Chapter 2

Why Software Engineering?

> A computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.
>
> *Bill Bryson*

Welcome to software engineering! If you are reading these lecture notes, you have started, perhaps without being aware of it, a journey into becoming a software engineer. Being a software engineer (also called a software developer) is something more than just being a computer programmer. To address the homework in your class, to write your doctoral dissertation, or to complete publishable papers, you will need to develop a whole computational system and not just to write one or a few pieces of code. You might need, for instance, code to gather and clean data, code to compute the equilibrium dynamics of a model that you postulate, code to estimate the model using observed data, code to perform robustness analysis to see how your model react to changes in some fundamental assumptions, etc. All these tasks will require that you understand the requirements of the whole project, that you plan a detailed roadmap of the steps involved by those requirements, that you organize and design the software to be coded -including consideration such as how to divide the work with coauthors or research assistants-, that you prepare procedures to test the accuracy of the results, and that you have an idea of how to keep the code updated and ready for possible changes. In other words, computational work in economics goes much beyond coding a smart algorithm or generating a table of results.

In the next pages, we will discuss some primary considerations of how to accomplish the view of a complete computational project. But, before getting into the details, we must outline the goals that any successful software project must satisfy.

2.1 The Goal

The goal of a computational project is to produce code that is: 1) correct; 2) efficient; 3) maintainable; 4) reproducible, 5) documented; 6) scalable; and 7) portable. Let us spend some time discussing each of these seven requirements.

2.1.1 Correctness

Correctness is the most straightforward requirement to understand and the most fundamental. If we are spending time and effort on a project is because we care about the answers that we get from it (or the usefulness of such answers for other purposes). Thus, that the answer we get is correct is a fundamental condition: an incorrect answer is not only useless from a scientific perspective, but it might also be damaging by leading us into wrong conclusions or decisions. Ensuring correctness, however, will be harder than it seems at first sight. Computer code will be plagued by bugs, some of which can be quite subtle to identify even after much diligence. Software engineering emphasizes ideas such as proper design, good coding practices, systematic debugging, and testing that minimize the incidence of mistakes.

2.1.2 Efficiency

Efficiency is a more practical but not much less important requirement than correctness: as an economist, you want to finish your Ph.D., to get tenure, to become an influential researcher in *finite* time. Efficiency is much more of a daunting task than it might seem. For instance, modern dynamic economic models are often characterized by a multitude of state variables, sometimes in the hundreds or even thousands of them (for example, when we have rich individual heterogeneity or multiple locations for economic activity). And, in econometrics, we might need to parse through millions of observations or complex estimators that require costly simulation-based methods and expensive parameter sweeps.

Most of the algorithms that we will present in these lectures will suffer from a "curse of dimensionality." Loosely speaking, this curse means that the computational demands required to solve a model increase exponentially in the number of state variables: a model with two state variables is more than twice as demanding to solve as a model with one state variable, and a model with three state variables much more than three times as demanding,

and so on. Much of our effort will be spent in handling this curse or, at least, in reducing its acuteness without sacrificing precision and accuracy.

Nevertheless, it is crucial here to introduce a fundamental classification regarding the computational time. The time that a project requires can be divided between *coding time* and *running time* (finer subdivisions are often presented, but this one will suffice for our purposes). Coding time is the number of hours involved in planning, coding, debugging, and testing your code. Running time, in comparison, is the number of hours involved in executing the code and obtaining the desired results.

Articles in economics are often written as if the only consideration regarding the computational requirement was running time. This is not a surprise: running time is objective. Given the same computer and the same conditions of the operating system, one code will take the same amount of time to run if one person executes it than if another person does (plus/minus a small variation due to fluctuations in the state of the operating system and memory, but we can ignore those here). In comparison, a good software developer will produce code much faster than a poor one. In fact, one objective of these lectures is to make you a faster developer. In real life, most projects will have coding times that are an order of magnitude larger than their running times and large running times correlate with even larger coding times (there are some exceptions, such as intensive Markov chain Monte Carlos, which can be quite fast to code and extremely expensive to run). Improving your coding time by, for instance, 10 percent, maybe then much more consequential than improving your running time by 50 percent. Also, when you are making decisions about which algorithms to apply to your problem, coding time may be a first-order consideration. Fast coding time will give you extra time to work on other tasks in your research. Given a choice between two algorithms that deliver an answer of the same quality, you want to consider both coding and running time and pick the one that minimizes the sum of the two times, not the one that minimizes running time alone.

