A Comparison of Programming Languages in Macroeconomics[*]

S. Borağan Aruoba[†] Jesús Fernández-Villaverde[‡]
University of Maryland University of Pennsylvania

May 4, 2015

Abstract

We solve the stochastic neoclassical growth model, the workhorse of modern macroeconomics, using `C++14`, `Fortran 2008`, `Java`, `Julia`, `Python`, `Matlab`, `Mathematica`, and `R`. We implement the same algorithm, value function iteration, in each of the languages. We report the execution times of the codes in a `Mac` and in a `Windows` computer and briefly comment on the strengths and weaknesses of each language.

Key words: Dynamic Equilibrium Economies, Computational Methods, Programming Languages.
JEL classifications: C63, C68, E37.

[*]We thank Manuel Amador for his help in making our `Python` and `Mathematica` codes more idiomatic, Matthew MacKay and John Stachurski for their help with `Numba`, basthtage for moving our code to `Cython`, Matt Dziubinski and Santiago González for alternative implementations of our `C++` code, Anastasios Stamulis for porting our code to `C#`, Marco Lugo for porting our code to `JavaScript`, Luigi Bocola, Gustavo Camilo and Pablo Cuba-Borda for research assistance, Thomas Jones, Thibaut Lamadon, Florian Oswald, Pau Rabanal, Pablo Winant, the members of the `Julia` user group for comments, five referees and the editor for helpful comments, and the NSF for financial support.

[†]University of Maryland, <aruoba@econ.umd.edu>.

[‡]University of Pennsylvania, NBER and CEPR <jesusfv@econ.upenn.edu>.

1. Introduction

Computation has become a central tool in economics. From the solution of dynamic equilibrium models in macroeconomics or industrial organization, to the characterization of equilibria in game theory, or in estimation by simulation, economists spend a considerable amount of their time coding and running fairly sophisticated software. And while some effort has been focused on the comparison of different algorithms for the solution of common problems in economics (see, for instance, Aruoba, Fernández-Villaverde, and Rubio-Ramírez, 2006), there has been little formal comparison of programming languages. This is surprising because there is an ever-growing variety of programming languages and economists are often puzzled about which language is best suited to their needs.[1] Instead of a suite of benchmarks, researchers must rely on personal experimentation or on "folk wisdom."

In this paper, we take a first step at correcting this unfortunate situation. The target audience for our results is younger economists (graduate students, junior faculty) or researchers who have used the computer less often in the past for numerical analysis and who are looking for guideposts in their first incursions into computation. We focus on a macroeconomic application but we hope that much of our conclusions and insights carry over to other fields such as industrial organization or labor economics, among others.

We solve the stochastic neoclassical growth model, the workhorse of modern macroeconomics, using `C++`, `Fortran`, `Java`, `Julia`, `Python`, `Matlab`, `Mathematica`, and `R`. We implement the same algorithm, value function iteration, in each of the languages, and measure the execution time of the codes in a `Mac` and in a `Windows` computer. The advantage of our algorithm, value function iteration, is that it is "representative" of many economic computations: expensive loops, large matrices to store in memory, and so on. Thus, while our investigation does not entail a full suite of benchmarks, both our model and our solution method are among the best available choices for our investigation. In addition, our two machines, a `Mac` and a `Windows` computer, are perhaps the two most popular environments for software development for economists.

The key take-aways of our analysis are as follows:

1. `C++` and `Fortran` are considerably faster than any other alternative, although one needs to be careful with the choice of compiler. The many other strengths of `C++` in terms of capabilities (full object orientation, template meta-programming, lambda functions, large user base) make it an attractive language for graduate students to learn. On the

[1] This also stands in contrast to work in other fields, such as Prechelt (2000) and Lubin and Dunning (2013), or web projects, such as *The Computer Language Benchmarks Game* (see `http://benchmarksgame.alioth.debian.org/`).

other hand, Fortran is simple and compact – and, thus, relatively easy to learn – and it can take advantage of large amounts of legacy code.

2. Julia delivers outstanding performance, taking only about 2.5 times longer to execute than C++, while Matlab takes about 10 times longer. Given how close Julia's syntax is to Matlab's and the fact that it is open-source and that the language has been designed from scratch for easy parallelization, many economists may want to learn more about it. However, Julia's standard is still evolving (causing potential backward incompatibilities in the future) and there are only a few libraries for it at the moment.

3. While Python and R are popular in economics, they do not perform well in our application, taking 44 to 491 times longer to execute than C++.

4. Hybrid programming and special approaches can deliver considerable speed-ups. For example, when combined with Mex files, Matlab takes only about 1.5 times longer to execute than C++ and when combined with Rcpp, R takes about 4 times longer to execute. Similar numbers hold for Numba (a just-in-time compiler for Python that uses decorators) and Cython (a static compiler for writing C extensions for Python) in the Python ecosystem. While Mex files were faster, we found Rcpp to be elegant and easy to use. These numbers suggest that a researcher can use the friendly environment of Matlab or R for everyday tasks (data handling, plots, etc.) and rely on Mex files or Rcpp for the heavy computations, especially those involving loops.

5. The baseline version of our algorithm in Mathematica is very slow, unless we undertake a considerable rewriting of the code to take advantage of the peculiarities of the language.

Some could argue that our results are not surprising as they coincide with the guesses of an experienced programmer. But we regard this comment as a point of strength, not weakness. It is a validation that our exercise was conducted under reasonably fair conditions. We do not seek to overturn the experience of knowledgeable programmers, but to formalize such experience under well-described and explicitly controlled conditions and to report the information to others.

