Computational Economics
https://doi.org/10.1007/s10614-020-09983-3

Matlab, Python, Julia: What to Choose in Economics?

Chase Coleman[1] · Spencer Lyon[1] · Lilia Maliar[2] · Serguei Maliar[3]

Accepted: 8 April 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

We perform a comparison of Matlab, Python and Julia as programming languages to be used for implementing global nonlinear solution techniques. We consider two popular applications: a neoclassical growth model and a new Keynesian model. The goal of our analysis is twofold: First, it is aimed at helping researchers in economics choose the programming language that is best suited to their applications and, if needed, help them transit from one programming language to another. Second, our collections of routines can be viewed as a toolbox with a special emphasis on techniques for dealing with high dimensional economic problems. We provide the routines in the three languages for constructing random and quasi-random grids, low-cost monomial integration, various global solution methods, routines for checking the accuracy of the solutions as well as examples of parallelization. Our global solution methods are not only accurate but also fast. Solving a new Keynesian model with eight state variables only takes a few seconds, even in the presence of an active zero lower bound on nominal interest rates. This speed is important because it allows the model to be solved repeatedly as would be required for estimation.

Keywords Toolkit · Dynamic model · New Keynesian model · Global nonlinear · Low discrepancy · Quasi Monte Carlo

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10614-020-09983-3) contains supplementary material, which is available to authorized users.

✉ Chase Coleman
cc7768@gmail.com

Spencer Lyon
spencerlyon2@gmail.com

Lilia Maliar
maliarl@stanford.edu

Serguei Maliar
smaliar@scu.edu

[1] New York University, New York, USA

[2] CUNY Graduate Center and CEPR, New York, USA

[3] Santa Clara University, Santa Clara, USA

Published online: 26 June 2020

JEL Classification C6 · C61 · C63 · C68 · E31 · E52

1 Introduction

We perform a comparison of Matlab, Python and Julia as programming languages to be used for implementing global nonlinear solution techniques. We consider two popular applications: a neoclassical growth model and a new Keynesian model. Our overall experience with each language was comparable, though it is useful to recognize that each language has its specific strengths. Both Julia and Python are open-source which facilitates transparency and reproducibility. Julia outperforms both Matlab and Python in algorithms that require numerical solvers or optimizers. Python benefits from an active community and strong package ecosystem which makes finding the right tools easy. Finally, Matlab benefits from extensive documentation, technical support, and various built in toolboxes. The running times were not significantly different across three languages considered. Ultimately, the best choice of programming language depends on various factors including: model size, solution complexity, ease of writing, and co-author preferences. Regardless of the programming language chosen, our experience shows that transitioning between each of the three languages we consider should not require a substantial learning effort.

Our paper is motivated by the growing interest among economists in global nonlinear solution methods. Value function discretization is the most well-known example of a global nonlinear solution method, but there are a variety of other iterative and Euler-equation based methods that differ in the way they approximate, interpolate, integrate and construct nonlinear solutions. These methods solve economic models on grids of points that cover a relevant area of the state space, and they generally produce more accurate solutions than perturbation methods which typically build approximations around a single point.[1] Also, global methods are useful for solving some models in which perturbation methods are either not applicable or their applications are limited; see Taylor and Uhlig (1990) and Judd (1998) for reviews of the earlier methods; and see Maliar and Maliar (2014) and Fernández-Villaverde et al. (2015) for the reviews of newer literature.[2] Finally, the recent developments of numerical techniques for dealing with high-dimensional data has greatly extended the class of models that can be studied with global solution methods. In particular, it is now possible to construct global nonlinear solutions to economic

[1] The accuracy of local perturbation solutions can decrease rapidly away from steady state even in relatively smooth models; see Kollmann et al. (2011). In models with stronger nonlinearities, such as new Keynesian models, approximation errors can reach hundreds of percent under empirically relevant parameterizations; see Judd et al. (2017). In models of labor search, a global solution is important for accurate capturing even first moments of equilibrium dynamics; see Petrosky-Nadeau and Zhang (2017) .

[2] For example, perturbation methods are not well suitable for analyzing sovereign default problems (e.g., Arellano (2008) and portfolio choice problems (e.g., Hasanhodzic and Kotlikoff (2013)). There are perturbation-based methods that can deal with occasionally binding constraints but they are limited to the first-order approximation, for example, see Laseen and Svensson (2011) and Guerrieri and Iacoviello (2015).

models with hundreds and even thousands of state variables that are intractable with conventional global solution methods (such as value-function discretization) due to the curse of dimensionality.[3]

However, global nonlinear solution methods are more difficult to automate than perturbation methods, and their implementation requires more substantial programming from researchers. In particular, there is still no consensus in the profession on what software to use for implementing global solution methods, unlike for perturbation methods where a common choice is the Dynare or IRIS platforms. One important aspect of the implementation of a global solution method is the choice of programming language to use. Traditionally, Matlab is used in economics, although some researchers have also used Fortan and C. Recently, Python and Julia have begun to see a more widespread use in the economics literature.

In this paper, we pursue two goals: First, we provide a comparison between global solution methods implemented in Matlab, Python and Julia in the context of two popular applications: a neoclassical growth model and a new Keynesian model. Our goal is to help economic researchers choose the programming language that is best suited to their own situation, and, if needed, help them transition from one programming language to another. The readers can see and compare how the same algorithm is implemented in different languages and, as a result, will understand some of the unique aspects of each language and choose the one which fits their needs.[4] The implementation, structure and number of lines of code are similar across all three languages.

Second, we provide a carefully documented collection of routines for Matlab, Python and Julia that can be used for constructing global nonlinear solutions with a special emphasis on modern techniques for problems with high dimensionality. In particular, our code includes routines for constructing random and quasi-random grids, low-cost monomial integration methods, approximating functions using complete polynomials, as well as routines for computing the numerical accuracy of solutions. Much of this code is generic in a way which allows it to be easily portable to other applications including problems with kinks and occasionally binding constraints. Our examples are solved using a variety of solution techniques including: conventional policy function iteration, conventional value function iteration, an Euler equation method, the endogenous grid method of Carroll (2005), and several variants of the envelope condition method of Maliar and Maliar (2015). We also include examples of parallelization in the three languages considered.

Aruoba and Fernandez-Villaverde (2015) is closely related to our work. In their paper, the authors compare 18 different programming languages by implementing the same algorithm—value-function discretization—to solve the stochastic

[3] For example, Maliar et al. (2019) use deep learning and Google Tensorflow platform to solve a version of Krusell and Smith's (1998) model by approximating decision functions with 2000 state variables; Lepetuyk et al. (2019) use a combination of unsupervised and supervised (deep) learning to solve a large-scale projection model of the Bank of Canada, etc.

[4] Additionally, we provide a github repository with the code and notebooks with a description of our code and algorithms on the QuantEcon Notebook site. The code is licensed under the BSD-3 license and the Jupyter notebooks are released under the CC BY-ND 4.0 International license.

neoclassical growth model and by reporting the CPU time needed to solve the model. They find that the choice of a programming language plays an important role in computation time: the fastest language in their study (C++) solves the model over 450 times faster than the slowest language (R).

The present work extends Aruoba and Fernandez-Villaverde (2015) in three important ways: First, our comparison is more representative of and relevant to the modern numerical analysis in economics. To be specific, their code for value function discretization consists of arithmetic operations and loops, while our value-iterative and Euler equation methods involve also numerical integration, interpolation and regression routines, including sparse and quasi-Monte Carlo grids and derivative free solvers tractable in problems with high dimensionality. Second, our code is written in a way that exploits the strengths of each language considered, while Aruoba and Fernandez-Villaverde (2015) use the same implementation in all languages. In particular, we speed up the Matlab code by replacing some of the loops with vectorized computation, and we speed up Python by using the "just-in-time" compiler. As a result, we do not observe such huge differences across the three considered languages as those reported in their paper—the running times were roughly similar across the three languages considered.[5] Finally, in addition to the speed comparisons, we also discuss some key features of each language that might be useful for economists.

Furthermore, for the new Keynesian model, we modify the method proposed in Maliar et al. (2015) to operate on random and quasi-random grids covering a fixed hypercube instead of operating on the high probability area of the state space. The Matlab code developed in the present paper achieves an almost a 60-time speed up over the original code, while still producing highly accurate solutions. Our code is sufficiently fast to be used for the estimation of a moderately-large new Keynesian model with 8 state variables. It takes us just a few seconds to construct the solution, including the model with active zero lower bound on the nominal interest rate in all three languages.