2.1.3 Maintainability

Maintainability is a subtle requirement and one that young researchers only learn to appreciate with experience. Think about how the publication process works in economics. You submit a paper to a journal. After some period (sometimes unbearably long), you will get a decision from the editor. By far the two most common decisions would be a "rejection" or a "revise and resubmit."[1] In both cases, either before re-submitting the paper to a different

[1] One of us (Jesús) has been serving as an editor of several journals for over a decade, and in thousands of decisions, he does not recall ever accepting a paper "as-if" in the first round. He does not believe his experience is unique.

journal or to comply with the editor's requests, you will need to go back to your original code and introduce changes to it. Often these changes can be substantial. A code that is easy to maintain is a code that allows you to introduce changes without wholly disrupting its structure. For instance, it is a code where you have not implemented a "clever" trick to accelerate running time that wholly depends on a particular parametric assumption that the editor can ask you to generalize. Or it is a code that has written in modular parts that can be easily modified separately.

Similarly, you may want to extend one of your previous papers and to be able to "recycle" old code or to use it as a jumping board for the new code is most valuable and it will make you much more productive. And, finally, once you start teaching, you will find that being able to go back to some of your old codes will provide with great pedagogical material for your lectures.

2.1.4 Reproducibility

You want to ensure that other researchers (and your future selves) must be able to replicate your quantitative results. Replication is not only the foundation of scientific progress, but also a fundamental check that our code is correct. Reproducibility has gained much importance in recent years as a response to the realization that many results in the literature were difficult to generate by third researchers. Furthermore, code repositories, which allow for much easier reproducibility, have become cheap and easy to access.

2.1.5 Documentation

Other researchers (and your future selves) must be able to understand how your code works. There is nothing more frustrating than looking at your code from five years ago and realizing you cannot follow your work. Believe us: it happens. Good documentation will help you to open your old code (or read someone else's) and be up and running in a short time. Furthermore, good documentation is fundamental to achieve the goal of reproducibility outlined above.

2.1.6 Scalability

The code that you develop should be easy to scale for other applications, such that you (or other researchers) can employ it as a base for further computational projects. This will reduce time requirements in future projects and incentivize the use of code that has been thoroughly tested over time.

2.1.7 Portability

You want to produce code that can work across a reasonable range of machines with as few changes as possible. First, because you may use different machines in the same project. For instance, you may develop your code in your laptop and later run it, with a higher precision level, in a more powerful server. And you might be working with a coauthor that has a different machine than yours. Second, because otherwise reproducibility will not be achieved. Many journals ask nowadays to submit code that can be run on their machines before final acceptance of a paper. Third, because you may migrate to other computers in the middle-run and you do not want to be stuck with code that only runs on the laptop you had ten years ago. Fourth, code that is portable is also usually a code with fewer bugs. Being forced to run in different environments means that errors float to the surface more prominently.

2.2 The Means

All the requirements in the previous section are easier to state than to implement. There is, however, much wisdom accumulated over decades in computational-intensive fields and by the industry. More concretely, software engineering has emerged as a vibrant discipline that aims at developing correct, efficient, maintainable, reproducible, documented, scalable, and portable software.

Although the principles of software engineering were implicit in the work of coders since the start of modern computers, the field received a decisive impulse with the creation of a study group on computer science in the fall of 1967 by the NATO Science Committee and the subsequent organization of two conferences on the fall of 1968 and 1969 of software engineering conferences. The objective of the group and the conferences was to address what some of the participants were calling the *software crisis*. As the use of computers was booming worldwide during the 1960s, more and more computational projects were failing at producing the required software in a timely and correct way. Code that was delivered late, over-budget, and without being able to meet the user requirements was becoming dangerously common. Developers and analyst reacted to the crisis by focusing on identifying and refining techniques that will reduce the problems of future projects.

Nowadays, software engineering is a standard part of an undergraduate degree in computer science. In the next chapters, we will cover some of the basics ideas and tools of software engineering. We will adapt them, though, to the requirements of an economist, at least, as determined by our own experience in the field. That is why we call these lectures "for Economists." For instance, you will probably not have different "releases" of code, Unified

Modeling Language (UML) and design patterns will not be a significant concern, and code testing will be undertaken differently than for commercial software.[2] At the same time, speed and reproducibility will be more critical than in other fields.