We also present some brief comments on the difficulty of implementation of the algorithm in each language and on the additional tools (integrated development environments or IDEs, debuggers, etc.) existing for each language. While this is a treacherous and inherently subjective exercise, perhaps our pointers may be informative for some readers. Since the codes are posted at our github repository, the reader can gauge our results and remarks for himself.[2]

[2] https://github.com/jesusfv/Comparison-Programming-Languages-Economics

The rest of the paper is structured as follows. First, in section 2, we introduce our application and algorithm. In section 3 we motivate our selection of programming languages. In section 4, we report our results. Section 5 concludes.

2. The Stochastic Neoclassical Growth Model

For our exercise, we pick the stochastic neoclassical growth model, the foundation of much work in macroeconomics. We solve the model with value function iteration. In that way, we compare programming languages for their ability to handle a task such as value function iteration that appears everywhere in economics and within a well-understood economic environment.

In this model, a social planner picks a sequence of consumption c_t and capital k_t to solve

$$\max_{\{c_t, k_{t+1}\}} \mathbb{E}_0 \sum_{t=0}^{\infty} (1-\beta)\, \beta^t \log c_t$$

where $\mathbb{E}_0$ is the conditional expectation operation, β the discount factor, and the resource constraint is given by $c_t + k_{t+1} = z_t k_t^\alpha + (1-\delta) k_t$, where productivity z_t takes values in a set of discrete points $\{z_1, ..., z_n\}$ that evolve according to a Markov transition matrix Π. The initial conditions, k_0 and z_0, are given. While, in the interest of space, we have written the model in terms of the problem of a social planner, this is not required and we could deal, instead, with a competitive equilibrium.

For our calibration, we pick $\delta = 1$, which implies that the model has a closed-form solution $k_{t+1} = \alpha\beta z_t k_t^\alpha$ and $c_t = (1-\alpha\beta)\, z_t k_t^\alpha$. This will allow us to assess the accuracy of the solution we compute. Then, we are only left with the need to choose values for β, α, and the process for z_t. But since $\delta = 1$ is unrealistic, instead of targeting explicit moments of the data, we just pick conventional values for these parameters and processes. For β we pick 0.95, 1/3 for α, and for z_t we have a 5-point Markov chain:

$$z_t \in \{0.9792, 0.9896, 1.0000, 1.0106, 1.0212\}$$

with transition matrix:

$$\Pi = \begin{pmatrix} 0.9727 & 0.0273 & 0 & 0 & 0 \\ 0.0041 & 0.9806 & 0.0153 & 0 & 0 \\ 0 & 0.0082 & 0.9837 & 0.0082 & 0 \\ 0 & 0 & 0.0153 & 0.9806 & 0.0041 \\ 0 & 0 & 0 & 0.0273 & 0.9727 \end{pmatrix}$$

The transition matrix is similar to the one that would come from a discretization of an AR(1) process for (log) productivity following Tauchen's (1986) procedure, except that we move mass from the diagonal to the upper and lower bands to induce more movements across states and to create a more challenging computation. Note that the algorithm leads to identical results in all languages, taking the identical path. As such, the relative speed comparisons that we report below are robust to different parameter values, including values of $\delta < 1$.

The recursive formulation of this problem in terms of a value function $V(\cdot, \cdot)$ and a Bellman operator (where we have already imposed that $\delta = 1$) is:

$$V(k, z) = \max_{k'} (1 - \beta) \beta^t \log (zk^\alpha - k') + \beta \mathbb{E} \left[V(k', z') | z \right]$$

We solve this Bellman operator using value function iteration and by searching for the optimal value of k'. We use a grid of $17,820$ points for k uniformly distributed ± 50 percent of the steady-state value of capital. To make the algorithm as transparent as possible, we force the choice of k' to be within the capital grid. We take advantage of monotonicity in the policy function and of an envelope condition to avoid unnecessary computations (see the online appendix for details). We choose this grid size so that `C++` or `Fortran` would solve the problem in about one second. Shorter run times would cause large relative measurement errors (due to issues such as the situation of the cache at any given time). We impose a tolerance of $1.0e - 07$ for convergence. The value function takes 257 iterations to converge. All codes achieve convergence in 257 iterations, the computed value functions are the same (up to at least 14 decimal digits), and the policy functions found in each code select the same level of capital in the grid for all combination of state variables in the grid.

In Figure 1, we plot the value (top panel) and policy function for capital next period (middle panel) along the capital dimension, with each color representing a different value of z_t. The value and policy functions are, as expected, increasing and concave. We also plot the difference between the exact and approximated policy function for capital in percentage terms (bottom panel). The maximum error is only -0.0059 percent, which illustrates the high accuracy achieved with 17,820 points for k.

3. Selection of Programming Languages

Since `Fortran` came around in 1957, hundreds of programming languages have been created. Even limiting ourselves to languages that have acquired a solid user base circa 2014, we need to choose from among dozens of them.

Fortunately, the task is simpler than it seems. There is little point to picking languages such as `Perl` or `PHP`, neither of which is particularly suited to, nor widely used for, scientific computing. Also, many languages are close relatives of each other and one member of the family will suffice for our comparison. With our choices of languages, we cover a wide range of possibilities, and, with the exception of the functional programming languages discussed below, we feel we have covered all the obvious choices for numerical computation.

3.1. Compiled Languages

Among compiled languages, we select `C++14`, `Fortran 2008`, and `Java`. `C++` is, perhaps, the most powerful language among those widely used. Together with `C` and `Objective-C`, it constitutes the backbone of much of the modern computing world. According to the well-cited TIOBE Index of programming language popularity (May 2014 edition), `C` is ranked number 1, `Objective-C` is ranked number 3, and `C++` number 4, with a total popularity of 34.7 percent.[3] Our `C++` code does not use any specific `C++` features such as objects. Thus, the `C` and `Objective-C` codes (which can be found on the github page) are nearly equivalent. We checked, also, that the run time of the `C` and `Objective-C` codes was nearly exactly the same. Thus, we will only report the `C++` running time.[4]

Two other relatives of `C++` are `C#` and `D`. `C#` is widely used in the industry (`C#` is ranked 6th in the TIOBE Index with 3.75 percent). However, design considerations that make `C#` attractive for commercial applications also render it slower for numerical computation and, thus, it is rarely employed for the tasks we are concerned with in this paper.[5] `D`, which generates code usually roughly of the same speed as `C++`, is less popular (ranked 26th with 0.60 percent). Including all five languages, the `C` family accumulates a popularity of 39.04 percent. `Swift`, a replacement for `Objective-C`, is not designed, at this moment, as a programming language for use outside `OS X` and `iOS`.

