




# Perturbating and Estimating DSGE Models in Julia

Alvaro Salazar-Perez<sup>1</sup> · Hernán D. Seoane<sup>1</sup> 

Accepted: 8 May 2024  
© The Author(s) 2024

## Abstract

This paper illustrates the power of Julia language for the solution and estimation of Dynamic Stochastic General Equilibrium models. We document large gains of the Julia implementation of Perturbation solution (first and higher orders) and Bayesian estimation using two workhorse models in the literature: the Real Business Cycle Model and a medium scale New-Keynesian Model. We release a companion package that implements 1st, 2nd a 3rd order approximation of Dynamic Stochastic General Equilibrium models and allows for estimation of (log-)linearized models using Sequential Monte-Carlo Methods. Our examples highlight that Julia has low entry costs and it is a language where it is easy to deal with parallelization.

**Keywords** Perturbation solution · Sequential Montecarlo · Julia programming

**JEL Classification** E0 · C63

## 1 Introduction

Dynamic Stochastic General Equilibrium (DSGE) models are part of the standard toolbox of any macroeconomist interested in quantitative analysis and the evaluation of macroeconomic policies. Approximating a solution to this type of models is computationally intensive and when the model is medium to large scale the only possibility is to rely on local approximation methods, such as the Perturbation

---

Hernán Seoane thanks the Agencia Estatal de Investigación for financial support RYC2020-030507-I, PID2019-107161GB-C31 and Ministerio de Asuntos Económicos y Transformación Digital through grant Number ECO2016-76818-C3-1-P. Funding for APC: Universidad Carlos III de Madrid (Agreement CRUE-Madroño 2024).

---

✉ Hernán D. Seoane  
hseoane@eco.uc3m.es  
  
Alvaro Salazar-Perez  
asalazar@eco.uc3m.es

<sup>1</sup> Department of Economics, Universidad Carlos III de Madrid, Calle Madrid 126, 28903 Getafe, Madrid, Spain

method, which provides an accurate approximation to the solution around a neighborhood of the approximating point. Perturbation methods are useful, fast, and accurate, and are applicable as the equations that characterize equilibrium are fully differentiable in the relevant region. Yet, even with perturbation methods, Bayesian estimation using standard packages is time consuming. This is true for the widely used estimation algorithms (such as the Metropolis-Hastings algorithm) and even more so for Sequential Montecarlo Methods, recently introduced for the estimation of macroeconomic models by Herbst and Schorfheide (2014). Moreover, non-linear estimation of DSGE models (requiring nonlinear solver and the particle filter) is still far from generalized in the profession due to computational limitations, even knowing that nonlinearities matter for relevant macroeconomic questions.

In this paper, we show how to design and build a non-linear Perturbation algorithm in Julia and how to use it for Bayesian estimation. We compare the performance in Julia to the one in Python and Matlab. We document that Julia language can help accelerate the solution and the estimation of DSGE models that are suitable for policy analysis, and it has very low entry costs. We develop a package for solving non-linear DSGE models with Perturbation methods up to third order and estimating them with a (log)-linearized solution and a Bayesian approach with Sequential Monte-Carlo.

An important question is “why Julia?”. Julia is one of the languages with fastest growth in Economics and has recently become popular due to the launch of the QuantEcon Project. An inexperienced researcher can use Julia in a similar way to MATLAB while a very experienced user can program in a faster way following guidelines like those of Fortran or C. Hence, Julia has relatively low entry costs compared to other languages. Second, parallelization in CPUs or integration of Julia with CUDA language of Nvidia GPU is straightforward, hence, the potential benefits of using Julia language are enormous. Last but not least, Julia is free, which may boost its adoption worldwide and this may allow for the development of a large community of users. Several resources have recently become available for learning and using Julia, such as Caraiani (2018), the QuantEcon Project or the project for solution and estimation of DSGE models in Julia developed by the New York Fed.<sup>1</sup>

In order to show the gains of Julia we use our package to solve the standard Real Business Cycle Model (RBC), a medium scale New-Keynesian Model (NK) and a large model as in the Smets and Wouters (2007). We document that Julia substantially improves solution and estimation times. In particular, the computation of the log-linear solution, which involves the largest time-consuming step of this method, the Generalized Schur decomposition, for the Real Business Cycle Model in Julia takes about 14  $\mu$ s while in Matlab it takes 128  $\mu$ s, that is about 10% of the time, the second order takes in Julia 4% of the time it takes in Matlab. Julia substantially improves over Python which takes 225  $\mu$ s and 1075  $\mu$ s, respectively. Additionally, the Sequential Montecarlo algorithm for the estimation of that model in Julia can require only 10-15% of the time it takes in Matlab.

