A Comparison of Programming Languages in Economics[*]

S. Borağan Aruoba[†] University of Maryland

Jesús Fernández-Villaverde[‡] University of Pennsylvania

June 16, 2014

Abstract

We solve the stochastic neoclassical growth model, the workhorse of modern macroeconomics, using `C++11`, `Fortran 2008`, `Java`, `Julia`, `Python`, `Matlab`, `Mathematica`, and `R`. We implement the same algorithm, value function iteration with grid search, in each of the languages. We report the execution times of the codes in a `Mac` and in a `Windows` computer and comment on the strength and weakness of each language.

Key words: Dynamic Equilibrium Economies, Computational Methods, Programming Languages.

JEL classifications: C63, C68, E37.

[*]We thank Manuel Amador for his help with making our `Python` and `Mathematica` codes more idiomatic, Matthew MacKay and John Stachurski for their incredible help with Numba, Florian Oswald for pointing out the importance of optional typing in `Julia`, Luigi Bocola, Gustavo Camilo and Pablo Cuba-Borda for research assistance, Pau Rabanal and Thibaut Lamadon for 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 experimentations or on "folk wisdom." For example, it is still commonly believed that `Fortran` is the fastest available language or that `C++` is too hard to learn given its potential advantages.

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 searching for guideposts in their first incursions into computation.

We solve the stochastic neoclassical growth model, the workhorse of modern macroeconomics, using `C++11`, `Fortran 2008`, `Java`, `Julia`, `Python`, `Matlab`, `Mathematica`, and `R`.[2] We implement the same algorithm, value function iteration with grid search for optimal future capital, in each of the languages[3] and measure the execution time of the codes in a `Mac` and in a `Windows` computer. The advantage of our algorithm, value function iteration with grid search, 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 the two most popular environments for software development for economists (in the `Mac` we compile and run some of the code from the command line, thus implying results very close to those that would come from equivalent `Unix/Linux` machines).

In section 4, we report speed results for each language (including several implementations

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

[2]From now on, we drop the year of the standard of the language unless it is needed.

[3]For `Python` and `Mathematica` we have two versions of the code: one with exactly the same algorithm than for the other languages and one with some variations. These two versions will illustrate the importance of idiomatic programming. The change in `Python` is small, but bigger in `Mathematica`.

of the same language and different compilers), but here is a brief summary of some of our main findings:

1. `C++` and `Fortran` are still considerably faster than any other alternative, although one needs to be careful with the choice of compiler.

2. `C++` compilers have advanced enough that, contrary to the situation in the 1990s and some folk wisdom, `C++` code runs slightly faster (5-7 percent) than `Fortran` code.

3. `Julia`, with its just-in-time compiler, delivers outstanding performance. Execution speed is only between 2.64 and 2.70 times the execution speed of the best `C++` compiler.

4. Baseline `Python` was slow. In the `Pypy` implementation, it runs around 44 times slower than in `C++`. In the default `CPython` interpreter, the code runs between 155 and 269 times slower than in `C++`.

5. However, a relatively small rewriting of the code and the use of `Numba` (a just-in-time compiler for `Python` that uses decorators) dramatically improves `Python`'s performance: the decorated code runs only between 1.57 and 1.62 times slower than the best `C++` executable.

6. `Matlab` is between 9 to 11 times slower than the best `C++` executable. When combined with `Mex` files, though, the difference is only 1.24 to 1.64 times.

7. `R` runs between 500 to 700 times slower than `C++`. If the code is compiled, the code is only between 240 to 340 times slower.

8. `Mathematica` can deliver excellent speed, about four times slower than `C++`, but only after a considerable rewriting of the code to take advantage of the peculiarities of the language. The baseline version our algorithm in `Mathematica` is much slower, even after taking advantage of `Mathematica` compilation.

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 of weakness. It is a validation that our exercise was conducted under reasonably fair conditions. Our goal is not 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 do not comment on the difficulty of implementation of the algorithm in each language, for a couple of reasons. First, such difficulty 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.[4] Second, 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).[5] Therefore, the final result is code that looks remarkably similar among languages. However, all our codes are posted online at our `github` repository `https://github.com/jesusfv/Comparison-Programming-Languages-Economics` and the reader is invited to gauge that difficulty for himself. The main point of this paper is provide a measure of the "benefit" in a cost-benefit calculation for researchers who are considering learning a new language. The "cost" part will be subjective.