`Fortran`, the oldest language of all, still maintains a significant presence in high performance scientific computing and among economists. Its latest incarnation, `Fortran 2008`, is updated with modern features and innovations such as coarrays. Reflecting this niche nature of `Fortran`, the TIOBE ranks it 32nd, with a 0.42 percent popularity.

An important strength of both `C++` and `Fortran` is that both languages have access to well-tested, state-of-the-art open source libraries to implement a large number of standard algorithms that appear in computations in macroeconomics. Beyond the more traditional `LAPACK`, `C++` has, for example, `Armadillo` and `Boost`.[6] Armadillo, with its syntax deliberately

[3]The TIOBE Index gathers information about the use of programming languages for a wide range of application development. Thus, it offers an imperfect gauge of the use of programming languages within the scientific computing community. Unfortunately, we could not find an index focused on the use of languages among scientists. See: `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`.

[4]For a comparison of syntaxes, see the Hyperpolyglot at `http://hyperpolyglot.org/cpp`.

[5]Anastasios Stamulis reported that a conversion of our `C++` to `C#` is around 20 percent slower.

[6]For `Armadillo`, see `http://arma.sourceforge.net/`, and for `Boost`, `http://www.boost.org/`.

based on `Matlab`'s, is particularly easy to use for researchers with less experience.

`Java` is a common vehicle for undergraduate education and the availability of the `Java Virtual Machine` in practically all computer environments makes it an attractive choice. In the TIOBE Index, it is ranked 2nd, with a popularity of `5.99` percent.

The performance of compiled languages also depends on the compiler used to generate the executable files.[7] Thus, we select a number of those. For `C++`, in the Mac machine, we pick `GCC`, `Intel C++`, and `Clang` (which shares the `LLVM` – lower level virtual machine – with `XCode` and delivers nearly identical speed) and in the Windows machine, `GCC`, `Intel C++`, and `Visual C++`. For Fortran, in both machines, we select `GCC` and `Intel Fortran`.[8] For `Java`, we select the standard `Oracle JDK`.

3.2. Scripting Languages

We pick as our scripting languages `Matlab`, `Mathematica`, `R`, `Python`, and `Julia`. `Matlab`, `Mathematica`, and `R` are sufficiently known among economists that it is not necessary to elaborate on our choice.[9]

`Python` is an elegant open-source language that has become popular in the scientific community (see Sargent and Stachurski, 2014), in particular the 2.7 version. Since there are different implementations of `Python`, we select `CPython`, the default `Python` interpreter that comes with `Mac` and `Linux` machines, and `Pypy` (`http://pypy.org/`), a speed-oriented replacement virtual machine that uses a just-in-time compiler. Our `Python` code for `CPython` and `Pypy` was exactly the same and it uses the `Numpy` library for matrix operations.

`Julia` (`http://julialang.org/`) is a new open-source high-performance programming language with a syntax very close to `Matlab`'s, `Lisp`-style macros, and many other modern features, and it also uses a just-in-time compiler for speed based on the `LLVM`. Three particularly attractive features of `Julia` are as follows. First, `Julia`'s default typing system is dynamic (to facilitate fast coding), but it is possible to indicate the type of certain values to avoid type-instability problems that often decrease speed in dynamically typed languages. Second, `Julia` can call `C` or `Fortran` functions without wrappers or APIs. Third, `Julia` has a library that imports `Python` modules and provides wrappers for all of the functions on them.

We do not use `Octave`, an open source clone of `Matlab`, because it is well-known that it is noticeably slower than `Matlab`. We also do not include `Gauss` (`http://www.aptech.com/`)

[7] See, for example, the comparison at `http://www.polyhedron.com/fortran-compiler-comparisons.`

[8] We could not find data on compilers' market share, but our picks include the most popular compilers in user forums. Our experience with other compilers, such as `PGI`, has been less satisfactory in terms of the speed of the generated executables.

[9] Marco Lugo provided us also with a `JavaScript` version of the code, posted on `github`. It runs approximately at the same speed as `Matlab`.

because, in preliminary testing, we found that it took roughly seven times longer to execute than `Matlab`.

3.3. Functional Programming Languages

The big missing items in our list of languages are those that belong to the functional programming family that inherits the insights from `Lisp`. In a companion paper (Amador, Aruoba, Fernández-Villaverde, 2014), we elaborate on the advantages of functional programming for economics and explain how to extend our benchmark investigation to functional languages such as `Ocaml` or `Haskell`. Since this comparison involves a number of issues of its own, we prefer to avoid them here to keep the paper focused.[10]

3.4. Hybrid and Special Approaches

Most languages allow for the used of mixed programming. This is particularly useful in `Matlab` and `R`, where one can send computation-intensive parts of the code to `C++` and keep the rest of the code in an easier scripting language format. Thus, in addition to "pure" `Matlab` and `R`, we also use `Mex` files, where part of the code is written in `C++` and compiled before the `Matlab` code runs, and `Rcpp`, a package in `R` that facilitates the integration of `R` and `C++` (`http://cran.r-project.org/web/packages/Rcpp/index.html`). In both cases, we sent to `C++` the Bellman operator that updates the value function and that consumes nearly all the computing time.