<sup>1</sup> <https://github.com/FRBNY-DSGE/DSGE.jl>.

As a by-product of this paper, we develop and release a solution package based on the Matlab package of Schmitt-Grohé and Uribe (2004) that is easily adaptable to any problem where the Perturbation method can be used and it is fully transparent as all functions are accessible directly by the user. Our package is designed to implement first, second and third order solutions. Moreover, the package also includes a simple implementation of Sequential Montecarlo estimation based on the package developed by Herbst and Schorfheide (2014).

Our paper is not the only article that compares numerical approaches, languages or software implementation for the solution or estimation of DSGE models. Aruoba et al. (2006) is one of the first articles doing an ample comparison of solution methods for the standard Real Business Cycle (RBC) model. More recently Aldrich et al. (2011) specifically focuses on the benefits of using the GPU for a value function iteration of a simple RBC model while Aruoba and Fernández-Villaverde (2015) compares various computational languages and Peri (2020) addresses the benefits of cloud computing. Our paper is different from those mentioned before as we quantify the gains of Julia programming under Perturbation methods decomposing its sources, non-linear solutions (i.e., based on perturbation methods) and its estimation with a Bayesian procedure and by providing a friendly environment that is easy to understand and modify.

The rest of the paper goes as follows. In Sect. 2 we describe the baseline models we use to test our package in Julia programming and measure Julia implementation times, the RBC and the NK model. In Sect. 3 we introduce the notation for the Perturbation method, where we heavily draw on the classical presentations in Schmitt-Grohé and Uribe (2004), Aruoba et al. (2006), and Andreasen et al. (2018); and document the gains in different steps of the solution method up to different orders for Julia compared to the implementation of the algorithm in Matlab. In Sect. 4 we introduce the notation for the Sequential Montecarlo Estimation following Herbst and Schorfheide (2014) and document the gains of estimation of log-linearized models. Sect. 5 provides a brief discussion on the implementation of the solution in different languages and about the performance under parallel implementation. Section 6 evaluates different languages for the solution and estimation of the Smets and Wouters (2007) model. Section 7 provides a few concluding remarks. The appendix provides a detailed explanation of the companion solution and estimation pack.

## 2 Models

We solve two models: first, as common practice in the literature, we use as an example the standard Real Business Cycle model (RBC) that has only one endogenous state variable and one exogenous state variable; second, we implement the package for a medium scale New Keynesian model (NK), that includes more endogenous and exogenous predetermined variables. In this section, for completeness, we describe the models as included in the solution and estimation package and map them into the notation of Perturbation methods.

## 2.1 The Real Business Cycle Model

Our first model is the standard Real Business Cycle model, along the lines of the Neoclassical Growth Model discussed in Uhlig (1999) and Schmitt-Grohé and Uribe (2004). Here, we model a representative consumer that maximizes the present discounted value of expected utility and decides optimal consumption, leisure, and investment decisions. The objective function of the representative consumer is given by:

$$\max_{\{c_t, k_{t+1}\}_{t=1}^{\infty}} \mathbb{E}_0 \sum_{t=1}^{\infty} \beta^t \left( \frac{c_t^{1-\gamma}}{1-\gamma} - \omega L_t \right)$$

and the maximization problem is subject to the feasibility constraint of the economy

$$c_t + k_{t+1} = A_t k_t^\alpha L_t^{1-\alpha} + (1 - \delta)k_t$$

and an exogenous process for productivity, that is assumed mean-reverting AR(1) with normal i.i.d. shocks. The problem is standard, capital depreciates at a rate  $\delta$  and produces using a Cobb-Douglas function. The equilibrium conditions are given by the Euler equation, the resource constraint, and the law of motion of exogenous technology shocks,

$$\begin{aligned} c_t^{-\gamma} &= \beta \mathbb{E}_t [c_{t+1}^{-\gamma} (r_{t+1} + 1 - \delta)] \\ \omega &= c_t^{-\gamma} ((1 - \alpha) A_t k_t^\alpha L_t^{1-\alpha}) \\ c_t + i_t &= q_t \\ q_t &= A_t k_t^\alpha L_t^{1-\alpha} \\ k_{t+1} &= (1 - \delta)k_t + i_t \\ r_t &= \alpha A_t k_t^{\alpha-1} L_t^{1-\alpha} \\ \ln(A_{t+1}) &= \rho \ln(A_t) + \sigma \epsilon_{t+1} \end{aligned}$$