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

We pick, for our comparison exercise, the basic stochastic neoclassical growth model, the foundation of much work in modern macroeconomics. We solve the model with value function iteration and a grid search for the optimal values of future capital. In that way, we compare programming languages for their ability to efficiently 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

[4]How does one compare the programming ease of a dynamicaly typed language with the strength of a statically typed one?

[5]It also means that proposals to improve the coding should be done for all languages at the same time (unless there is an obvious problem with one of the languages). The game is not to write the best possible `C++` code, it is to write `C++` code that is comparable to, for example, `Matlab` code in computational complexity. We are not interested in speed itself, but on *relative* speed.

the interest of space, we have directly 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 chose 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 a more challenging computation. While our choice of parameters may affect the accuracy of the solution, the relative speed comparisons that we report below are robust to different calibrated 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 is:

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

(where we have already imposed that $\delta = 1$). We solve this Bellman operator using value function iteration and grid search on k'. We take advantage of monotonicity in the policy function and the concavity of the value function to avoid unnecessary computations. We use a grid of 17,820 points for k uniformly distributed ± 50 percent of the steady-state value of capital.

We found that smaller grids cause a problem for our comparison since `C++` or `Fortran` would solve the problem in such a short fraction of time that all the computation times are subject to large relative measurement error (due to issues such as the situation of the cache at any given time). Thus, we choose our grid size to be large enough so that `C++` or `Fortran` would solve the problem in about one second. More grid points are particularly bad for languages such as `Python` or `R`, which face speed penalties handling large vectors. We impose a tolerance of 1.0e-07 for convergence. The value function took 257 iterations to converge. We checked that all codes achieved convergence in the same number of iterations and that the computed value and policy functions were exactly the same.

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. 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 face the need to choose among several 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. For instance, we pick `C++` instead of `C`, but given that we do not use specific `C++` features such as objects or meta-programming, the `C` code would look nearly the same and run in nearly exactly the same time. 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++`, `Fortran`, and `Java`. `C++` is, perhaps, the most powerful language available among those widely used. Together with `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 and `C++` number 4, with a total rating of 22.91 percent.[6] Two very close relatives, `Objective-C` and `C#`, are also widely used in the industry (`Objective-C` is ranked 3 and `C#` is ranked 6 in the

[6] The interpretation and computation of these percentages can be found at `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`.

TIOBE Index. Including all four languages, the C family accumulates a popularity index of 38.45 percent). However, design considerations in `Objective-C` and `C#` that make them attractive for commercial applications also render them slower for numerical computation and, thus, they are rarely employed for the tasks we are concerned with in this paper.[7]

`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 intriguing innovations such as coarrays. Reflecting this niche nature of `Fortran`, the TIOBE ranks it 32, with a 0.419 percent popularity.

`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 for development. In the TIOBE Index, it is ranked 2 with a popularity of `5.99` percent.

The performance of compiled languages also depends on the compiler used to generate the executable files.[8] 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`.[9] For `Java`, we select the standard `Oracle JDK`.

3.2. Scripting Languages

We pick as our scripting languages `Matlab`, `Python`, `Julia`, `Mathematica`, and `R`. `Matlab`, `Mathematica`, and `R` are sufficiently known among economists that it is not necessary to elaborate on our choice. For `Matlab`, we looked at both standard `Matlab` and at the use of `Mex` files, where part of the code is written in `C`. We did not check `Octave`, an open source clone of `Matlab`, because it is well-known to be considerably slower than `Matlab`.

`Python` is an elegant and simple open-source language that has become hugely popular in the scientific community (see Sargent and Stachurski, 2014), in particular the 2.7 version.[10] Since there are different implementations of `Python`, we select:

[7] For a comparison of syntaxes, see the Hyperpolyglot at `http://hyperpolyglot.org/cpp`. Another close relative, `D`, which generates code usually roughly of the same speed as `C++`, is much less popular (ranked 26 with 0.593 percent). The recently released `Swift`, a descendent of `Objective-C`, is not designed, at this moment, as a programming language for use outside `OS X` and `iOS`.

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

[9] 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.

[10] A key advantage of `Python` is the existence of a variety of libraries such as `Numpy`, `Scipy`, `SymPy`, `MatPlotLib`, `pandas` and of interactive shells such as `IPython`. Some of these libraries have only been partially ported to `Python 3+`.