As with `Matlab` and `R`, we compile in `Python` the Bellman operator. We do so using two different approaches: first, with `Numba` (`http://numba.pydata.org/`), a just-in-time compiler that uses decorators to compile `Python` to `LLVM`; second, with `Cython (http://cython.org/)`, a compiler that converts type-annotated Python into generated C code that can be imported as a module.[11]

Finally, we have `Mathematica`. Although `Mathematica` allows for multiparadigm programming (including our imperative algorithm), its kernel strongly prefers a more functionally oriented approach. Thus, we will also use its `Compile` function plus a rewriting of the code to take advantage of the peculiarities of the language. While this would make the results from this last computation hard to interpret, some readers may find them of interest.

[10] We run, though, an experiment with `Scala`, a "trendy" language that allows for multi-paradigm programming by integrating imperative, object-orientation, and functional features. Our `Scala` code built with imperative features runs, not surprisingly, at roughly the same speed as the `Java` code (`Scala` compiled Java bytecode runs in the `Java Virtual Machine`) and, thus, we decided not to include it in our results.

[11] The `Python` ecosystem is incredibly rich and continuously expanding. Thus, it is well beyond our abilities to survey every single possibility. The interested reader can check a list of compilers at `http://compilers.pydata.org/`.

4. Results

4.1. Speed Comparisons

We start with a speed comparison. A straightforward execution time benchmark has three advantages. First, it is easier to measure. Second, speed comparisons give us an indication of the potential benefits to researchers from mastering a new programming language. Third, many real-life applications in macroeconomics are considerably more computationally intensive than our simple exercise. As we increase, for example, the number of state variables or we nest a value function iteration in an estimation loop, speed considerations become central for many research projects where the code may take weeks to run.

We report the results of this speed comparison in Table 1, where we show the average run time and the performance of each code relative to the best performer in each group (`C++` with `GCC` in the `Mac` machine and `C++` with `Visual C++` in the `Windows` machine). For those codes that run in less than 60 seconds, we average 10 runs (after a warm-up) to smooth out small differences caused by the operating system. In the codes that run in more than 60 seconds, we report only one run, as any small difference does not have a material effect on relative performance. Also, we report the processor time consumed by the code, not the watch time, except for `R`, where we report user time (to avoid the problems of the overhead of the `REPL` shell).[12] At the bottom of the table, separated by a double line, we report the hybrid and special cases: `Matlab` with `Mex` files, `R` with `Rcpp`, `Numba`, `Cython`, and the rewrite of `Mathematica`.

Our first result is that `C++` and `Fortran` still have a considerable speed advantage with respect to all other alternatives. Second, `C++` and `Fortran` codes execute in roughly the same time. Third, even for our very simple code, there are noticeable differences among compilers. We find speed improvements of more than 100 percent between different executables of the same underlying code (and using equivalent compilation flags). While the open-source `GCC` compilers are superior in a `Mac` environment relative to the Intel compilers, `GCC` compilers do less well in a `Windows` machine.[13] The deterioration in performance of the `Clang` compiler was expected given that the goal of the `LLVM` behind it is to minimize compilation time and executable file sizes, both important goals when developing general-use applications but often (but not always!) less relevant for numerical computation.

Fourth, `Java` takes between 2.1 to 2.69 times longer to execute than `C++`. This difference in speed plus Java's issues with floating point arithmetic in high-performance scientific computa-

[12]`REPL` stands for `read--eval--print loop`, the interactive language shell that many users are familiar with. The details of each machine and the compilation instructions are reported in the online appendix.

[13]We thank an anonymous referee for pointing out to us that, in a `Linux` machine, Intel compilers also deliver better performance than the `GCC` compiler.

tion suggests that there is no obvious advantage for choosing `Java` over `C++` unless portability across platforms or the wide availability of `Java` programmers is an important factor.

Fifth, turning to scripting languages, `Julia`, with its just-in-time compiler, delivers an outstanding performance. The `Julia` code takes only between 2.37 and 2.62 times longer to execute than the `C++` code. `Matlab` takes between 9 to 11 times longer to execute than the best `C++` executable. The difference in performance between compiled languages and this widely used scripting language seems to have stabilized over the last decade. In the `Pypy` implementation, the `Python` code takes around 44-45 times longer to execute than in `C++`. In the "traditional" implementation of `Python` (often called `CPython`), the code takes between 155 and 269 times longer to execute than in `C++`.[14]

`R` takes between 475 to 491 times longer to execute than `C++`, although the performance improves somewhat (to between 244 and 281 times longer to execute) if the `R` code is compiled using the `R compiler package`. This poor performance is well-understood in the `R` community and it is due, in part, to some choices in the original design of `R` back in the 1990s, when nobody could have forecasted its future success. In fact, there are a number of initiatives to increase `R`'s speed, such as `pqR`, `Renjin`, and `Riposte`.[15] `Mathematica`, in its imperative version, takes up to 809 times longer to execute than `C++`.

We move now to analyzing the hybrid and special cases. When we use a `Mex` file written in `C++`, `Matlab` takes 1.29 and 1.64 times longer to execute than `C++`. When we use `Rcpp` in `R`, the resulting code takes between 3.66 and 5.41 times longer to execute than `C++`.

In the `Python` world, `Numba`'s decorated code takes between 1.57 and 1.62 times longer to execute than the best `C++` executable and `Cython` code takes between 1.41 and 2.49 times longer to execute than `C++`. Both approaches demonstrate a great performance.

`Mathematica` is a particular case. If we just use the function `Compile` to compile the Bellman operator, performance improves, but not dramatically. If, instead, we both rewrite the code to have a more functionally oriented structure and use the function `Compile`, `Mathematica` takes between 1.67 and 2.22 times longer to execute than `C++`. As we mentioned before, we do not emphasize this performance, as the code was tuned to `Mathematica` requirements, something we did not do for other languages.

