The Art of Computational Science

Open Knowledge

0: Manifesto

Piet Hut and Jun Makino

September 13, 2007

Contents

Preface

0.1 Tacit Knowledge

In many areas of science, computer simulations of complex physical systems cannot be performed with off-the-shelf software packages. Instead, computational scientists have to design and build their own software environment, just as experimental scientists have to design and build their own laboratories, before being able to use them. For a long time, however, the know-how needed to construct computational laboratories has remained only a form of tacit knowledge.

Unlike explicit knowledge that can be found in manuals, this type of implicit knowledge has been alive in conversations among experts, and has been passed down in that way only as a form of oral tradition. This kind of knowledge has not been written down anywhere in sufficient detail to allow it to be passed down without direct personal instructions, or indirect osmosis through personal participation in a joint project.

We have started the *The Art of Computational Science (ACS)* series with the aim of making explicit the implicit knowledge of experts of scientific simulations. Besides offering detailed explanations of the structure of the computer codes used, in an 'open source' style, we provide a deeper layer of knowledge. Besides the *what* and *how* for any computer code, we also provide the *why*: the motivation for writing it the way it was written, within the context in which it was conceived. This will give the user more appreciation for the background of the structure chosen, and most importantly, this will give the user the ability to easily modify and extend the codes presented, without finding oneself at odds with the original style and aim.

0.2 Projects

So far, we have started two projects within the ACS series. The umbrella project is called *Open Knowledge*. Volumes in this series will address issues that are common for all branches of computational science. Other projects will be more specific, addressing issues for a particular set of scientific problems. The *Maya* project is an example of a project in astrophysics, aimed at developing a computational laboratory for the study of dense stellar systems.

It is our sincere hope that our example will inspire the start of various other projects, following the general philosophy of the ACS initiative. We welcome contributions from others along these lines, in any field of science.

The current volume forms the start of the *Open Knowledge* series. It will contain information about the background for the ACS approach, and general issues that come up with respects to its implementation. For now, we have written only the first chapter. In future releases we plan to add more material, based in a large part on reactions from colleagues and students, using our material.

The next volume will contain a detailed discussion of the infrastructure of ACS documentation. We have drawn our inspiration from various sources, including Donald Knuth's notion of *literate programming*. Our current implementation is based on extensions of the Rdoc system (see "http://rdoc.sourceforge.net/"), which we found to have a type of balance between generality and ease of implementation which was just right for our purpose.

0.3 ACS versions

We use the name *The Art of Computational Science* not only for our book series, but more generally for the software environment for which the books provide the narrative. The environment includes the collection of computer codes discussed in the books, together with the infrastructure to make it all work together seamlessly. This implies extensive comments provided in the codes themselves, as well as manual pages.

Our plan is to make successive stable versions of this software environment available, starting with ACS 1.0, which contains a small but self-sufficient core of simple N-body programs and accompanying documentation and narrative. These versions can be freely downloaded from our web site "http://www.ArtCompSci.org". They include all completed and partly completed volumes in our book series. Text, code, and everything else is presented as open source software under the conditions of the MIT license:

0.4 A Historical Note

In preparing for our project, we wrote a complete manuscript, titled *Moving Stars Around MSA)*, also in dialogue form, but aimed more at beginning students who may not yet have much familiarity with computer programming and the use of numerical methods. We have no current plans to continue that particular approach, but even so, the manuscript will remain a useful guide for N-body calculations, with a somewhat different emphasis than the one presented in the *Kali* series.

For example, MSA provides a very quick shortcut to playing with simple algorithms and simple graphics representations of orbit calculations. Both topics are treated in far more detail in the *Kali* series, leading to far more robust and general code, but it will also require more patience from the reader to get there. This is one of the reasons that we decided to keep the MSA volume on our web site indefinitely, as additional introductory material.

0.5 Acknowledgments

Besides thanking our home institutes, the Institute for Advanced Study in Princeton and the University of Tokyo, we want to convey our special gratitude to the Yukawa Institute of Theoretical Physics in Kyoto, where we have produced a substantial part of our ACS material, including its basic infrastructure, during extended visits made possible by the kind invitations to both of us by Professor Masao Ninomiya. We thank Martin Hansen and Douglas Heggie for comments on the manuscript.

Piet Hut and Jun Makino

Kyoto, June 2004

(written for the occasion of the ACS 1.0 release)

Chapter 1

ACS Manifesto

1.1 A Celestial Lab

Computers have given scientists a wonderful virtual laboratory, in which they can simulate any part of reality for which we have sufficient knowledge of the underlying dynamics. Nowhere is this more helpful than in astronomy. While being the oldest science, astronomy has never been able to put stars and galaxies in a test tube – until computers finally gave astrophysicists their own lab.

Letting stars dance, and studying their interactions, is one of the most fun types of pure research we have been engaged in. What has been less fun, though, is the struggle we have endured trying to squeeze results from inadequate software tools. Ideally, tools should be transparent to the user, letting him or her focus on the job to be done, while also being flexible enough to be put to new and unanticipated use. In practice, few software packages live up to this combined ideal of transparency and flexibility.