1. CPython, the default Python interpreter that comes, for example, with Mac and Linux machines.

2. Pypy (http://pypy.org/), a speed-oriented replacement virtual machine that uses a just-in-time compiler.

3. Numba (http://numba.pydata.org/), another just-in-time compiler that uses decorates to compiles Python to LLVM.

We did not explore Cython (http://cython.org/), a superset of the Python language that supports important C programming features (such as calling C functions and declaring C types) because the most recent releases of Numba are capable of delivering similar speed-ups at a fraction of the cost in terms of additional programming effort.[11]

Finally, Julia (http://julialang.org/) is a new open-source high-performance programming language with a syntax very close to Matlab, Lisp-style macros and many other modern programming features, and it also uses a just-in-time compiler for speed based on the LLVM. Three particularly attractive features of Julia are:

1. Julia's default typing system is dynamic (to facilitate fast coding), but it is possible to indicate the type of certain values to accommodate the need for speed.

2. Julia can call C or Fortran functions without wrappers or APIs.

3. Julia has a library to imports Python modules and provides wrappers for all of the functions on them.

3.3. Functional Programming Languages

The big missing items in our list of languages are those that belong to the functional programming family that inherit 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 Ocalm or Haskell. Since this comparison involves a number of issues of its own, we prefer to avoid those here to keep the paper focused.[12]

[11] The Python ecosystem is increadibly 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/.

[12] 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 surprisinaly, 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.

4. Results

We report our results in table 1, where we report the average run time and the relative performance of each code in terms of 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 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 the small differences caused by the operating system do not have a material effect on relative performance. Also, we report elapsed time, not watch time, except for `R`, where we report user time (to avoid the problems of the overhead of the REPL shell).[13] At the bottom of the table we report the three cases where we departed from the standard algorithm: `Python` with `Numba`, where we wrap the inner loop of our grid search as a function to be compiled just-in-time, `Matlab` with `Mex` files, and a rewrite of the algorithm to adapt it to the strengths of `Mathematica`.

Our first result is that `C++` and `Fortran` still maintain a considerable speed advantage with respect to all other alternatives. For example, these compiled languages are between 2.10 and 2.69 times faster than `Java`, around 10 times faster than `Matlab`, and around 48 times faster than the `Pypy` implementation of `Python`.

Second, `C++` compilers have advanced enough that, contrary to the situation in the 1990s, `C++` code runs slightly faster (5-7 percent) than `Fortran` code. The many other strengths of `C++11` 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 other hand, `Fortran 2008` is simple and compact -and, thus, relatively easy to learn- and it can take advantage of large amounts of legacy code.[14]

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 optimization compilation flags). While the open-source `GCC` compilers are superior in a `Mac/Unix/Linux` environment (for which they have been explicitly developed) to the Intel compilers, they do less well in a `Windows` machine. 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.

Our fourth result is that `Java` imposes a speed penalty of 110 to 169 percent. Given the

[13]The details of each machine and the compilation instructions are reported in the appendix.

[14]In fact, many `Fortran` compilers nowadays are just front-end parsers for a `C` compiler, thus the slight advantange of `C++` in performance times.

similarity between `Java` and `C++` syntax, there does not seem to be an obvious advantage for choosing `Java` unless portability across platforms or the wide availability of `Java` programmers is an important factor.

`Julia`, with its just-in-time compiler, delivers an outstanding performance. Execution speed is only 2.64 to 2.70 times the execution speed of the best `C++` compiler. `Julia` is also slightly faster than `Java` and close to 4 times faster than `Matlab`. Given how close `Julia`'s syntax is to `Matlab`'s, the fact that is open-source, and that the language has been designed from scratch for easy parallelization, many researchers may want to learn more about it. However, `Julia`'s standard is still evolving (causing potential backward incompatibilities in the future) and there are few existing libraries for it at the moment.

`Matlab` runs between 9 to 11 times slower 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. When we wrap the inner loop of our grid search as a `Mex` files written in C (a `Mex` file is a dynamically linked subroutines that the `Matlab` interpreter loads and executes), execution time is between 1.29 and 1.64 times the execution time of pure `C++` code. This suggests that a researcher can use the friendly environment of `Matlab` for everyday tasks (data handling, plots, etc.) and rely on `Mex` file written in C++ for the heavy computations, especially those involving loops.