[14]Other benchmarks have also found similar results. For example, the *Computer Languages Benchmark Game* finds many examples where `Python` takes over 100 times longer to execute than `C++`. See also Lubin and Dunning (2013), `https://modelingguru.nasa.gov/docs/DOC-1762`, or `http://wiki.scipy.org/PerformancePython`.

[15]See the discussions and speed tests in `http://www.pqr-project.org/`, `http://www.renjin.org/`, and `https://github.com/jtalbot/riposte`.

4.2. Coding Comparison

Our previous results have focused on execution speed. However, in many cases, the real constraint is not execution time, but how long it takes the economist to code the algorithm he is interested in. A researcher may want to code in a slower language if the difference in coding productivity compensates for the difference in execution speed. One objective measure of complexity, however incomplete, is the length of the code. Table 2 reports the lines of code required by each program (for `Mex` and `Rcpp` we report the lines of code of the two programs, the main program in `Matlab` or `R` and the inner loop program in `C++`). The number of lines is very similar across languages, except for the compiled codes, which need to pay an overhead in terms of type declarations.

Unfortunately, comparing coding complexity (except for the number of lines) is subjective and depends on the familiarity of a researcher with a particular programming language or perhaps just with his predisposition toward a programming paradigm. Nevertheless, and with all due caution, we find it of interest to highlight a few factors.

A first important difference, which we already pointed out in section 3, is whether the language is compiled (such as `C++` or `Fortran`) or scripted (such as `Matlab` or `Julia`). Compiled programs pay for their superior speed with the overhead of having to go through the edit-compile-run-debug cycle to test it and to identify potential problems. Scripted languages allow for a more interactive coding process where the programmer can easily test small chunks of the code through the `REPL`. Especially for less experienced programmers, this latter form of exploratory programming can be more attractive.[16]

A second difference is the type system. `C++` and `Fortran` impose static typing, i.e., the type of the variables, functions, and other code components is verified and enforced at compile-time. This type needs to be manifest, as in `Fortran`, of it can be deduced, as in `C++14`. Type deduction makes the code more adaptable and less tedious to read and write (although also more open to subtle mistakes). Other languages, such as `Matlab` and `Julia`, are dynamically typed, i.e., the type is verified and enforced at run-time. Also, the type of the variable is inferred (although type annotations are possible). Static compilation usually makes coding the first version of the program slower and less interactive, but also safer. Dynamic and/or inferred typing is the source of numerous bugs. Finding and fixing those bugs may end up taking longer than being explicit about typing from the start.

A third difference is the complexity of the language. `C++` is an extremely complex language, with dozens and dozens of features. Even the most experienced programmers sometimes are not familiar with all of them. Although the core of the language is relatively simple, the

[16] There are also `REPLs` for `C++`, such as `https://root.cern.ch/drupal/content/cling`, although their use is less common and less powerful than the full-fledged language.

daunting amount of information required to code advanced C++ can be challenging. On the other hand, Fortran, as we mentioned above, is simple and compact, although also rather limited for implementing advanced programming techniques. Matlab and R are somewhere in the middle. While their core is also relatively straightforward, the presence of dozens of toolboxes for Matlab and of thousands of packages for R can overwhelm some young researchers. Many observers have commented, also, on the steep learning curve of R. Python is extremely intuitive and easy to learn, as proven by the fact that it is widely used as the language of choice in introductory classes in computer science. Finally, Mathematica can be idiosyncratic and difficult to master for those programmers coming from backgrounds in other languages, but it is also extremely elegant once one familiarizes oneself with its structure.

A fourth difference is the set of tools available for a programmer and how active the community of users in that language is. For example, a good IDE or a powerful debugger is key for fast development. Given our choice of languages, the programmer will find outstanding IDEs, debuggers, or profilers for most of the languages. For less experienced programmers, for instance, the IDE of Matlab or RStudio for R are easy to use and intuitive and include both debuggers and profilers of high quality. The only partial exception is Julia, which suffers from its early stage of development and the absence of a good IDE. With respect to the user communities, Fortran has the important disadvantage of being a niche language. Matlab, Python, and R have extremely active communities in scientific computation. Perhaps the main strength of R is in statistics and econometrics where the extraordinary richness of existing packages (over 6,560 at the CRAN repository as of April 2015) makes it an outstanding alternative. Similarly, a key advantage of Python is the existence of libraries such as Numpy, Scipy, SymPy, MatPlotLib, and pandas and of shells such as IPython (although some of these libraries have only been partially ported to Python 3+).

Finally, one must always remember that different programming languages can be used by one researcher to address different problems (for example, a complicated value function iteration in C++ and a statistical analysis of some data in R) or even mixed with care in the same project.

4.3. Some Additional Remarks

We close with several additional remarks about our exercise. First, to make the comparison as unbiased as possible, we coded the same algorithm in each language without adapting it to the peculiarities of each language (which could reflect more about our knowledge of each language than of its objective virtues).[17] Therefore, the final code looks remarkably similar

[17] It also means that proposals to improve the coding should be made for all languages (unless there is an obvious problem with one of the languages). The game is not to write the best possible C++ code, but to write

among the languages, with the exception of one version of the code in `Mathematica`.

Second, we do not take advantage of the particular features of each language. For a simple algorithm such as ours, these adaptations will not make much difference, but they would make the comparison extremely cumbersome.[18]

Third, our computational task (value function iteration, monotonicity in the decision rule, and an envelope condition) is not well-suited to vectorization. The argument is explained in detail in the online appendix. In particular, the nesting of a `while` loop with three `for` loops and an `if` control statement, far from being a poor programming choice, saves considerable time in execution. We have vectorized versions of our code in `Matlab` or `R` (languages that often profit from vectorization) that take longer to execute than the baseline codes. Furthermore, the deterioration in performance becomes worse as the number of grid points increases.