Finally, we should mention an additional important factor for the choice of the programming language, which is a specific collection of packages and libraries designed for solving economic models. In fact, for those researchers who are deeply interested in a specific model, the choice between the languages may amount to the choice between the packages that are most suitable for the problem that they are trying to solve. Matlab users have access to the Dynare and IRIS perturbation software as well as a large collection of routines developed by macroeconomists over the last decades, for example, a value function iteration toolkit by Robert Kirby vfitoolkit.com; the PHACT toolbox for solving heterogenous-agent models by SeHyoun Ahn, Greg Kaplan, Benjamin Moll, Thomas Winberry and Christian Wolf github.com/

[5] In a 2018 update to the 2015 paper, Aruoba and Fernandez-Villaverde (2018) recomputed the previously reported running times and find great improvements in several languages, including Matlab, Julia, and Python—their running times are only slightly (less than two times) slower than the fastest C++ code under efficient implementation (MEX, just-in-time compilers). This finding indicates that the users of high-level languages such as Matlab, Julia, and Python do not sacrifice much of computational speed relative to the fastest low-level alternatives.

gregkaplan/phact; a collection of routines for solving large scale problems developed by Lilia Maliar and Serguei Maliar lmaliar.ws.gc.cuny.edu/codes; etc. Python users can benefit from the Heterogeneous Agents Resources and toolKit (HARK) developed by the team led by Christopher Carroll econ-ark.org. In turn, Julia users have access to the HetSol toolkit developed by Michael Reiter elaine.ihs.ac.at/~mreiter/installhetsol.txt. Some developers provide both Python and Julia versions of their software, for example, the QuantEcon site administrated by Tom Sargent and John Stachurski and the Dolo platform for constructing global solutions developed by Pablo Winant github.com/EconForge/dolo. In turn, we provide identical routines for all three languages, so that our users do not face the choice between Matlab, Python and Julia but can use the language that is the most suitable for the given problem.

The rest of the paper is organized as follows. In Sect. 2, we describe the three languages, Matlab, Python and Julia. In Sect. 3, we outline seven algorithms we implement to solve a variant of the standard neoclassical stochastic growth model. In Sect. 4, we present an algorithm for solving a medium-scale new Keynesian model with a zero lower bound on nominal interest rates and describe the results of our analysis. Finally, in Sect. 5 we conclude.

2 Programming Languages in Economics

In this section, we provide a brief description of the three programming languages discussed in this paper, Matlab, Python and Julia. We provide step-by-step installation instructions for Julia and Python along with a description of the accompanying software in a sequence of Jupyter notebooks; see QuantEcon Notebook site.

2.1 Matlab

In our experience, Matlab is the programming language that most economists currently use. The most compelling reason for this is that it is a “batteries included” solution. What we mean by “batteries included” is that Matlab provides a code editing and execution environment, plotting capabilities, and a vast catalog of pre-written numerical routines. For example, in this project, we were able to leverage built-in routines to construct Sobol sequences (quasi-Monte Carlo grids) and for doing numerical optimization. Another benefit is that, because so many other economists already use Matlab, collaborating on research code with others is efficient. There are two key examples of this. The first is that Dynare, which is mostly Matlab based, has become widely used across academia and policy institutions which facilitates communication and sharing of models. The second is that because many algorithms and routines have been previously written in Matlab, it should be easy for researchers to adapt these pieces of code to new projects.

Matlab, however, is not a perfect language for economists. Some of its downsides are:

- *Commercial* As a commercial product, in order to use Matlab users must first obtain a non-free license. Such a license can be costly if an institution does not already have a license available for use. This cost could be prohibitive for outside parties interested in replicating research.
- *Closed source* Users cannot look at the Matlab source code to study (or change) the implementation of built-in routines.
- *Restricted parallel programming possibilities (due to limited license availability)* As computer hardware becomes more powerful and computer resources in the cloud become more accessible, economists are increasingly implementing parallel versions of their algorithms. While basic parallelism is convenient in Matlab, running code at scale requires one license per process. This can increase the cost of large scale computations.[6]
- *Limited notion of community packages* Apart from the official Matlab toolboxes, Matlab does not have a well-developed system of add-on packages.[7] Using external libraries requires users to download and place the files in the same directory as the scripts they are running or to manage the path Matlab searches when calling functions. Although not insurmountable, this friction discourages good software development by making it difficult to build and reuse modular libraries.
- *Mainly a numerical programming language* Tasks outside of this purview can be more difficult than in a more general programming language.

2.2 Python

While Matlab has long been widely used in economics, Python has been steadily gaining users in economics during recent years. Some of Python's most compelling selling points are as follows:

- *General programming language* Python was built as a general programming language which has resulted in it being capable at fulfilling a wide variety of tasks—one saying that is often used to describe Python is, "while it may not be the best for any one task, it is the second best at everything."
- *A mature and rich package ecosystem.* At the time of writing, Python has 112,232 registered packages that perform a variety of tasks and can be easily downloaded and installed.
- *Advanced data manipulation and numerics libraries* For example, `pandas` is a very powerful library containing data structures and algorithms for doing data analysis. `numpy` and `scipy` are the base of array-based numerical computing in

[6] As an example, if one wanted to run a Matlab program on 16 optimized processors on the Amazon Compute Cloud, then in addition to paying $0.68/h for the processor time, it would cost an additional $0.06/h per processor to license each processor bringing the total cost from $0.68 to $1.64/h. For details see Matlab's pricing site and Amazon's pricing site.

[7] An exception is *Matlab Central* which is an open exchange for the Matlab users with 365,000 current contributors.

Python and provide linear algebra, optimization, and other foundational numerical algorithms. `matplotlib` is the main plotting library.

- *Many users and ample learning resources* Python is one of the top five most widely used programming languages in the world. This large user base is beneficial to economists because of the libraries mentioned above and the sheer volume of learning materials and resources available online for learning Python. As one point of data, at the time of writing there are 1,205,629 questions tagged with Python on http://stackoverflow.com[8], while there are only 84,329 for Matlab and 5551 for Julia.
- *Wide adoption in the private sector* This is beneficial because industry resources are being applied to enhance the language and create powerful tools economists can leverage in their computational code.
- *Open source* This means that the source code is publicly available on the internet and it is 100 percent free to view, install, distribute and use.

Python is not perfect, however. Some of its shortcomings are:

- *External libraries* External libraries are needed for efficient numerical computation. This means that users need to identify which libraries to use, know how to install them properly, and load them into their codes. Fortunately this is not as difficult as it sounds thanks to scientific Python distributions like Anaconda that come with Python and all the most common scientific packages installed and pre-configured.
- *Slower than other languages for certain types of algorithms.* This can often be circumvented by vectorizing operations (as in Matlab) or by leveraging tools such as numba or Cython to compile Python code into a fast and efficient machine code. In many cases this is a suitable approach, but it does require extra effort on behalf of the programmer. In other cases, it is difficult or impossible to express only the slower parts of an algorithm using these tools.
- *The Global interpreter lock (GIL)* The GIL prevents multiple threads from accessing Python objects at the same time which makes shared parallelism more difficult in Python than in the other two languages. Some packages such as `numpy`, `numba`, and `cython` have developed easy-to-use ways around this, but these solutions are aimed at certain types of operations.[9]

2.3 Julia

The most recently released language of the three is Julia. Julia released a stable 1.0 version of their language in August 2018. Despite it's youth, many enthusiastic economists are starting to pick it up because of the benefits that it offers. The list of Julia's strengths includes the following:

[8] This is a popular question-and-answer website for programming.

[9] The solutions mostly involve allowing for shared memory parallel for-loops, numerical operations, and linear algebra.

