A Comparison of Programming Languages in Economics*

S. Boraǧan Aruoba† University of Maryland

Jesús Fernández-Villaverde‡ University of Pennsylvania

August 5, 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 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 with 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#`, Luigi Bocola, Gustavo Camilo and Pablo Cuba-Borda for research assistance, Thomas Jones, Thibaut Lamadon, Florian Oswald, Pau Rabanal, Pablo Winant, and the members of the `Julia` user group 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.

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

[1] This also stands in comparison with work in other fields, such as Prechelt (2000), 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.

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` delivers outstanding performance. Execution speed is only between 2.64 and 2.70 times slower than the execution speed of the best `C++` compiler.

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

5. `Matlab` is between 9 to 11 times slower than the best `C++` executable.

6. `R` runs between 475 to 491 times slower than `C++`. If the code is compiled, the code is between 243 to 282 times slower.

7. Hybrid programming and special approaches can deliver considerable speed ups. For example, when combined with `Mex` files, `Matlab` is only 1.24 to 1.64 times slower than `C++` and when combined with `Rcpp`, `R` is between 3.66 and 5.41 times slower. 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.

8. `Mathematica` is only about three 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 of our algorithm in `Mathematica` is considerably slower.

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. 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 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. 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).[3] Therefore, the final code looks remarkably similar among languages, with the exception of one version of the code in `Mathematica`. The codes are posted 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 to 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 exercise, the stochastic neoclassical growth model, the foundation of much work in 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 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

[3] 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, 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.

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. 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:

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

(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 of an envelope condition 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 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 error (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 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 (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 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++`, `Fortran`, 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 rating of 34.70 percent.[4] 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.[5]

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.[6] `D`, which generates code usually roughly of the same speed as `C++`, is less popular (ranked 26 with 0.60 percent). Including all five languages, the `C` family accumulates a popularity index of 39.04 percent. `Swift`, a proposed 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

[4] See the definition and interpretation of the index at:
`http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`.

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

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

updated with modern features and innovations such as coarrays. Reflecting this niche nature of Fortran, the TIOBE ranks it 32, with a 0.42 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. 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.

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.[9] 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, 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:

1. 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.

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

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

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

We did not check `Octave`, an open source clone of `Matlab`, because it is well-known that it is considerably slower than `Matlab`. In preliminary testing, we checked that `Gauss` (`http://www.aptech.com/`) was roughly seven times slower than `Matlab` and, thus, we decided not to include it in our exercise.

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 computer-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 in the case of `Matlab` and `R`, we compile in `Python` the Bellman operator that updates the value function. We do so using two different approaches:

1. `Numba` (`http://numba.pydata.org/`), a just-in-time compiler that uses decorators to compile `Python` to `LLVM`.

2. `Cython (http://cython.org/)`, a compiler that converts type-annotated Python into generated C code that can be imported as a module.[11]

[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

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.

4. Results

We report our results in Table 1, where we show 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 (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 elapsed time, not 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, and 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 maintain a considerable speed advantage with respect to all other alternatives. For example, Java is between 2.10 and 2.69 times slower than C++, Matlab around 10 times slower, and the Pypy implementation of Python around 48 times slower.

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

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/Unix/Linux environment (for which they have been explicitly developed) to the Intel compilers, GCC compilers do less well in a Windows machine. The deterioration in performance of the Clang compiler was expected given that the goal of the

our abilities to survey every single possibility. The interested reader can check a list of compilers at http://compilers.pydata.org/.

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

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 is between 2.2 to 2.69 times slower than C++. This difference in speed plus Java's issues with floating point arithmetic in high-performance scientific computation 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.

Julia, with its just-in-time compiler, delivers an outstanding performance. Execution speed is only 2.64 to 2.70 times slower than the speed of C++. Julia is 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 it 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 only a few 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.

In the Pypy implementation, the Python code runs around 44-45 times slower than in C++. In the "traditional" implementation of Python (often called CPython), 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++.[13]

R runs between 475 to 491 times slower than C++, although the performance improves somewhat (to between 243 and 281 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.[14] 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. Mathematica, in its imperative version, runs up to 809 times slower than C++.

We move now to analyzing the hybrid and special cases. When we use a Mex file written in C++, Matlab runs 1.29 and 1.64 times slower than C++. When we use Rcpp in R, the resulting

[13] See also Lubin and Dunning (2013), https://modelingguru.nasa.gov/docs/DOC-1762, or http://wiki.scipy.org/PerformancePython.

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

code runs between 3.66 and 5.41 times slower than `C++`. 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.

In the `Python` world, `Numba`'s decorated code runs between 1.57 and 1.62 times slower than the best `C++` executable and `Cython` code runs between 1.41 and 2.49 times slower than `C++`. Both approaches demonstrate a great performance.[15]

`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` runs between 1.67 and 2.22 times slower 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.

We close with three caveats about our exercise. First, our computational task (value function iteration with grid search, monotonicity in the decision rule, and an envelope condition), is not well-suited to vectorization. The argument is explained in detail in the 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 run slower than the baseline codes. Furthermore, the deterioration in performance becomes worse as the number of grid points increases.

Second, we did not try to take advantage of the particular features of each programming language (for example, we do not change the order or the iteration of the loops between rows and columns to suit the preference of each languages; a choice that degrades, for instance, the performance of `C++`). In our personal assessment, that would make the comparison extremely cumbersome. However, the reader is invited to look at our `github` page and check how much her favorite language can improve from being coded by an expert.

Third, and also beyond 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 being easy to parallelize and others having more issues with it (for example, in `Python` due to its Global Interpreter Lock that synchronizes the execution of threads).