Fourth, and also beyond the scope of this paper, we do not compare how easy it is to parallelize the code written in each language. This may be an important factor with some languages, such as `Julia`, that are designed from scratch to be easy to parallelize, and others that have more issues with it (for example, `Python` due to its Global Interpreter Lock that synchronizes the execution of threads).

Finally, and somewhat speculatively, how would our results change in other, more involved projects in macroeconomics, such as models with heterogeneous agents or in estimation? While we cannot offer anything more than educated guesses, our experience over the years is that the relative magnitudes of speed and coding comparisons survive surprisingly well across many applications. For example, in a typical estimation by simulation (using a particle filter or Markov Chain Monte Carlo), `C++` still has a small speed advantage over `Fortran`, `Julia` takes around 2-3 times longer to run than `C++` or `Fortran` code, while `Matlab` takes around 10 times longer, and `Python` and `R` take even much longer (these two languages have a wide variance in the degree of their speed deterioration from application to application). All of these results are very much in line with our findings in this paper. We emphasize simulation-based estimation methods because their algorithmic structure is sufficiently far away from value function iteration as to be an informative alternative. However, many of the tasks highlighted in our benchmark (loops, large vector and matrices, etc.) are the same when we estimate by simulation or even when we solve problems with heterogeneous agents. Interestingly enough,

`C++` code that is comparable to, for example, `Matlab` code in computational complexity. We are not interested in speed itself, but on *relative* speed.

[18]For example, we do not change the order or the iteration of the loops (which is row-major) to suit the column-major structure of `C++` (the other languages are either row-major or the code is not affected by this change). In additional testing, we documented that rearranging the loops leads to an improvement of time of 9-10% in `C++`. We do not emphasize this difference because `C++` was already the fastest language. Also, improving the speed for the `C++` code only changes the unit that we are using to measure all the other languages, with all the other relative comparisons left unchanged.

these simulation-based estimation methods are also not very well-suited for vectorization and, therefore, loop speed is still key. For these more complex tasks, `Matlab`, `Python`, or `R` become just too slow and the researcher definitively needs to move to `C++` or `Fortran` or at least to some of the hybrid approaches we discussed. Similar statements in terms of relative speed hold for non-linear solution methods for DSGE models such as projection methods.

5. Concluding Remarks

In this short paper we have taken a first step at a comparison of programming languages in macroeconomics. We have offered results on execution speed and some, more subjective, comments on coding complexity. Our simple exercise leaves many questions unanswered. For example: How do our results extend to other problems, such as those in econometrics? Are there improvements in our algorithm that would benefit one programming language much more than others? Can we re-arrange loops in ways that change relative speeds? However, our results in and of themselves should be of interest to a wide audience of researchers. We hope to see more comparisons of programming languages in economics in the future and a discussion of our coding choices through our `github` repository, where readers can fork their own versions of our programs.

References

[1] Amador, M., S.B. Aruoba, J. Fernández-Villaverde (2014). "Functional Programming in Economics." Mimeo in preparation.

[2] Aruoba, S.B., J. Fernández-Villaverde, and J. Rubio-Ramírez (2006). "Comparing Solution Methods for Dynamic Equilibrium Economies." *Journal of Economic Dynamics and Control* 30, 2477-2508.

[3] Lubin, M. I, Dunning (2013). "Computing in Operations Research using Julia." *Mimeo*, MIT.

[4] Prechelt, L. (2000). "An Empirical Comparison of Seven Programming Languages." *IEEE Computer* 33(10), 23-29.

[5] Sargent, T. and J. Stachurski (2014). *Quantitative Economics.* Mimeo, `http://quant-econ.net/_static/pdfs/quant-econ.pdf`.

[6] Tauchen, G. (1986), "Finite State Markov-chain Approximations to Univariate and Vector Autoregressions." *Economics Letters* 20, 177-181.