- *All of Julia is based around just-in-time (JIT) compilation* .[10] Since all Julia code is compiled before it is executed, Julia has the potential to run at the same speed as languages like C or Fortran.[11,12]
- *Most of Julia is written in Julia* This offers at least two main benefits: (1) core language features and constructs are written and defined using the same tools available to user code; (2) users can look to the implementation of Julia itself to see best practices and tips to make their own code better. One consequence of this that is often surprising to Matlab or Python users is that a Julia user can write a routine in Julia that performs as well, or often better, than implementations in common libraries.
- *Flexibility* Of the three languages that we analyze, Julia is the most expressive and flexible. It features advanced computer science concepts such as meta-programming (see below) and multiple dispatch (a choice of which method to execute when a function is applied; such a choice is made using all of a function's arguments, not just the first one).
- *Similarity to Matlab*. Julia's syntax is intentionally similar to that of Matlab. In fact, in many instances, copying and pasting Matlab code produces the same result in Julia as in Matlab. This was an intentional decision intended to allow Matlab users to transition seamlessly from Matlab to Julia.
- *Built in parallel and multi-threaded programming constructs* Having support at the language level for both distributed and shared memory parallel processing means running code in serial or in parallel is often a one word change to the user's code. Also, because Julia is open source like Python, its programs can scale up to run on arbitrarily many processes without incurring the additional costs from licensing.
- *Meta-programming* One relatively advanced and compelling feature that is unique to Julia (amongst the languages considered here) is meta-programming. A language that allows meta-programming treats the code that is executed as data that can be manipulated by the program itself. This means users can write code that writes the code that will be run. The most complete example of how we leveraged this feature for this project was in implementing routines to compute basis functions for complete polynomial. In Python and Matlab, our implementation was quite naive. We allow for a complete polynomial of degree between two and five and write out one `for` loop for each degree. The structure of each loop is identical. In contrast, the Julia version of this routine allows an arbitrary degree complete polynomial. To achieve this, we leverage Julia's meta-programming capabilities, receive the desired degree of complete polynomial from the user and then instruct Julia to emit code that contains that many for loops—each having the same structure as the hand-written loops in our Python and Matlab versions.

[10] Just-in-time compilation means that a function is compiled the first time it is called and all subsequent calls to that function are faster.

[11] ,See https://julialang.org/benchmarks/ for some example benchmarks.

[12] There has been work to add just-in-time compilation to both Matlab and Python, but, because it isn't native to either language, it does not work as generically as it does in Julia.

This results in the complete polynomial Julia routines having less code written by us and being more flexible and generic.

Despite many promising features, there are still some shortcomings to Julia. These include:

- *Slow compile times* Because the JIT compiler aggressively optimizes the machine code for each function that is executed, it can feel sluggish when doing rapid iterative development (make a change, run code, make a change, run code, etc...) because you pay the fixed compile cost after each edit.
- *Requires type inference* Julia must be able to infer a type for each variable in a function in order to emit fast code. This is not usually a problem, but sometimes if you are careless then the code will run slower than it should—though, even in the worst case, it is still often as fast as Matlab or Python.
- *Emerging package system* Many of the packages throughout the Julia community are surprisingly advanced given the youth of Julia, but they are usually not as polished as what you would find in the Matlab sponsored toolboxes or the best Python packages. We believe this is changing quickly and found suitable packages for many of the tools that we needed for this project.[13]
- *Lack of tooling* Unlike the more mature languages Matlab and Python, Julia currently lacks polished developer tools like an integrated development environment (IDE) or debugger. These are actively being worked on and will hopefully emerge in the near future.

2.4 Syntax Comparison

Julia and Matlab have many syntax elements in common. For example, if one wants to perform matrix multiplication of two matrices A and B then one can simply write `A*B`, but if one wants to perform element wise multiplication then one writes `A.*B`. Additionally, blocks are started with the same words (`for`, `if`, `while`) and terminated with the word `end`. These similarities make it very easy to translate code from Matlab to Julia and vice-versa. The one major difference between Julia and Matlab is that Julia uses square brackets to index into arrays (`A[i, j]`) while Matlab uses parenthesis (`A(i, j)`).

On the other hand, Python differs slightly in its conventions. In Python `A*B` represents element-wise multiplication between matrices while `A@B` represents matrix multiplications.[14] Additionally, rather than depend on `end` to denote the end of blocks, Python forces each new block to be indented four spaces from the previous

[13] The main exception to this was interpolation with complete polynomials but this was also missing in both Python and Matlab.

[14] The use of `@` for matrix multiplication was added in Python version 3.5. Readers who have seen or written Python code written for Python versions earlier than version 3.5 might have come across `A.dot(B)` or `np.dot(A, B)` instead.

one. Despite these differences, we found that syntax differences did not present a large obstacle for writing code in all three languages.

2.5 Parallelization Features

We briefly discuss the parallelization features contained in each language. There are two main paradigms for parallel programming: distributed memory and shared memory.

- In a distributed memory framework, the relevant variables are messaged to each individual processor and are then private to the individual processor. The processors then execute the sequence of commands using their private variables, compute the result, and then send it back to the original processor.
- In a shared memory framework, all processors share a set of memory. They execute the sequence of commands and typically store the result somewhere in the shared memory

Distributed memory benefits from being quite scalable (it doesn't require the processors to even be on the same chip), but the downside is that it requires some message passing which can take some time. Shared memory benefits from the fact that it does not require message passing, but it is less scalable because all processors must have access to the same RAM. Additionally, shared memory programming can sometimes be dangerous if one doesn't ensure that each process write the output to its own location in memory.

Julia supports many different types of parallelism, you can read more about it in the Julia documentation. We have found that for many problems, the simplest and most effective way to perform parallelization is to add `Threads.@threads` in front of a for-loop. This will use the shared memory paradigm and will parallelize over each of the iterable items in the for-loop.

In Matlab, parallelization can only be done through distributed memory parallelization. To do an operation in parallel, you can replace the keyword `for` with `parfor`. You can read more about this on the Matlab website.

Python supports both distributed and shared memory computing, but we have found that the easiest way to implement parallelism is to use `numba`'s `prange` function—You can read more about this in the numba documentation. Similar to Julia's `Threads.@threads`, this will use a shared memory approach and automatically distribute the work in the for-loop across multiple threads.

3 Comparison Using a Neoclassical Growth Model

The neoclassical growth model is often used as a benchmark to measure the effectiveness of a solution method. We keep with this tradition and analyze the stochastic neoclassical growth model with seven alternative solution methods, which are outlined in Arellano et al. (2016). Each method is implemented in Julia, Matlab, and

Python which allows us to see how the performance of different solution methods may vary by programming language.

3.1 The Model

We consider a dynamic programming problem of finding the value function, V, that solves the Bellman equation,

$$V(k, z) = \max_{c,k'} u(c) + \beta E\left[V(k', z')\right] \tag{1}$$

$$\text{s.t.} \quad k' = (1 - \delta)k + zf(k) - c \tag{2}$$

$$\ln z' = \rho \ln z + \varepsilon', \qquad \varepsilon' \sim \mathcal{N}(0, \sigma^2), \tag{3}$$

where k, c and z are capital, consumption and productivity level, respectively; $\beta \in (0, 1)$; $\delta \in (0, 1]$; $\rho \in (-1, 1)$; $\sigma \geq 0$; the utility and production functions, u and f, respectively, are strictly increasing, continuously differentiable and strictly concave. The primes on variables denote next-period values, and $E\left[V\left(k', z'\right)\right]$ is an expectation conditional on state (k, z).

Optimality conditions The first order condition (FOC) and envelope condition (EC) of the problem (1)–(3), respectively, are

$$u'(c) = \beta E\left[V_1\left(k', z'\right)\right], \tag{4}$$

$$V_1(k, z) = u'(c)\left[1 - \delta + zf'(k)\right], \tag{5}$$

By combining (4) and (5), we obtain the Euler equation

$$u'(c) = \beta E\left\{u'\left(c'\right)\left[1 - \delta + z'f'\left(k'\right)\right]\right\}. \tag{6}$$

Parameterization and implementation details We parameterize the model (1)–(3) by using a Constant Relative Risk Aversion (CRRA) utility function, $u(c) = \frac{c^{1-\gamma}-1}{1-\gamma}$, and a Cobb–Douglas production function, $f(k) = Ak^{\alpha}$. We choose parameters to be set at $A = \frac{1/\beta-(1-\delta)}{\alpha}$, $\alpha = 1/3$, $\beta = 0.99$, $\delta = 0.025$, $\rho = 0.95$ and $\sigma = 0.01$. As a solution domain, we use a rectangular, uniformly spaced grid of 10×10 points for capital and productivity between 0.9 and 1.1 for both variables.