for all  $t \geq 0$ , given  $k_0$  and  $A_0$ . To start introducing the notation we need for the Perturbation solution method, these equilibrium conditions can be written as

$$\mathbb{E}_t [f(y_{t+1}, y_t, x_{t+1}, x_t)] = 0$$

where  $f(y_{t+1}, y_t, x_{t+1}, x_t)$  collects all equilibrium conditions equalized to zero,  $x_t = [k_t; \ln(A_t)]$  is a vector of predetermined variables with  $x_t^1 = k_t$  and  $x_t^2 = \ln(A_t)$ , and  $y_t = [c_t; i_t; q_t; r_t; L_t]$  is the vector of non-predetermined variables.

## 2.2 The New-Keynesian (NK) Model

The RBC is the typical model in these types of exercises, but it has a small number of predetermined variables, and we would like to study the way Julia Language deals with larger state spaces. For this reason, we use a medium scale variant of the New-Keynesian model for a closed economy with complete financial

markets and capital accumulation. The economy is populated by households, intermediate goods producers under monopolistic competition, competitive final goods producers and the government that determines the monetary policy and runs a balanced budget. In what follows we introduce each agents' problem.

### 2.2.1 Intermediate Goods Producers

There is a continuum of intermediate goods producers, each of them produces a variety in the interval  $[0,1]$  and given that each firm produces a differentiated good, they have market power and are demand takers. Demand of variety  $i$  is given by:

$$a_{it} = \left( \frac{P_{it}}{P_t} \right)^{-\eta} a_t.$$

This demand depends on its price and the average price of the economy.  $P_{it}$  is a choice variable for the firm.  $\eta$  determines the elasticity of substitutions among varieties and  $a_t$  denotes the aggregate demand. The key friction is that firms cannot change prices with  $\alpha$  probability each period, only  $\alpha$  firms will not be able to change prices. Firms that do not change prices must set prices as  $P_{it} = P_{it-1}$ , that is, we assume no-indexation. These firms maximize expected present discounted value of profits given by

$$\mathbb{E}_0 \sum_{t=0}^{\infty} r_{0,t} P_t \Phi_{it},$$

where  $r_{0,t}$  is the discount rate and their period's  $t$  profits in real terms are

$$\Phi_{it} = \frac{P_{it} a_{it}}{P_t} - u_t k_{it} - w_t h_{it}.$$

Each firm hires labor and capital in competitive markets at price  $w_t$  and  $u_t$ , respectively, to produce using a Cobb–Douglas technology and subject to an aggregate productivity shock,

$$y_{it} = z_t F(h_{it}, k_{it}).$$

Firms in the intermediate goods sector face the constraint that they must serve the demand of its variety, hence:

$$y_{it} \geq a_{it}.$$

Additionally, we assume that the productivity shock follows an AR(1) process in logs,

$$\ln(z_{t+1}) = \rho \ln(z_t) + \sigma \epsilon_{t+1},$$

with normal i.i.d. innovations.

### 2.2.2 Households

Households maximize the present discounted value of utility that depends on consumption varieties and leisure, invest in assets and capital, and own the intermediate firms.

$$\max_{\{c_t, h_t\}_{t=1}^{\infty}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t, h_t),$$

where  $h_t$  denotes labor supply, and the consumption bundle is  $c_t$ .

The households face two intratemporal problems, the leisure-consumption choice, and the choice of varieties. We find the intratemporal allocation between varieties solving a cost minimization problem given by:

$$\min P_t c_t = \int_0^1 P_{it} c_{it} di,$$

subject to the aggregation technology:

$$c_t = \left[ \int_0^1 c_{it}^{1-\frac{1}{\eta}} di \right]^{\frac{1}{1-\frac{1}{\eta}}}$$

with  $\eta$  determining the elasticity of substitution between varieties.

