generation of package managers, these newcomers don't build on the conventions of today's operating systems, but have to work against them. Guix has already been used for improving the reproducibility of computational research.[4]

Could we take another step and get rid of libraries and packages completely? If we apply the content-addressable storage idea to the smallest building blocks of a computation, which are subroutines and data structures, we wouldn't need any kind of structured software aggregate. A computer's internal storage would simply be a lookup table that tells you for each unique identifier where to find the full computational building block that it was derived from. To copy software to another machine, you would just need to locate all the required building blocks references and copy them over. You can see this approach in action in the experimental programming platform Unison (http://unisonweb.org).

It might seem that understanding the role of composition in computation is of interest only for programming language designers and developers of operating systems. As a computational scientist, is there any way you can put this knowledge to good use in practice, to make your life easier and your research more reliable? I believe you can, otherwise I wouldn't have written this article!

We might not be developers of composition schemes, but as users we make choices that influence developers. With your newly acquired understanding of software installation and reproducibility, you can now add "ease of managing software aggregates" to your list of criteria when choosing computers and operating systems. You might then want to give Nix or Guix a try, even though they're a bit exotic and, for now, not nearly as mature as what you find in the Linux world. In a similar vein, you could add "ease of distributing and installing modules" to your list of criteria for choosing programming languages and their support tools, in particular, when you make that choice in a teaching context.

Another case of composition that computational scientists frequently have to deal with is data management. As I explained, reading a data file is nothing but a way of composing algorithms with the data they work on into a complete computation. File format conversion, another pain point in the life of computational scientists, is one of those "glue and translation" activities made necessary by the diversity of conventions in computing technology. If you ever have to design a file format, choose one for a piece of software or steer a community toward the adoption of common file formats: doing so greatly helps you understand the issues related to composition, including composition with software (reading and writing data files but also composing data with other data). Do you want a format for individual data items, for collections of data items, or both? Do you expect that in the future, there will be a need to store additional data along with what you want to store now? In science, the answer to the last question is almost always yes, yet few scientific data formats are designed with extensibility in mind.

Finally, if you start to look at computational science from the point of view of building blocks, you'll notice a strange contradiction: when it comes to data, our attitude as a community is very conservative, whereas our attitude to software is progressive. Most scientists agree that datasets should be archived for preservation and published for verification and reuse. There's also widespread agreement on the importance of using community-standard data formats to facilitate the exchange and reuse of data. Concerning scientific software, there's a growing support for publishing its source code, but the goal of maintaining executability is considered unrealistic or too expensive by most of today's computational scientists. Given limited resources, today's dominant view is that we should emphasize new science and update our software with that goal in mind, even if that means rendering older work nonreproducible and the software of the past unusable. Once we realize that software and data are just roles we attribute to different building blocks of a computation, and that both software and data represent scientific knowledge, we can develop a unified view of the relative importance of preserving existing knowledge versus developing new knowledge. Better yet, we might actually succeed in lowering the cost of both preservation and development by a judicious choice of composition technologies.

References

1. A.M. Turing, "On Computable Numbers, with an Application to the 'Entscheidungsproblem,'" *Proc. London Mathematical Soc.*, vo.. 42, no. 2, 1937, pp. 230–265.
2. "Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 2: Instruction Set Reference," Intel, 2015; www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.
3. K. Hinsen, "The Power to Create Chaos," Computing in Science & Eng., vol. 18, no. 4, 2016, pp. 75–79.
4. L. Courtès and R. Wurmus, "Reproducible and User-Controlled Software Environments in HPC with Guix," *Proc. 2nd Int'l Workshop Reproducibility in Parallel Computing* (RepPar), 2015; https://arxiv.org/abs/1506.02822.

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.*