We integrate by using a 10-node Gauss-Hermite quadrature rule and approximate the policy and value functions by using complete ordinary polynomials up to degree 5. As an initial guess, we use a linear approximation to the capital policy function. To solve for polynomial coefficients, we use fixed-point iteration.

All computations are performed using Julia v1.1.0, Matlab version 9.4.0 (R2018a), and Python 3.6.8 on a MacBook Pro with a 2.7 GHz Intel Core i7

processor and 16 GB of RAM. For Julia and Python, the particular package versions can be found in the corresponding notebooks submitted on QuantEcon's Notebook site.[15]

3.2 Value Iterative Methods

We analyze three value iterative algorithms for solving the model: (1) conventional value function iteration method analyzed in e.g., Fernández-Villaverde et al. (2015; 2) a variant of envelope condition method (ECM) of Maliar and Maliar (2013) that finds a solution to Bellman equation by iterating on value function; and (3) endogenous grid method (EGM) of Carroll (2005). We present a brief description of each algorithm below. A detailed description of each algorithm is included in Online Appendix A.

3.2.1 Conventional Value Function Iteration

Conventional VFI constructs the policy rule for consumption by combining FOC (4) and budget constraint (2):

Algorithm 1. Conventional VFI

Given V, for each point (k, z), define the following recursion:

i). Numerically solve for c satisfying $u'(c) = \beta E\left[V_1\left((1-\delta)k + zf(k) - c, z'\right)\right]$.

ii). Find $k' = (1-\delta)k + zf(k) - c$.

iii). Find $\widehat{V}(k, z) = u(c) + \beta E\left[V(k', z')\right]$.

Iterate on i)-iii) until convergence $\widehat{V} = V$.

3.2.2 Envelope Condition Method

ECM, proposed in Maliar and Maliar (2013), finds consumption from envelope condition (5) and budget constraint (2). In this specific model, the consumption function can be constructed analytically:

Algorithm 2. ECM-VF

Given V, for each point (k, z), define the following recursion:

i). Find $c = u'^{-1}\left[\frac{V_1(k,z)}{1-\delta+zf'(k)}\right]$.

ii). Find $k' = (1-\delta)k + zf(k) - c$.

iii). Find $\widehat{V}(k, z) = u(c) + \beta E\left[V(k', z')\right]$.

Iterate on i)-iii) until convergence $\widehat{V} = V$.

[15] The Matlab notebook, Python notebook, Julia notebook.

Table 1 Accuracy and speed of value iterative methods

Degree	L_1	L_∞	Julia CPU (s)	Matlab CPU (s)	Python CPU (s)
Conventional VFI (VFI)					
2nd	– 3.83	– 2.76	1.05	30.94	10.97
3rd	– 4.97	– 3.32	0.92	20.85	6.67
4th	– 6.06	– 4.03	1.13	18.02	5.80
5th	– 7.00	– 4.70	1.00	13.80	4.51
Envelope condition method (ECM)					
2nd	– 3.83	– 2.76	0.29	0.32	0.37
3rd	– 4.97	– 3.32	0.21	0.21	0.24
4th	– 6.06	– 4.03	0.27	0.22	0.27
5th	– 7.00	– 4.70	0.14	0.10	0.15
Endogenous grid method (EGM)					
2nd	– 3.81	– 2.76	0.35	27.19	5.05
3rd	– 4.95	– 3.34	0.25	18.51	3.43
4th	– 6.06	– 4.05	0.29	13.95	3.19
5th	– 7.04	– 4.73	0.19	10.43	2.28

[a] L_1 and L_∞ are, respectively, the average and maximum of absolute residuals across optimality conditions and test points (in log 10) units on a stochastic simulation of 10,000 observations. CPU is the time necessary for computing a solution (in s)

3.2.3 Endogenous Grid Method

Finally, EGM of Carroll (2005) constructs a grid on (k', z) by fixing the future endogenous state variable k' and by treating the current endogenous state variable k as unknown; see also Barillas and Fernandez-Villaverde (2007)for a discussion of the EGM method. Since k' is fixed, EGM computes $E\left[V_1\left(k', z'\right)\right]$ up-front and thus can avoid costly interpolation and approximation of expectation in a root finding procedure.

Algorithm 3. EGM of Carroll (2005)

Given V, for each point (k', z), define the following recursion:

i). Find $c = u'^{-1}\{\beta E[V_1(k', z')]\}$.

ii). Solve for k satisfying $k' = (1-\delta)k + zf(k) - c$.

iii). Find $\widehat{V}(k, z) = u(c) + \beta E[V(k', z')]$.

Iterate on i)-iii) until convergence $\widehat{V} = V$.

In Step (ii) of EGM, we still need to find k numerically. However, for the studied model, Carroll (2005) shows that a change of variables makes it possible to avoid finding k numerically on each iteration (except of the very last iteration). In order to streamline our code, we do not use this change of variables so the running time of our version of the EGM method will be larger than for the version of the method in Carroll (2005). This decision has an impact on the relative performance of the three languages since Julia performs best when a numerical solver is involved, however,

we are ok with this difference since in slightly more complicated settings the change of variables will not always apply. We view this as a more general comparison than if we had applied the change of variables.

3.2.4 Comparison Results of Matlab, Python and Julia for Value Iterative Methods

The results from our comparison of the three iterative methods are in Table 1.

Conventional VFI is the most time consuming in all three languages because it requires us to find the root of (4) for each (k, z). Given (k, z), each evaluation of equation (4) requires the computer to compute conditional expectation by interpolating over the $t + 1$ values. Numerical root finders must do this repeatedly. The reason EGM performs better than VFI is that it is easier to solve equation (4) with respect to c given $\left(k', z\right)$ than to solve it with respect to c given (k, z) because we only need to evaluate conditional expectation once if we know k'. The ECM method requires no numerical solver which is why it is so efficient relative to the other two methods.[16]

Julia produces solutions between 5 and 50 times faster than Matlab and Python for VFI and EGM. The reasons Julia performs so much better on these two algorithms is because the entire language is built on JIT compilation. This means that during the root solving step the repeated calls to the Julia objective function are cheap (relative to an interpreted language like Matlab or Python) because these subsequent function calls execute already compiled machine code.

Our results for the ECM show similar performance for all three languages. The main reason for this is that the method does not require a non-linear solver to compute the policy for consumption and thus the computations can be vectorized. The vectorization puts the languages on more similar footing because they all end up calling out to the same high preference BLAS routines implemented in C or Fortran.

3.3 Policy Iterating Methods

We analyze four algorithms that construct policy functions for the standard neoclassical stochastic growth model: Algorithm 4 is a conventional policy iteration (PI) method, e.g., Santos and Rust (2008); Algorithm 5 is a variant of ECM that implements policy iteration; Algorithm 6 is a version of ECM that solve for derivative of value function instead of value function itself; Algorithm 7 an Euler equation algorithm. Again we give a brief description of each algorithm but a more detailed description of these algorithms is provided in Online Appendix A.

3.3.1 Conventional Policy Iteration

Conventional policy iteration constructs a solution to the Bellman equation by iterating on the consumption function using FOC (4).

[16] A version of the envelope condition argument is used in Achdou et al. (2017) to construct a new class of fast and efficient numerical methods for solving dynamic economic models in continuous time.

Algorithm 4. Conventional PI
Given C, for each point (k, z), define the following recursion:
i). Find $K(k, z) = (1-\delta)k + zf(k) - C(k, z)$
ii). Solve for V satisfying $V(k, z) = u(C(k, z)) + \beta E[V(K(k, z), z')]$
iii). Find $\widehat{C}$ satisfying $u'\left(\widehat{C}(k, z)\right) = \beta E\left[V_1\left((1-\delta)k + zf(k) - \widehat{C}(k, z), z'\right)\right]$
Iterate on i)-iii) until convergence $\widehat{C} = C$.

3.3.2 ECM Policy Iteration

ECM-PI the variant of ECM that performs PI instead of VFI. It constructs a solution to the Bellman equation by iterating on the consumption function using EC (5).

Algorithm 5. ECM-PI
Given C, for each point (k, z), define the following recursion:
i). Find $K(k, z) = (1-\delta)k + zf(k) - C(k, z)$
ii). Solve for V satisfying $V(k, z) = u(C(k, z)) + \beta E[V(K(k, z), z')]$
iii). Find $\widehat{C}(k, z) = u'^{-1}\left[\frac{V_1(k,z)}{1-\delta+zf'(k)}\right]$
Iterate on i)-iii) until convergence $\widehat{C} = C$.