`Python` has a mixed performance. In the `Pypy` implementation, our code runs around 44-45 times slower than in `C++`.[15] In the "traditional" implementation of `Python` (often called `CPython`, which comes preinstalled in many machines), the code runs between 155 and 269 times slower than in `C++`. Other benchmarks have also found similar results. For example, the Computer Languages Benchmark Game finds many examples where `Python` is over 100 times slower then `C++`.[16]

However, when we use `Numba`, and after some rewriting of the code (we took the inner loop of the grid search and made it into a separate function for the just-in-time compiler to handle; in comparison, the code from `CPython` and `Pypy` is exactly the same), the results improved dramatically: the decorated code runs only between 1.57 and 1.62 times slower than the best `C++` executable (and ahead of `Java` or some of the worse compilers of `C++` and `Fortran`).[17] Since `Numba` is a young project, we are not sure of how general these improvements in speed are and of how difficult is to find the right decorations in more involved algorithms. However, our results are certainly most promising.

`R`'s performance is poor overall, between 500 to 700 times slower than `C++`, although

[15]Note that the code already uses `Numpy` for the matrix operations.

[16]Other comparisons point out similar differences at `https://modelingguru.nasa.gov/docs/DOC-1762` and `http://wiki.scipy.org/PerformancePython`. or Lubin and Dunning (2013).

[17]Without this rewriting of the code, `Numba` was actually slower than `Pypy`.

the performance improves somewhat (to between 240 and 340 times slower) 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`.[18] Of course, the strength of R is in statistics and econometrics where the overwhelming richness of existing packages (over 5,500 at the CRAN repository as of May 2014) makes it an outstanding alternative for those computations that are not particularly intensive in time.

Finally, `Mathematica` is a particular case. Although it allows for multiparadigm programming (including our imperative algorithm), its kernel strongly prefers a more functionally oriented approach. We implemented both versions of the code. In the functional-algorithm version, and compiling the code, `Mathematica` runs only between 3 to 4 times slower than the executable generated by the best `C++` compiler. We do not want to overemphasize, however, this excellent performance, as the code was tuned to `Mathematica` requirements, something we did not do for other languages. In the imperative version, the `Mathematica` code takes up to 809 times longer than `C++`. Results with partial compilation were somewhat better, but not by much.

We close with three caveats about our exercise. First, we did not try to take advantage of the particular features of each programming language.[19] That would make the comparison extremely cumbersome. However, issues such as avoiding loops through vectorization, which could help `Matlab` or `R`, are less important in our case. With 17,820 entries in a vector (our grid size), vectorization rarely helps much in comparison with standard loops (also, since we take advantage of monotonicity in the policy function and the concavity of the value function, loops can actually beat vectorization in many cases). Second, except for the `Mex` files in `Matlab`, we did not explore the possibility of mixing language programming such as `Rcpp` in `R`. While such alternatives are often useful (although also cumbersome to implement), a detailed analysis falls beyond the scope of this paper. Finally, and also beyond this paper, we do not compare how easy to parallelize the code written in each language. This may be an important factor, with some languages such as `Julia` being designed from scratch to be easy to parallelize and others having more issues with it (for example, in `Python` due to its Global Interpreter Lock that synchronize the execution of threads).

[18]See `http://www.pqr-project.org/`, where some speed tests are reported., `http://www.renjin.org/`, and `https://github.com/jtalbot/riposte`.

[19]For example, we discovered in some additional explorations, that `Matlab` can improve its performance by nearly 25 percent by reinitializing the arrays where we store the value and policy functions in each iteration. We checked that such improvements do not hold in other languages (as one would have expected anyway, since theoretically reinitializing the arrays only adds an extra computational burden).

5. Concluding Remarks

In this short paper we have taken a first step at a comparison of programming languages in economics. Our focus on speed should not be taken as the only important metric for language comparison. Other issues (ease of programming, existence of auxiliary tools, vibrant communities of fellow programmers) should be considered as well. Also, 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`). However, speed has the inherent advantage of being easier to measure. Furthermore, as we mentioned above, speed comparisons give us an indication of the potential benefits for researchers from mastering a new programming language.

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 rearrange loops in ways that change relative speeds? However, our results should already 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 vigorous discussion of our coding choices through our `github` repository, where the reader can start with own fork of modifications to 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.