1.2 Research = Education

Software architecture is a very young craft. Humanity has had experience with building material buildings for many thousands of years, but software building has a history of only half a century. In addition, a major software environment is even more complex, and contains more parts, than the most ornate building. It should not come as any surprise that software failures, delays, and cost overruns are part of our daily news. Clearly, there are still major lessons to be learned about some of the basics involved in setting up a large software project.

In our project, the Art of Computational Science, we explore a radical break with the way software has been developed so far for scientific simulations. We have decided to focus on computational science, simply because that is our terrain of expertise. However, we expect our new approach to have applications for software development in general, and we would welcome and encourage any attempt by others to extend our philosophy to other areas of software development.

At the core of our approach lies the notion that, in any large-scale software project, research = education. As soon as such a project grows beyond what any one person can keep in mind and view as a whole, the different researchers have no choice but to educate each other about the structure and purpose and functionality of the different parts. Documentation is key, but not only documentation of *what* a given program does, but also *how* it fits in with other programs, and more importantly *what type* of vision it is part of, *why* the many design decisions leading to this program were made the way they were, *how* it will be possible to modify and/or extend that design, and so on.

After having struggled for more than two decades in our professional work with the limitations of software products of all kinds – commercial packages as well as legacy codes within our own field, those written by others as well as by ourselves – we have come to the conclusion that there ought to be a better way. After a detailed analysis of our frustrations with most any type of large software package that we have ever used, we came to the conclusion that *the* major lack in almost all of those packages was the lack of complete documentation.

1.3 Open Knowledge

What we mean by complete documentation goes beyond the requirement of 'open source' access to the source code. This is an important first step, but it is only the beginning. In order to use a complex piece of software comfortably, wearing it like well fitting clothes, it should not harbor any secrets. And the best kept secret, even for codes with great manuals and worked-out examples of usage, is often the path that was taken to arrive at the code, while building it. Unfortunately, without that knowledge, any attempt to modify or extend the code is likely to be haphazard at best, and likely to lead to proliferation of code growth with dissonant and poorly connected pieces, leading to the phenomenon of 'spaghetti code' before long.

When we reached these conclusions a few years ago, we decided that the only radical solution to the problem of software development would be to make the design process itself 'open source', on a human level as well as on a code level. Software should run accurately on a computer. And software should be deeply understandable in all its ramification by human beings as well. The latter is the more challenging task, and that is what should consume the major amount of resources, in terms of time and energy of software builders. The result will be a move from 'open source' to 'open knowledge.'

How to write software as a tightly interwoven and complete set of explanations for computers and humans? The answer to the computer part has been developed over the last few decades, in the form of a hierarchical layer of, at bottom, machine code, and layered thereupon assembly code, traditional computer languages, and higher-level scripting languages, graphic user interfaces, etc. In contrast, the answer to the human part has not progressed as much. We have learned to comment our code, to write manuals, and in the best cases to construct elaborate on-line help facilities. But something is missing.

What is missing is the knowledge of what went on in the designer's minds, while writing the software. How often have we wished to have been present, in fly-on-the-wall mode, while software writers discussed and debated, in the middle of being engaged with developing a crucial piece of software that we were struggling to work with! A living human dialogue, recorded during those crucial stages, would undoubtedly have given valuable clues to the tacit assumptions and motivations that color any software product. And knowledge of those clues would be invaluable in any attempt to extend that product in major new ways.

1.4 Dialogues

The solution we felt driven to was simple, in retrospect: to take the classic literary device of a dialogue, following in the footsteps of Plato and Galileo, who used dialogues to convey not only the information content of their knowledge, but also the flavor and setting and ramifications.

Of course, we realized that it would take a lot of our time to write dialogues covering the whole process of software development. In fact, we had no idea at all, in the beginning, of how much time this would actually take us. The amount of essential but tacit knowledge hidden in the mind of any expert is enormous, typically far more than the expert is aware of, as emphasized already half a century ago by the physical chemist turned social scientist Michael Polanyi.

Even in areas where we were convinced that we had all the necessary knowledge in our finger tips, as soon as we forced ourselves to be explicit about all the 'hows' and 'whys', we found ourselves discovering new aspects in the middle of our dialogue writing. In quite a number of cases, we found new and better ways to do standard operations in N-body simulations, to our great surprise.

1.5 Audience

Having worked with this approach now for a few years, we have reached the point that we feel confident in making our results publicly available. From the beginning, we have resisted the temptation to define an audience, since we viewed our project as pure research in the strictest possible terms. Starting with a target audience would slant our dialogues in a particular applied direction, diminishing completeness and free flow.

In fact, from the beginning we decided to write this series of books for an audience of only two: younger versions of ourselves. In other words, we set out to write the type of books that we wished we had been able to read when we started our careers in computational science. Throughout our writing, we have stuck to this idea of this original audience of two – or zero, really, since our younger selves are no longer here – and we had no idea to what extent our ruminations would be useful for others.

It was therefore with no particular expectations, when we tested out our first volume with a group of students who attended an N-body School, organized by Christian Boily in Strasbourg, France, in March 2004, while we were both visiting the Observatory there. To our happy surprise, most students in fact reacted the way we would have reacted, had we had our book in our hands when we were students. While it is still too early to judge how effective our approach will be for others, this initial experience formed a strong encouragement for us to proceed with our unconventional approach.

nil nil nil nil nil nil nil nil nil nil