3.3.3 Derivative Policy Iteration

The ECM analysis also suggests a useful recursion for the derivative of value function. We first construct consumption function $C(k, z)$ satisfying FOC (4) under the current value function

$$\beta E\left[V_1\left(k', z'\right)\right] = u'(C(k, z)),$$

and we then use (5) to obtain the derivative of value function for next iteration. This leads to a solution method that we call ECM-DVF.

The main difference between ECM-DVF from the previously studied ECM-VF consists in that we iterate on V_1 without computing V on each iteration. We only compute V at the very end, when both V_1 and the optimal policy functions are constructed. Again, neither numerical maximization nor a numerical solver is necessary under ECM-DVF but only direct calculations. Similarly to PI methods, Euler equation methods do not solve for value function but only for decision (policy) functions. One possible decision function is a derivative of value function. Thus, the ECM-DVF recursion can be also viewed as an Euler equation written in terms of the derivative of value function.

Algorithm 6. ECM-DVF

Given V_1 for each point (k, z), define the following recursion:

i). Find $c = u'^{-1}\left[\frac{V_1(k,z)}{1-\delta+zf^{prime}(k)}\right]$

ii). Find $k' = (1-\delta)\,k + zf\,(k) - c$

iii). Find $\widehat{V}_1\,(k, z) = \beta\,[1-\delta+zf'\,(k)]\,E\,[V_1\,(k', z')]$

Iterate on i)-iii) until convergence $\widehat{V}_1 = V_1$

Given the converged policy functions, find V satisfying $V\,(k, z) = u\,(c) + \beta E\,[V\,(k', z')]$

3.3.4 Euler Equation Methods

Policy iteration has similarity to Euler equation methods; see Judd (1998), Santos (1999) for a general discussion of such methods. Euler equation methods approximate policy functions for consumption $c = C(k, z)$, capital $k' = K(k, z)$ (or other policy functions) to satisfy (2), (3) and (6). Below, we provide an example of Euler equation method (many other recursions for such methods are possible).

Algorithm 7. Euler equation algorithms

Given $C\,(k, z)$, for each point (k, z), define the following recursion:

i). Find $K\,(k, z) = (1-\delta)\,k + zf\,(k) - C\,(k, z)$

ii). Find $c' = C\,(K\,(k, z)\,, z')$

iii). Find $\widehat{C}\,(k, z) = u'^{-1}\,\{\beta E\,[u'\,(c')\,(1-\delta+z'f'\,(k'))]\}$

Iterate on i)-iii) until convergence $\widehat{C} = C$.

3.3.5 Comparison Results of Matlab, Python and Julia for Policy Iterative Methods

The results from our comparison of the policy iterative methods are in Table 2.

We see that conventional PI performs the worst for each language due to its reliance on a numerical solver, but, as with the value iterative methods, this reliance is less problematic for Julia than for Matlab or Python.

Overall, we found that the speed of most algorithms was comparable across all three languages. Again, the main exception to this was that Julia performed significantly better than the other programs when there was any numerical optimization or root-finding involved. However, these speedups came at the cost of spending a (relatively small) amount of extra time being careful about how the code was written to ensure that the Julia compiler is able to infer the type of each variable in the most time-intensive parts of the program.

Table 2 Accuracy and speed of policy iterative methods

Degree	L_1	L_∞	Julia CPU (s)	Matlab CPU (s)	Python CPU (s)
Conventional PI					
2nd	− 3.81	− 2.76	0.29	20.32	5.51
3rd	− 4.95	− 3.34	0.78	16.49	5.27
4th	− 6.06	− 4.05	1.14	17.73	5.37
5th	− 7.04	− 4.73	1.04	13.46	4.36
Envelope condition method iterrating on policy functions (ECM-PI)					
2nd	− 3.81	− 2.76	0.27	0.30	0.48
3rd	− 4.95	− 3.34	0.22	0.17	0.32
4th	− 6.06	− 4.05	0.28	0.22	0.40
5th	− 7.04	− 4.73	0.16	0.11	0.26
Envelope condition method iterating on derivative of value function (ECM-DVF)					
2nd	− 3.81	− 2.76	0.57	0.57	0.62
3rd	− 4.95	− 3.34	0.67	0.70	0.68
4th	− 6.06	− 4.05	0.75	0.69	0.72
5th	− 7.04	− 4.73	0.87	0.74	0.80
Euler equation method					
2nd	− 3.81	− 2.76	0.35	0.38	0.38
3rd	− 4.95	− 3.34	0.26	0.29	0.25
4th	− 6.06	− 4.05	0.32	0.25	0.24
5th	− 7.04	− 4.73	0.28	0.11	0.12

[a] L_1 and L_∞ are, respectively, the average and maximum of absolute residuals across optimality conditions and test points (in log 10) units on a stochastic simulation of 10,000 observations. CPU is the time necessary for computing a solution (in seconds)

3.4 Parallel Computing

We explore the parallelization features of each language by solving the growth model in serial and in parallel. We then compare the time taken to compute the solution using a various numbers of processors. Rather than solve the original continuous-state and choice model (1)–(3), we discretize the state and choice variables. The resulting problem is described by the following Bellman equation

$$V(k_i, z_j) = \max_{k^* \in \mathbb{K}} u((1-\delta)k_i + Ak_i^\alpha - k^*) + \beta \sum_k \Pi_{jk} V(k^*, z_k),$$

where the states are discretized according to $\mathbb{K} = \{k_1, k_2, \dots, k_{N_k}\}$, $z_j \in \mathbb{Z} = \{z_1, \dots, z_{N_z}\}$, and Π denotes the transition matrix for z with Π_{jk} denoting the probability of transition from state j to state k. We determine $\mathbb{Z}$ and Π from the Rouwenhorst method for approximating AR(1)s and $\mathbb{K}$ is chosen to be equally spaced points between 0.75 and 1.25. We choose to use 201 points on the grid for k and 25 points on the grid for z. All other parameters are the same as before.

We solve the model using conventional VFI. In particular, we choose an initial guess at V to be a matrix of zeros. We then iterate over each possible combination of k_i and z_j

Table 3 Paralelization results for discretized value iterative problem

Number of processors	Julia (s)	Matlab (s)	Python (s)
1	36.91	67.3	21.30
4	12.58	24.1	7.61
16	4.21	8.03	4.33
64	2.84	4.56	3.01

[a]Speed at which a particular problem is solved. The columns are each a different programming language and the rows denote the number of threads we use to do the computation

and determine the optimal response, k^*, at each state. In each iteration s, we update the value function according to

$$V_{i,j}^{s+1} = u((1-\delta)k_i + Ak_i^\alpha - k_{i,j}^*) + \beta \sum_k \Pi_{j,k} V_{k_{i,j}^*,k}^s.$$

We include a timing comparisons with different numbers of threads for each of these examples in Table 3.

Again, we see that all three languages generate comparable running time. The code for this exercise can be found in the github repository with the rest of this paper's code. We should point out that the best way to parallelize computation may differ by languages. For example, in Matlab matrices are stored by column and so parfor across columns is much faster than parfor across rows. For more extensive discussion of parallelization, we recommend to read Fernández-Villaverde and Zarruk Valencia (2018).

4 Comparison Using a New Keynesian Model

In this section, we solve a stylized medium-scale Keynesian model using equivalent codes implemented in Julia, Matlab, and Python. The model was previously analyzed in Maliar and Maliar (2015). It features Calvo-type price frictions and a Taylor rule with a zero lower bound on the nominal interest rate.

4.1 The Model

To save on space, we present the fully-specified model in Online Appendix B.