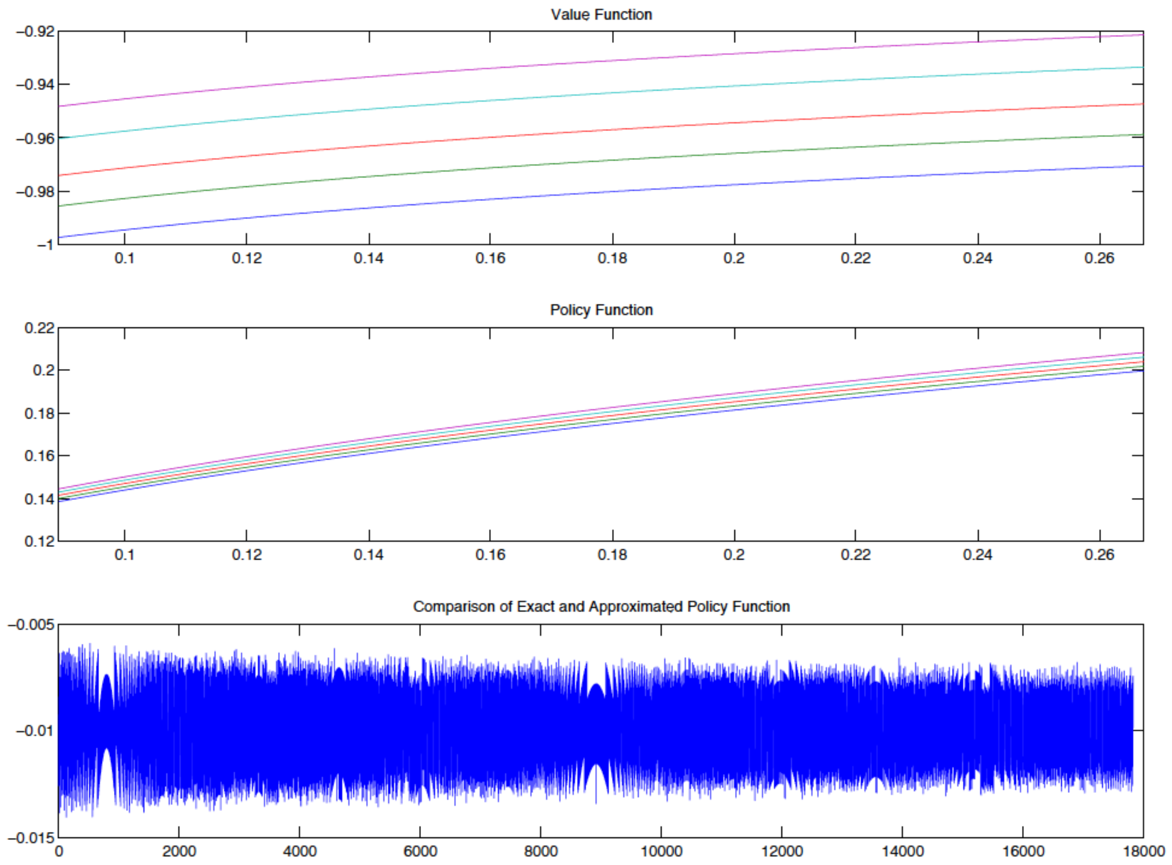

Figure 1: Value function, policy function for capital, and relative difference between exact and approximated solution

Table 1: Average and Relative Run Time (Seconds)

	Mac			Windows		
Language	Version/Compiler	Time	Rel. Time	Version/Compiler	Time	Rel. Time
C++	GCC-4.9.0	0.73	1.00	Visual C++ 2010	0.76	1.00
	Intel C++ 14.0.3	1.00	1.38	Intel C++ 14.0.2	0.90	1.19
	Clang 5.1	1.00	1.38	GCC-4.8.2	1.73	2.29
Fortran	GCC-4.9.0	0.76	1.05	GCC-4.8.1	1.73	2.29
	Intel Fortran 14.0.3	0.95	1.30	Intel Fortran 14.0.2	0.81	1.07
Java	JDK8u5	1.95	2.69	JDK8u5	1.59	2.10
Julia	0.3.7	1.91	2.62	0.3.7	1.80	2.37
Matlab	2014a	7.91	10.88	2014a	6.74	8.92
Python	Pypy 2.2.1	31.90	43.86	Pypy 2.2.1	34.14	45.16
	CPython 2.7.6	195.87	269.31	CPython 2.7.4	117.40	155.31
R	3.1.1, compiled	204.34	280.90	3.1.1, compiled	184.16	243.63
	3.1.1, script	345.55	475.10	3.1.1, script	371.40	491.33
Mathematica	9.0, base	588.57	809.22	9.0, base	473.34	626.19
Matlab, Mex	2014a	1.19	1.64	2014a	0.98	1.29
Rcpp	3.1.1	2.66	3.66	3.1.1	4.09	5.41
Python	Numba 0.13	1.18	1.62	Numba 0.13	1.19	1.57
	Cython	1.03	1.41	Cython	1.88	2.49
Mathematica	9.0, idiomatic	1.67	2.29	9.0, idiomatic	2.22	2.93

Table 2: Lines of Code

Language	Lines
C++	182
Fortran	149
Java	160
Julia	106
Matlab	114
Python	119
R	108
R compiled	118
Mathematica	114
Matlab, Mex	108+127
Rcpp	86+52
Python, Numba 0.13	135
Cython	104
Mathematica	130

6. Online Appendix (Not for Publication)

In this online appendix we report our compilation flags and we explain why vectorization is unlikely to be a good approach to solve our problem.

6.1. Compilation Flags

Our `Mac` machine had an Intel Core i7 @2.3 GHz processor, with 4 physical cores, and 16 GB of RAM. It ran OSX 10.9.2. Our `Windows` machine had an Intel Core i7-3770 CPU @3.40GHz processor, with 4 physical cores, hyperthreading, and 12 GB of RAM. It ran Windows 7, Ultimate-SP1.

The compilation flags were:

1. GCC compiler (Mac): `g++ -o testc -O3 RBC_CPP.cpp`
2. GCC compiler (Windows): `g++ -Wl,--stack,4000000, -o testc -O3 RBC_CPP.cpp`
3. Clang compiler: `clang++ -o testclang -O3 RBC_CPP.cpp`
4. Intel compiler: `icpc -o testc -O3 RBC_CPP.cpp`
5. Visual C: `cl /F 4000000 /o testvcpp /O2 RBC_CPP.cpp`
6. GCC compiler: `gfortran -o testf -O3 RBC_F90.f90`
7. Intel compiler: `ifortran -o testf -O3 RBC_F90.f90`
8. `javac RBC_Java.java` and run as `java RBC_Java -XX:+AggressiveOpts`.

6.2. Vectorization and the Properties of the Solution

Our paper uses value function iteration. We take a value function $V^{n-1}(k,z)$, we apply the Bellman operator:

$$V^n(k,z) = \max_{k'} (1-\beta)\beta^t \log(zk^\alpha - k') + \beta \mathbb{E}\left[V^{n-1}(k',z')\,|z\right]$$

and we get a new value function $V^n(k,z)$. Following standard arguments, one can ensure that for any initial $V^0(k,z)$, $V^n(k,z) \to V(k,z)$ as $n \to \infty$ in the sup norm.