Also, we will cover some material that it is taught in introductory courses in computer science (or just assumed that students will pick up by themselves one way or another), but that economists may be less familiar with.[3] For example, we will discuss integrated development environments (IDEs), debuggers, profilers, objected-oriented programming (OOP), version control systems, and many others. After many years of teaching this material, we have discovered that, more often than not, the real barrier for young researchers is not to understand an algorithm at an abstract level (nowadays, doctoral students in economics have a high level of mathematical maturity), but knowing how to implement it in practice or do it efficiently. With these lectures on high-performance computing, we aim at bridging some of these difficulties.

2.2.1 Textbooks and classic monographs

We are, in any case, realistic. The brief introduction in these lecture notes that cannot substitute a real course on software engineering (and other techniques in high-performance computing) in your local computer science department or some online course.[4] It cannot replace either of the knowledge and detail that you can find in standard books of software engineering and development.

There are dozens of undergraduate textbooks in software engineering and selecting a few of them is unfair to all other ones. But two popular textbooks that we have used are:

1. *Object-Oriented and Classical Software Engineering (8th Edition)*, by Stephen Schach, McGraw-Hill Education.

2. *Software Engineering (10th Edition)*, by Ian Sommerville, Pearson.

There are also some classic monographs. The most famous is *The Mythical Man-Month: Essays on Software Engineering*, by Fred Brooks. Although more centered on industry applications, Brooks has such an astute eye for the problems of code development that many readers over the decades have profited immensely from reading the book.

[2] With more and more economists moving into industry, this might be less true now than a decade ago, but in any case, will still skip this material.

[3] If you have a degree in computer science or engineering, you might find the next chapters trivial. Indeed, our coverage of these topics is merely introductory. You can just skim the chapters to check whether there is some material that you may want to review and move quickly to the chapters on more substantive computational methods.

[4] `Coursera` and several commercial publishers such as `Packt` and `O'Reilly` offer a rich selection of online material.

As we move along these lectures, we will cite many other books that we find of relevance or particular usefulness.

2.2.2 The manual: your best friend

The very first source for you to learn about high-performance computing is to read the technical documentation (on, in the less delicate language of the community: RTFM or "read the f*** manual"). Manuals are often (but not always!) well designed and full of information about the tool you are using. Also, remember that the on-line versions of manuals are usually more detailed and updated.

2.2.3 Additional resources

Nowadays, thanks to the internet, economists have access to a wealth of information that goes well beyond their personal experience.

The first port of call should always be searching the internet, or, "Google it yourself (GIYF). The ability of internet search engines to find information is breathtaking. We find that it is usually quicker to introduce a search term in Google (i.e., "how to format input/output in `C++`") and look at one of the top returns than to pick up a textbook in our office and search for the same answer, even when we positively know that such answer is in the textbook. As someone said: "Am I a developer, or just a good googler?"[5]

Another fantastic resource is `Youtube` and other video web pages. These sites are full of tutorials and lectures. Although one has to apply common sense to separate the wheat from the chaff, once you have gained some experience, such task is not particularly onerous, and you can get access to tons of excellent lessons, from five minutes tutorials on how to configure your editor to two hours lectures on functional programming.

Three other resources that we rank highly are:

1. Software carpentry: http://software-carpentry.org/index.html, a repository of workshops and tutorials on basic computing skills for academic work. Their lessons, freely available under the Creative Commons - Attribution License, are gems of concision and information.

2. Stack Overflow: http://stackoverflow.com/, a question and answer site center in computer programming. Although the quality of answers varies (and some user are

[5]This question, of course, can be interpreted in two ways: relying too much on the internet might lower your understanding of what you are doing and lead to the introduction of unexpected bugs.

plainly rude with newcomers), you will find tons of examples of code to perform basic tasks.

3. O'Reilly: http://oreilly.com/, is a media company specialized in publishing books and videos related to computer programming. Its books are often excellent sources to learn both basic and advanced computer skills.

4. no starch press: https://www.nostarch.com, publishes a nice (although not very long) series of books on programming and coding tools.

5. Slant: www.slant.co, is an IT product recommendation community organized around problems (i.e., best editor for programmers in Mac, best color themes for text editors). Sometimes the recommendations are not our first choices (remember: the work of an average coder is different than the work of a developer in scientific applications), but they usually list the most reasonable choices, and you can choose among them based on your judgment.