First-order Conditions Here, we summarize the model by showing the set of optimality conditions used to construct the numerical solutions:

$$S_t = \frac{\exp\left(\eta_{u,t} + \eta_{L,t}\right)}{\exp\left(\eta_{a,t}\right)} L_t^\vartheta Y_t + \beta\theta E_t\left\{\pi_{t+1}^\varepsilon S_{t+1}\right\}, \tag{7}$$

$$F_t = \exp\left(\eta_{u,t}\right) C_t^{-\gamma} Y_t + \beta\theta E_t\left\{\pi_{t+1}^{\varepsilon-1} F_{t+1}\right\}, \tag{8}$$

$$\frac{S_t}{F_t} = \left[\frac{1-\theta\pi_t^{\varepsilon-1}}{1-\theta}\right]^{\frac{1}{1-\varepsilon}}, \tag{9}$$

$$\Delta_t; = \left[(1-\theta)\left[\frac{1-\theta\pi_t^{\varepsilon-1}}{1-\theta}\right]^{\frac{\varepsilon}{\varepsilon-1}} + \theta\frac{\pi_t^{\varepsilon}}{\Delta_{t-1}}\right]^{-1}, \tag{10}$$

$$C_t^{-\gamma} = \beta\frac{\exp\left(\eta_{B,t}\right)}{\exp\left(\eta_{u,t}\right)} R_t E_t\left[\frac{C_{t+1}^{-\gamma}\exp\left(\eta_{u,t+1}\right)}{\pi_{t+1}}\right], \tag{11}$$

$$Y_t = \exp\left(\eta_{a,t}\right) L_t \Delta_t \tag{12}$$

$$C_t = \left(1 - \frac{\bar{G}}{\exp\left(\eta_{G,t}\right)}\right) Y_t \tag{13}$$

$$R_t = \max\left\{1, R_*\left(\frac{R_{t-1}}{R_*}\right)^{\mu}\left[\left(\frac{\pi_t}{\pi_*}\right)^{\phi_\pi}\left(\frac{Y_t}{Y_{N,t}}\right)^{\phi_y}\right]^{1-\mu}\exp\left(\eta_{R,t}\right)\right\}, \tag{14}$$

where S_t and F_t are supplementary variables reflecting profit maximizing conditions of firms; Δ_t is a measure of price dispersion across firms; C_t, Y_t, L_t are consumption, output and labor, respectively; π_t and R_t are inflation and interest rate, respectively; $\pi*$ and $R* = \frac{\pi*}{\beta}$ are the inflation target and the steady state interest rate, respectively; $Y_{N,t} = \left[\frac{\exp\left(\eta_{a,t}\right)^{1+\vartheta}}{\left[\exp\left(\eta_{G,t}\right)\right]^{-\gamma}\exp\left(\eta_{L,t}\right)}\right]^{\frac{1}{\vartheta+\gamma}}$ is the natural level of output. The parameters are β (discount factor), θ (fraction of non-reoptimizing firms), ε (inverse of the elasticity of substitution in the final-good production), ϑ and γ (utility-function parameters), $\overline{G}$ (steady-state government spending), and μ, ϕ_y and ϕ_π (Taylor-rule parameters). Each exogenous variable $\eta_{z,t} \in \left\{\eta_{u,t}, {}_{L,t}, \eta_{a,t}, \eta_{u,t}, \eta_{B,t}, \eta_{G,t}\right\}$ follows an AR(1) process $\eta_{z,t+1} = \rho_z\eta_{z,t} + \epsilon_{z,t+1}$ with $\epsilon_{z,t+1} \sim N\left(0, \sigma_z^2\right)$, $\sigma_z > 0$ and $\rho_z \in (-1, 1)$.

4.2 Parameterization and Implementation Details

We parameterize the model by $\beta = 0.99$, $\gamma = 1$, $\vartheta = 2.09$, $\varepsilon = 4.45$, $\theta = 0.83$ and $\overline{G} = 0.23$. The autocorrelation coefficients in six exogenous variables are $\rho_a = 0.95$, $\rho_B = 0.22$, $\rho_R = 0.15$, $\rho_u = 0.92$, $\rho_G = 0.95$, $\rho_L = 0.25$, and the corresponding

standard deviations are $\sigma_u = 0.54\%$, $\sigma_G = 0.38\%$, $\sigma_L = 18.21\%$, $\sigma_a = 0.45\%$, $\sigma_B = 0.23\%$ and $\sigma_R = 0.28\%$. The parameters in the Taylor rule are $\phi_y = 0.07$, $\phi_\pi = 2.21$, and $\mu = 0.82$. These values are used in Maliar and Maliar (2015) and are in line with the estimates in del Negro et al. (2007) and Smets and Wouters (2007).

As a solution domain, we use random and quasi-random grids constructed inside an 8-dimensional hypercube; we describe the grid construction in Sect. 4.4. To approximate the equilibrium policy rules, we use a family of ordinary polynomials up to degree 5. To compute conditional expectations in Euler equations (7), (8) and (11), we use a monomial integration rule (either a formula with $2N$ nodes or the one with $2N^2 + 1$ nodes); see Judd et al. (2011) for a detailed description of the monomial integration formulas. To solve for the polynomial coefficients, we use fixed point iteration. Again, Julia, Matlab, and Python software can be found in the corresponding notebooks submitted on QuantEcon.[17]

4.3 Computational Method

We now outline the Euler equation algorithm used to solve the neoclassical model. An extended description of this algorithm is provided in Online Appendix B.

Algorithm 8. Euler equation algorithms for the new Keynesian model.

Given $S(\Delta, R, \eta_z)$, $F(\Delta, R, \eta_z)$ and $C(\Delta, R, \eta_z)$ for each point (Δ, R, η_z), define the following recursion:

i). Find π from $\frac{S}{F} = \left[\frac{1-\theta\pi^{\varepsilon-1}}{1-\theta}\right]^{\frac{1}{1-\varepsilon}}$ and $\Delta' = \left[(1-\theta)\left[\frac{1-\theta\pi^{\varepsilon-1}}{1-\theta}\right]^{\frac{\varepsilon}{\varepsilon-1}} + \theta\frac{\pi^\varepsilon}{\Delta}\right]^{-1}$;

– $Y = \left(1 - \frac{\overline{G}}{\exp(\eta_G)}\right)^{-1} C$, and $L = Y\left[\exp(\eta_a)\Delta'\right]^{-1}$;

– $Y_N = \left[\frac{\exp(\eta_a)^{1+\vartheta}}{[\exp(\eta_G)]^{-\gamma}\exp(\eta_L)}\right]^{\frac{1}{\vartheta+\gamma}}$;

– $R' = \max\left\{1, R_*\left(\frac{R}{R_*}\right)^{\mu}\left[\left(\frac{\pi}{\pi_*}\right)^{\phi_\pi}\left(\frac{Y}{Y_N}\right)^{\phi_y}\right]^{1-\mu}\exp(\eta_R)\right\}$.

ii). Find $S(\Delta', R', \eta_z')$, $F(\Delta', R', \eta_z')$ and $C(\Delta', R', \eta_z')$;

iii). Find π' from $\frac{S'}{F'} = \left[\frac{1-\theta\left(\pi'\right)^{\varepsilon-1}}{1-\theta}\right]^{\frac{1}{1-\varepsilon}}$;

– $\widehat{S}(\Delta, R, \eta_z) = \frac{\exp(\eta_u+\eta_L)}{\exp(\eta_a)} L^\vartheta Y + \beta\theta E\left\{(\pi')^\varepsilon S'\right\}$;

– $\widehat{F}(\Delta, R, \eta_z) = \exp(\eta_u) C^{-\gamma} Y + \beta\theta E\left\{\left(\pi'\right)^{\varepsilon-1} F'\right\}$;

– $\widehat{C}(\Delta, R, \eta_z) = \left\{\frac{\beta\exp(\eta_B)R'}{\exp(\eta_u)} E\left[\frac{(C')^{-\gamma}\exp(\eta_u')}{\pi'}\right]\right\}^{-1/\gamma}$.

Iterate on i)-iii) until convergence $\widehat{F} = F$, $\widehat{S} = S$, $\widehat{C} = C$.

4.4 Grid Techniques for High Dimensional Applications

Tensor product grids can be successfully used for analyzing small-scale applications but their cost increases exponentially with the number of the state variables (curse of dimensionality). In particular, tensor product grid are expensive even for

[17] Matlab notebook, Python notebook, Julia notebook.