[15] Without compiling the Bellman equation as a separate function, `Numba` was slower than `Pypy`.

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, or 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`).

Nevertheless, speed has three inherent advantages. First, it is easier to measure. Second, speed comparisons give us an indication of the potential benefits for researchers from mastering a new programming language. Third, many real-life applications in macro 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.

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 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.

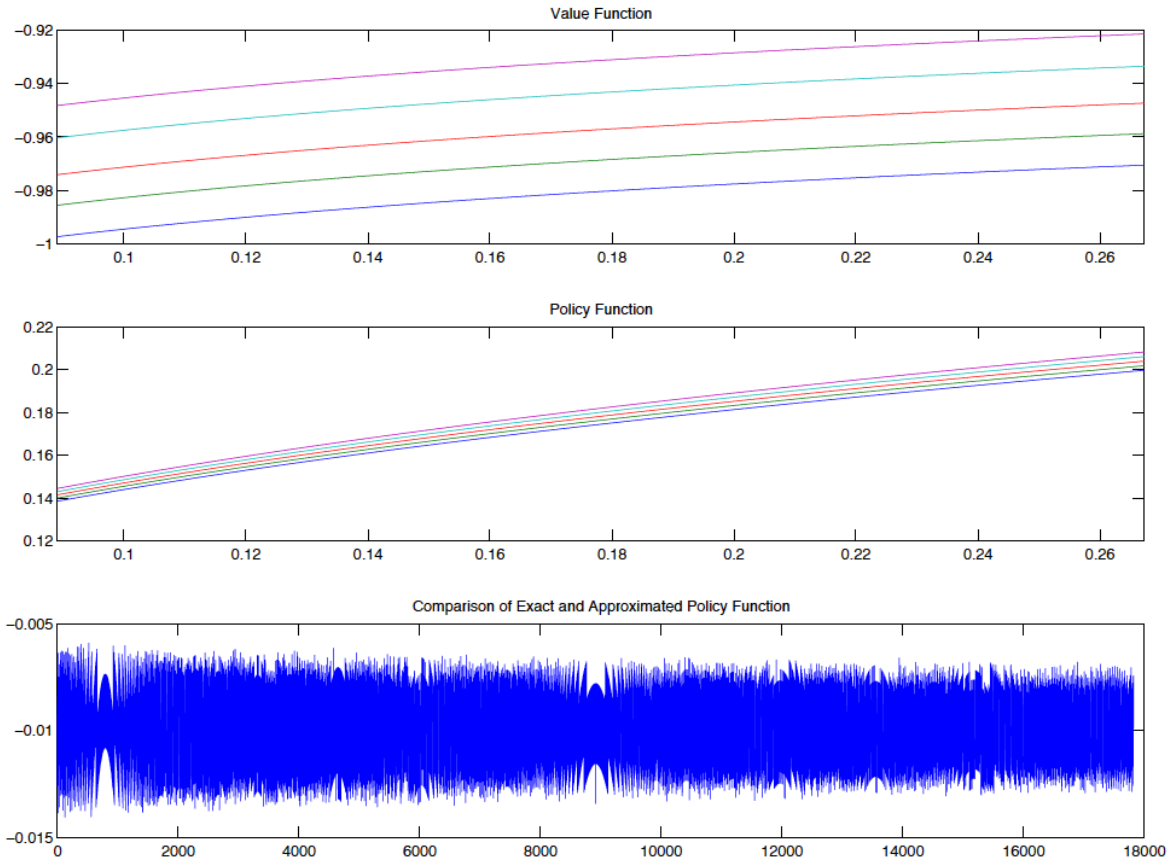

Figure 1: Value function, policy function for capital, and relative difference between exact and approximated solution

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	0.2.1	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	3.1.1, 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

6. Appendix

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

In the main paper, we use value function iteration with grid search over capital as our solution method. In particular, we take a value function $V^{n-1}(k,z)$, we apply the Bellman operator:

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

and we get a new value function $V^n(k,z)$. Using standard arguments, one can show 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 the state variables, k and z. In the main paper, we have 17820 points in the grid of capital (k_i, for $i \in \{1,\ldots,17820\}$) and 5 points in the grid of productivity (z, for $i \in \{1,\ldots,5\}$), for a total of 89100 points. Second, we need to solve the $\max_{k'}$ problem using grid search. That is, for each of the 89100 points, we need to search among the 17820 possible choices of k'_m, for $m \in \{1,\ldots,17820\}$.[16]

[16] Given our choice of capital grid, all the choices of k' are feasible for any point in the state space.

To ease the computational burden of the maximization problem, we take advantage of two key 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

$$k'_m = g\left(k_i, z_j\right),$$

then we also know that $k'_n = g\left(k_{i+1}, z_j\right) \geq k'_m = g\left(k_i, z_j\right)$. The decision rule is also monotone along the productivity dimension (although we do not exploit monotonicity along the second dimension in our algorithm: 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

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

then we know that the optimal choice of k' cannot be higher than k'_n, or $k'_n = g\left(k_i, z_j\right) < k'_{n+1}$. In other words, once we have reached the optimal choice of k', higher future capital only decreases the value function (the increase in continuation utility would be lower than the cost of a lower current consumption).

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.

A good way to see the efficiency of this algorithm is to note that 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 17820 (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 235656 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. Therefore, the inner `for` loops and the `if` control statement are the consequence of understanding 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 17820 grid

points. Table A.1 reports the running time in seconds. We can see that 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. Therefore, we are not sanguine about the chances of vectorization in this particular 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