2.3 Tools and Techniques

In the next chapters, we will discuss tools to help you accomplish your computational project such as editors, IDEs, report generators (`Jupyter`, `Markdown`, `Pandoc`, ...), compilers, libraries (modules, toolboxes,...), automatized compilation (`make files`), `lint` and other static code analyzers, debuggers, and profilers. We will also offer some discussion on programming languages (`C++`, `Fortran`, `Python`, `Matlab`, `R`, ...) and a tutorial on `Julia`, a modern, high-performance open-source language.

It will be unfeasible, and probably undesirable, to cover all existing tools. First, because we (or anyone for the matter) are not familiar with all of them. Second, because they constantly change and are adapted to new problems and requirements. Third, because such coverage will be too cumbersome to be of practical use. Instead, we will select a few of the most important ones, give you a flavor about them, and point out directions that you can follow to improve your knowledge on your own and decide which one is best for you (all tools have an aspect of personal taste). It is much more important that you learn about how tools can help your computational work than about the details of one specific tool, even if just because those details can change with a new release of the tool tomorrow morning.

We will also introduce you to some basic techniques in software development. We will talk, for instance, about programming approaches (structured, object-oriented, functional,...), coding style, version control, prototyping, testing, performance optimization, parallelization

(`OpenMP`, `MPI`, `GPU`,...), and multilanguage programming (calling `Python` packages in `Julia`, `Rcpp`,...). Again, the emphasis will be more on providing you with some basic ideas than in compiling an encyclopedic coverage of topics.

2.4 Warnings

Before we move to the next chapters of these lectures, we would be remiss if we were not to offer some basic warnings.

First, and most important, nothing in these lectures is a substitute for common sense, self-discipline, and hard work. Algorithms in computation usually work under conditions that are hard to verify in practice (i.e., the algorithm will find a solution to a non-linear equation if we start "sufficiently close" to the exact solution). One needs to apply common sense when applying the algorithm (i.e., use economic "intuition" to select an initial value that is likely to be "sufficiently close"), self-discipline to follow procedure (i.e., coding the algorithm according to the rules we will outline in later chapters), and hard work (i.e., not to be satisfied with the very first answer the algorithm produces and check again and again that this indeed the solution we are looking for or that there are no other solutions).

Among the most important rules of self-discipline is to continually ask yourself, when something goes wrong: is this a problem with the "economics of the problem" or with the "computational implementation"? If the economics is wrong, no amount of good software engineering will rescue the project.

Second, experience is more important than concepts, tools, and techniques. You can read all the books ever writing about swimming and watch as many videos on the internet as you wish of swimmers but, at the end of the day, there is no substitute for jumping into the water and learn to swim in real-life conditions. Similarly, one only really learns about, let's say, numerical optimization when one spends many hours trying to maximize a function in the computer.

Third, there are no silver bullets out there. Computational problems are hard: that is what makes the field both interesting and challenging. No matter how clever your numerical algorithm is, one can cook up a case where it will break down. Old and proven procedures have survived the test of time for a reason. And everyone will oversell their product. Software vendors want to make a profit, and they will very carefully select some benchmark results about the speed or power of their products. As corporate lawyers will tell you, one does not need to lie to be highly misleading. Similarly, researchers fall in love with their new algorithms and will only see the positive parts of it. As they say in Naples, Italy: Ogni

scarrafone è bello a mamma soja ("every beetle is beautiful to its mother").[6]

Finally, beware of the temptation of "If I just update my computer/OS/app everything would be fine." Of course, sometimes your computer may be the bottleneck that is slowing you down and purchasing a new one may solve your problems. And, yes, often switching from one programming language to another will allow you to handle the tasks at hand. But more often than not, when you reach a wall, the real constraints lay elsewhere: you might have picked the wrong algorithm, you may have a bug or the economic model you are computing needs to be changed. A shiny new computer will not fix any of these problems.

But enough of preliminaries and warnings and let's get into business.

[6]Beetles have, in any case, powerful admirers. The famous British biologist J.B.S. Haldane (1892-1964) loved to say, tongue-in-check, that the main lesson that he had learned from evolution was that "God has an inordinate fondness for beetles," given how many different species of them existed.