There are two computational costs in value function iteration. First, we need to evaluate the operator for any value of k and z. In the paper, we have 17,820 points in the grid of capital (k_i, for $i \in \{1,\ldots,17,820\}$) and 5 points in the grid of productivity (z, for $i \in \{1,\ldots,5\}$), for a total of 89,100 points. Second, we need to solve the $\max_{k'}$ problem. To make our problem as transparent as possible, by imposing that k' belongs to the grid of capital. That is, for each of the 89.100 points, we need to search among the 17,820 possible choices of k'_m, for $m \in \{1, \ldots, 17,820\}$ (given our choice of capital grid, all the choices of k' are feasible for any point in the state space).

To ease this computational burden, we take advantage of two properties of the solution. First, the monotonicity of the decision rule. That is, if we know that for state variables k_i and z_j, the optimal choice is $k'_m = g(k_i, z_j)$, then we know that $k'_n = g(k_{i+1}, z_j) \geq k'_m = g(k_i, z_j)$. The decision rule is also monotone along the productivity dimension (although our algorithm does not exploit monotonicity along the second dimension: we found in preliminary testing that the improvements in speed from doing so were minimal).

Second, we know that an *envelope condition* applies. More concretely, if

$$\begin{gathered}(1-\beta)\beta^t \log\left(z_j k_i^\alpha - k'_{n+1}\right) + \beta \mathbb{E}\left[V^{n-1}\left(k'_{n+1}, z'\right)|z\right] \\ < (1-\beta)\beta^t \log\left(z_j k_i^\alpha - k'_n\right) + \beta \mathbb{E}\left[V^{n-1}\left(k'_n, z'\right)|z\right]\end{gathered}$$

then we know that the optimal choice of k' cannot be higher than k'_n, or $k'_n = g(k_i, z_j) < k'_{n+1}$. In other words, once we have reached the optimal choice of k', higher future capital only decreases the value function.

These two properties allow us to write a particularly efficient algorithm. We copy the code here from the `Matlab` version, but all the other codes are nearly identical:

```
------------------------------------------
for nCapitalNextPeriod = gridCapitalNextPeriod:nGridCapital
    consumption = mOutput(nCapital,nProductivity)-vGridCapital(nCapitalNextPeriod);
    valueProvisional = (1-bbeta)*log(consumption)...
+bbeta*expectedValueFunction(nCapitalNextPeriod,nProductivity);
    if (valueProvisional>valueHighSoFar)
        valueHighSoFar = valueProvisional;
        capitalChoice = vGridCapital(nCapitalNextPeriod);
        gridCapitalNextPeriod = nCapitalNextPeriod;
    else
        break; % We break when we have achieved the max
    end
end
------------------------------------------
```

What are we doing here? The counter `nCapitalNextPeriod` will search over optimal values of k' given some values k_i and z_j. But we do not initialize the counter at 1 (except with the first point of the grid). We initialize the counter at `gridCapitalNextPeriod`, that is, the optimal choice of capital in the previous point of the grid k_{i-1} and z_j. Then, we evaluate consumption and the value function for the choice of k' capital given by the counter. If the choice improves the previous evaluation of the value function (`valueProvisional>valueHighSoFar`), we move to the next point of the grid. Otherwise, we break the search since we have already found our optimal choice.

Therefore, we will only need to check a few points in the vector `nCapitalNextPeriod`. For example, in the last iteration of the value function (iteration 257), we search on average 2.65 points of k' for each value k_i and z_j, instead of 17,820 (as a naive search would require). In other words, instead of having to evaluate the value function 1.5878e+09 times in iteration 257, we only evaluate it 235,656 times. The average number of searches is very stable from iteration 1 and already at iteration 9 it has settled down at 2.65 points.

A cursory inspection of our code, where we nest a `while` loop with three `for` loops and an `if` control statement, could suggest the possibility of improving performance by taking advantage of vectorization. Unfortunately, our algorithm is unlikely to benefit from vectorization. The reason is that vectorization cannot easily accommodate monotonicity and the envelope condition. We could, for example, evaluate the value function for

$$(1-\beta)\beta^t \log\left(z_i k_j^\alpha - k'\right) + \beta \mathbb{E}\left[V^{n-1}\left(k', z'\right)|z\right]$$

for all possible values of k' using vectorized functions and obtain a vector $\widehat{V}\left(k'\right)$. But then we would need to search $\widehat{V}\left(k'\right)$ for its maximum value, a costly task. Monotonicity and the envelope condition tell us that we do not need to compute the whole of $\widehat{V}\left(k'\right)$, just 2.65 points on average. Furthermore, as the number of grid points k' increases, the performance of vectorization deteriorates more and more because we need to search among more values of k'. Note that this is not directly linked to the efficiency with which we store matrices in memory. Our inner `for` loops and the `if` control statement, while not very elegant, exploit the mathematical structure of our problem. The two outer `for` loops are just convenient ways to move along the state space without the need to store large matrices. Similarly, the `while` loop is unavoidable: the very core of value function iteration is that it is an `iteration`.

We can illustrate the drawbacks of vectorization running `Matlab` and `R` code with and without vectorization and for three different grids for capital: 179, 1782, and 17,820 grid points. Table A.1 reports the running time in seconds. Our original code is substantially faster than a vectorized version and that the deterioration in performance increases with the number of grid points.

Our vectorization was very simple: we eliminated the inner loop and the conditional statement and replaced them with vector operations on the whole grid of k'. While a more carefully designed vectorization could reduce the distance with respect to our loop code, vectorization would always need to beat a very efficient loop search that uses monotonicity and an envelope condition. As a result, we are not sanguine about the chances of vectorization for our problem.

Table A.1: Loop vs. Vectorization

k'	179	1782	17820	k'	179	1782	17820
`Matlab`, loop	0.09	0.80	7.91	`R`, loop	2.14	22.00	345.55
`Matlab`, vectorized	1.51	76.04	3408.18	`R`, vectorized	2.29	108.72	10785.20