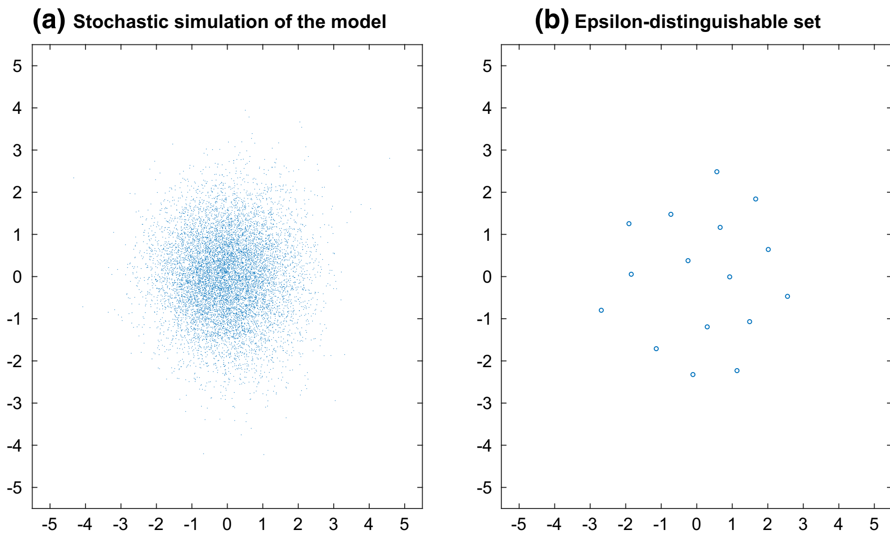

Fig. 1 Stochastic-simulation grid and epsilon-distinguishable set grid

moderately-large models like our model with 8 state variables. There is a number of alternative grid techniques that do not rely on tensor products and that ameliorate the curse of dimensionality.

To solve the model (7)–(14), Maliar et al. (2015)construct the grid on the high-probability area of the state space which is approximated by stochastic simulation. However, crude simulated points are not uniformly spaced, in particular, many simulated points are situated close to one another and are redundant for the purpose of the grid construction. They introduce two techniques for selecting a roughly uniformly-spaced subset of simulated points. One technique is clustering analysis: the simulated points are grouped into clusters and the centers of the clusters are used as a grid. Another technique is an *ε-distinguishable set* (EDS) method that selects a subset of points situated at the distance of at least ε from one another. As an example, in Fig. 1 we show a stochastic-simulation grid and epsilon-distinguishable set grid.

The EDS or cluster grids are constructed iteratively using the following steps: (i) guess policy functions and use them to simulate the time series solution; (ii) build an EDS or cluster grid; (iii) solve the model on this grid; (iv) then use the new solution to form a new grid. This procedure is repeated until the solution and grid both remain constant on successive iterations. The advantage of this approach is that it focuses on the right area of the state space—a high probability set. However, it requires to have initial guess for the solution (the paper uses linearization solution from Dynare) and it reconstructs the EDS or cluster grid iteratively, which leads to larger running times. Finally, the code is more complicated.

In the present paper, we modify the grid construction relative to the method in Maliar and Maliar (2015). We specifically replace the EDS and cluster grids with random and quasi-random grids covering a fixed multi-dimensional hypercube.

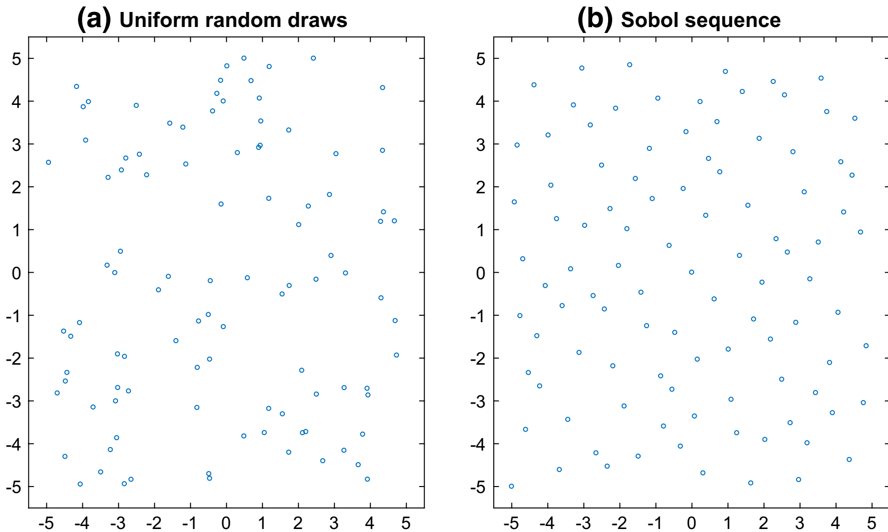

Fig. 2 Uniformly-spaced random grid and quasi-Monte Carlo grid

To construct the hypercube, we choose the interval $\left[-\frac{2\sigma_z}{\sqrt{1-\rho_z^2}}, \frac{2\sigma_z}{\sqrt{1-\rho_z^2}}\right]$ for each of six exogenous variable $\eta_z \in \{\eta_u, \eta_L, \eta_a, \eta_u, \eta_B, \eta_G\}$, where σ_z and ρ_z are standard deviation and auto-correlation coefficient. For endogenous state variables R and Δ, we chose intervals that cover the high probability set, $\Delta \in [0.95, 1]$ and $R \in [1, 1.05]$.

We consider two alternative grid techniques in the constructed fixed hypercube. One is a *random grid* obtained by drawing an 8-dimensional set of random uncorrelated uniformly distributed points. The other is *quasi-random grids* (also known as *quasi-Monte Carlo* sequence or *low discrepancy sequence*). Uniformly distributed random draws converge to an evenly spaced grid as a number of draws increases but the convergence rate is low; in turn, low disrepancy sequences are constructed to be evenly spaced and their convergence rate is much faster. We use Sobol low discrepancy sequence for constructing the grid but there is a variety of other low discrepancy sequences that can be used for this purpose; see Niederreiter (1992). The random and quasi-random grids are shown in Fig. 2.

Using random grids has its advantages and disadvantages. Unlike the EDS and cluster grids used in Maliar and Maliar (2015), the random and quasi-random grids operate on a fixed hypercube and do not benefit from the domain reduction. However, the random and quasi-random grids are much faster and easier to construct than the EDS and cluster grids. By constructing grids on predefined intervals, we did not have to write our own rough solution routine or rely on external software like Dynare to construct a preliminary solution to the model as we would with cluster grid or EDS techniques and we do not need to re-construct the grids iteratively. The Matlab code accompanying the present paper achieves an

Table 4 Accuracy and speed of a projection method for solving new Keynesian model

Degree	L_1	L_∞	Julia CPU (s)	Matlab CPU (s)	Python CPU (s)
Inflation target $\pi_* = 1.0598$					
1st	− 3.41	− 1.94	0.45	0.73	1.04
2nd	− 4.71	− 3.13	1.35	1.64	2.17
3rd	− 6.07	− 4.25	7.96	7.39	6.99
4th	− 6.73	− 4.65	52.79	52.60	58.07
5th	− 7.00	− 5.47	756.03	877.40	1496.60
Inflation target $\pi_* = 1$					
1st	− 3.12	− 1.73	0.22	0.40	0.52
2nd	− 4.40	− 2.77	0.52	0.71	0.93
3rd	− 5.71	− 3.54	3.05	3.11	2.94
4th	− 6.82	− 4.89	22.35	22.51	23.67
5th	− 7.01	− 5.12	318.44	389.47	675.92
Inflation target $\pi_* = 1$ with ZLB					
1st	− 3.12	− 1.73	0.22	0.41	0.54
2nd	− 4.40	− 2.16	0.49	0.71	0.95
3rd	− 5.60	− 2.15	3.48	2.82	2.97
4th	− 6.17	− 2.15	22.09	22.44	24.69
5th	− 6.20	− 2.15	313.24	389.03	635.58

[a] L_1 and L_∞ are, respectively, the average and maximum of absolute residuals across optimality conditions and test points (in log 10) units on a stochastic simulation of 10,000 observations. CPU is the time necessary for computing a solution (in seconds)

almost a 60-time speed up over the original code in Maliar et al. (2015), while still producing highly accurate solutions.

4.4.1 Numerical Results

We have provided an implementation of this algorithm in the Matlab, Python, and Julia programming languages. In this section we discuss our experience with the three languages.