6. Appendix

Our `Mac` machine had an Intel Core i7 @2.3 GHz processor, with 4 physical cores, and 16 GB of RAM. It rsn 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`.

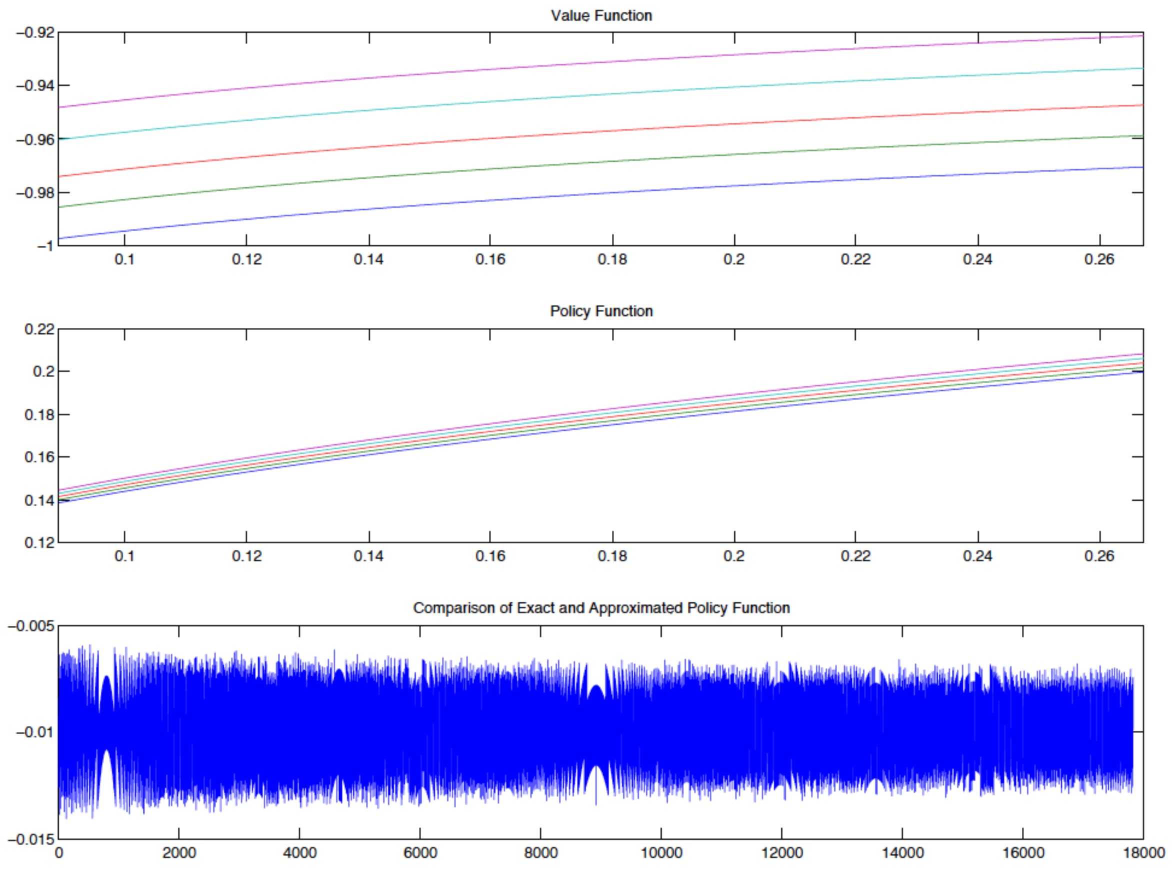

Figure 1: Value function, policy function for capital, and relative difference between exact and approximated solution

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	0.2.1	1.92	2.64	0.2.1	2.04	2.70
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	3.0.3, compiled	249.44	342.96	3.0.1, compiled	183.97	243.38
	3.0.3, script	517.53	711.55	3.0.1, script	381.80	505.09
Mathematica	9.0, base	588.57	809.22	9.0	473.34	626.19
Python	Numba 0.13	1.18	1.62	Numba 0.13	1.19	1.57
Matlab, Mex	2014a	1.19	1.64	2014a	0.98	1.29
Mathematica	9.0, idiomatic	2.26	3.11	9.0, idiomatic	2.89	3.83