We report the solution for two parameterizations that are previously analyzed in Maliar and Maliar (2015). To save on space, we report the solution only for the quasi-random (Sobol) grid. The solution for the random grid is similar (although slightly less accurate). Our two parameterizations differ in the value of the inflation target π_*. In our first experiment, we use $\pi_* = 1.0598$, which corresponds to long term average as estimated in del Negro et al. (2007), while in the second experiment, we use $\pi_* = 1$. For the second experiment, we report results when we solve the model both when the ZLB is and is not imposed (in the first experiment, ZLB is never active) (Table 4).

The computational cost increases rapidly with a polynomial degree. The number of points in the grid must be at least as large as the number of the polynomial coefficients to identify such coefficients. In particular, for the polynomial degrees 1,

2, 3, 4 and 5, we construct 20,100, 300, 1000 and 2000 grid points respectively to identify 9, 45, 165, 495 and 1287 coefficients (we use about 50% more of grid points than the polynomial coefficients to enhance the numerical stability of the algorithm). The cost of second-order polynomial approximations is about 1–3 s depending on specific parameterization used. It is fast enough to repeatedly solve the model inside an estimation procedure.

Concerning the accuracy, in our first experiment, the inflation target is relatively high and the probability of reaching a zero lower bound (ZLB) on nominal interest rates is so low that it is never observed in finite simulation (unless we use the initial condition that leads to ZLB). In the absence of binding ZLB, the accuracy of our numerical solutions increases considerably with the degree of polynomial, in particular, the maximum residuals across the model's equations decrease from −1.94 to −5.47 when the polynomial order increases from 1 to 5 respectively. The accuracy of the solutions is surprisingly high given that we assume very large volatility of labor shock of 18%. If we reduce the size of the labor shock to 5%, the residuals in the table decrease by nearly 2 orders of magnitude and the accuracy levels will be comparable to those we obtained for the neoclassical growth model.

In our second experiment, the inflation target is low and the probability of reaching the ZLB increases to about 2%. When we do not impose the ZLB, the accuracy of the solution looks very similar to our first parameter set. However, with the ZLB imposed, the results are different. Relatively small average residuals indicate that the solution is sufficiently accurate in most of the domain and large maximum residuals show that the accuracy declines sharply in the ZLB area. As this example shows, even the global solution methods might still be insufficiently accurate in the presence of active ZLB because smooth polynomials functions are not well suitable for approximating the ZLB kink. Aruoba et al. (2017) and Maliar and Maliar (2015) show that the accuracy can be slightly increased by using piecewise linear and locally adaptive approximations, respectively, but these extensions lie beyond the scope of our analysis.

5 Conclusion

We have described and implemented seven algorithms to solve the stochastic neoclassical growth model, and we have described and analyzed a modification of an algorithm that has been previously used to globally solve new Keynesian models. The implementation has provided us a "laboratory" to explore some of the strengths and weaknesses of the three languages we consider. One insight we gained was that for some algorithms, the choice of programming language has little effect on performance, but for others, there are important speedups that can be obtained. In particular, if an algorithm depends heavily on numerical optimization or root-finding then it is likely that Julia will provide significant speedups, but these gains may come at the cost of ensuring that your code provides enough information to the compiler.

We hope we have brought some clarity in the trade-off between these three programming languages which will first, provide information to economists who are thinking about which language is the best fit for them; and second, provide a clear

description, examples, and tools which can facilitate other researchers implement similar algorithms for their own models.

Funding Lilia Maliar and Serguei Maliar acknowledge the support from NSF grants SES-1949413 and SES-1949430, respectively.

References

Achdou, Y., Han, J., Lasry, J.-M., Lions, P.-L., & Moll, B. (2017). Income and wealth distribution in macroeconomics: A continuous-time approach. NBER Working Paper No. w23732.

Arellano, C. (2008). Default risk and income fluctuations in emerging economies. *American Economic Review*, *98*(3), 690–712.

Arellano, C., Maliar, L., Maliar, S., & Tsyrennikov, V. (2016). Envelope condition method with an application to default risk models. *Journal of Economic Dynamics and Control*, *69*, 436–459.

Aruoba, S., & Fernandez-Villaverde, J. (2015). A comparison of programming languages in economics. *Journal of Economic Dynamics and Control*, *58*, 265–273.

Aruoba, S., & Fernandez-Villaverde, J. (2018). A comparison of programming languages in economics: An update.

Barillas, F., & Fernandez-Villaverde, J. (2007). A generalization of the endogenous grid method. *Journal of Economic Dynamics and Control*, *31*, 2698–2712.

Carroll, K. (2005). The method of endogenous grid points for solving dynamic stochastic optimal problems. *Economic Letters*, *91*, 312–320.

Fernández-Villaverde, J., Rubio-Ramírez, J., & Schorfheide, F. (2015). Solution and estimation methods for DSGE models. *Handbook of Macroeconomics*, *2*, 527–724.

Fernández-Villaverde, J., & Zarruk Valencia, D. (2018). A practical guide to parallelization in economics.

Guerrieri, L., & Iacoviello, M. (2015). OccBin: A toolkit for solving dynamic models with occasionally binding constraints easily. *Journal of Monetary Economics*, *70*, 22–38.

Hasanhodzic, J., & Kotlikoff, L. (2013). Generational risk—Is it a big deal? Simulating an 80-period OLG model with aggregate shocks. NBER Working Paper 19179.

Judd, K. (1998). *Numerical methods in economics*. Cambridge, MA: MIT Press.

Judd, K., Maliar, L., & Maliar, S. (2011). Numerically stable and accurate stochastic simulation approaches for solving dynamic models. *Quantitative Economics*, *2*, 173–210.

Judd, K., Maliar, L., & Maliar, S. (2016). Lower bounds on approximation errors to numerical solutions of dynamic economic models. *Econometrica*, *85*(3), 991–1012.

Kollmann, R., Kim, S., & Kim, J. (2011). Solving the multi-country real business cycle model using a perturbation method. *Journal of Economic Dynamics and Control*, *35*, 203–206.

Krusell, P., & Smith, A. A, Jr. (1998). Income and wealth heterogeneity in the macroeconomy. *Journal of Political Economy*, *106*(5), 867–896.

Laseen, S., & Svensson, L. (2011). Anticipated alternative policy rate paths inpolicy simulations. *UCB International Journal of International Banking*, *7*(3), 1–36.

Lepetuyk, V., Maliar, L., & Maliar, S. (2019). When the U.S. Catches a Cold, Canada Sneezes: A Lower-Bound Tale Told by Deep Learning. CEPR working paper DP 14025.

Maliar, L., & Maliar, S. (2013). Envelope condition method versus endogenous grid method for solving dynamic programming problems. *Economic Letters*, *120*, 262–266.

Maliar, L., & Maliar, S. (2014). Numerical methods for large scale dynamic economic models. In K. Schmedders & K. Judd (Eds.), *Handbook of computational economics* (Vol. 3). Amsterdam: Elsevier Science.

Maliar, L., & Maliar, S. (2015). Merging simulation and projection approaches to solve high-dimensional problems with an application to a new Keynesian Model. *Quantitative Economics*, *6*, 1–47.

Maliar, L., Maliar, S., Taylor, J.B., & Tsener, I. (2015). A tractable framework for analyzing a class of nonstationary markov models. NBER working paper 21155.

Maliar, L., Maliar, S., & Winant, P. (2019). Will Artificial Intelligence Replace Computational Economists Any Time Soon?. CEPR working paper DP 14024.

Negro, M. D., et al. (2007). On the fit of new Keynesian models. *Journal of Business & Economic Statistics*, *25*(2), 123–143.

Petrosky-Nadeau, N., & Zhang, L. (2017). Solving the Diamond–Mortensen-pissarides model accurately. *Quantitative Economics*, *8*(2), 611–650.

Santos, M. (1999). Numerical solution of dynamic economic models. In J. Taylor & M. Woodford (Eds.), *Handbook of macroeconomics* (pp. 312–382). Amsterdam: Elsevier Science.

Santos, M., & Rust, J. (2008). Convergence properties of policy iteration. *SIAM Journal on Control and Optimization*, *42*(6), 2094–2115.

Smets, F., & Wouters, R. (2007). Shocks and frictions in US business cycles: A Bayesian DSGE approach. *American Economic Review*, *97*(3), 586–606.

Taylor, J., & Uhlig, H. (1990). Solving nonlinear stochastic growth models: A comparison of alternative solution methods. *Journal of Business and Economic Statistics*, *8*, 1–17.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.