Given this intratemporal allocation, the households solve the dynamic problem by maximizing the present discounted value of utility subject to an infinite sequence of period  $t$  budget constraints:

$$\mathbb{E}_t r_{t,t+1} A_{t+1} + P_t c_t + P_t x_t = P_t w_t h_t + P_t u_t k_t + A_t + P_t \int_0^1 \Phi_{it} di.$$

Here,  $A_t$  denotes a complete set of assets,  $x_t$  denotes investment,  $\Phi_{it}$  are lump-sum transfers of intermediate goods producers' profits and  $r_{t,t+1}$  denotes the price of financial assets. Finally, the evolution of capital is:

$$k_{t+1} = x_t + (1 - \delta)k_t.$$

### 2.2.3 The Government

We assume that the government runs a balanced budget and monetary policy is based on interest rate feedback rules. Hence,

$$\ln\left(\frac{R_t}{R^*}\right) = \phi_R \ln\left(\frac{R_{t-1}}{R^*}\right) + \phi_\Pi \ln\left(\frac{\Pi_{t-1}}{\Pi^*}\right) + \phi_y \ln\left(\frac{s_t(c_t + i_t)}{s^*(c^* + i^*)}\right) e_t^u$$

where  $u_t$  is the monetary policy shock,  $\rho$  captures the persistence of the rule and  $\phi$  captures responsiveness of monetary policy to inflation.

This model, as opposed to the case of the RBC has a larger set of predetermined variables (capital, the cost of price dispersion, inflation, and the nominal interest rate). The equilibrium conditions of this model as well as its notation in terms of the perturbation method is included in the appendix.

### 3 The Perturbation Method

This section presents an overview of the perturbation solution method. A more detailed exposition is in Judd and Judd (1998), Klein (2000), Aruoba et al. (2006) and Schmitt-Grohé and Uribe (2004), among others. The notation and structure of this section draws heavily from Schmitt-Grohé and Uribe (2004) as the solution we implement in Julia uses this notation and method for solving the quadratic matrix equation of the first order approximation.

We focus on models whose solutions can be characterized by a set of equilibrium equations that we can write as

$$\mathbb{E}_t[f(y_{t+1}, y_t, x_{t+1}, x_t)] = 0,$$

where  $\mathbb{E}_t$  is the expectation operator conditional on the information set available at time  $t$ ,  $y_t$  denotes a vector of non-predetermined variables (assumed to be of length  $n_y$ ) and  $x_t$  denotes the vector of states, or predetermined variables (endogenous and exogenous, of length  $n_x$ ). Exogenous predetermined variables are hit by shocks  $\epsilon_t$  (with dimension  $n_\epsilon$ ) that are i.i.d. with mean 0 and variance equal to 1. We introduce the perturbation parameter as  $\sigma$ , a positive scalar that represents the degree of uncertainty of the model and denote  $\eta$  to a matrix  $n_x \times n_\epsilon$ . The solution to this problem would be a set of policies that depend on the degree of uncertainty of the economy

$$y_t = g(x_t, \sigma)$$

and

$$x_{t+1} = h(x_t, \sigma) + \eta \sigma \epsilon_{t+1}$$

The perturbation method allows us to find a local approximation of  $g$  and  $h$  around their approximation points, i.e. the steady state that we denote by  $(\bar{x}, \bar{\sigma})$  with  $\bar{\sigma} = 0$  being the point in which all shocks are shut down. In particular, if  $n_x = n_y = 1$  the Taylor theorem and the Implicit function theorem implies that our solution can be written as:

**Table 1** Parametrization

Parameters	Interpretation	RBC	NK
$\alpha$	Capital exponent in production	0.30	0.30
$\beta$	Discount factor	0.95	1.04–1/4
$\gamma$	CRRA coefficient	2.00	2.00
$\delta$	Capital depreciation	1.00	0.01
$\rho_z$	TFP persistence coefficient	0.90	0.8556
$\sigma_z$	TFP standard deviation	0.05	0.05
$Z$	TFP mean	1.00	1.00
$\eta$	Coeff. of elasticity of subst b/w varieties	–	5.00
$\theta$	Prob. of Calvo lottery	–	0.80
$\varphi_\Pi$	Interest rate response to inflation	–	3.00
$\varphi_Y$	Interest rate response to output	–	0.01
$\varphi_R$	Smoothing coefficient of the MP rule	–	0.80

Parametrization for the RBC and NK model used both in the solution and estimation stages

$$\begin{aligned}
 g(x, \sigma) = & g(\bar{x}, \bar{\sigma}) + g_x(\bar{x}, \bar{\sigma})(x - \bar{x}) + g_\sigma(\bar{x}, \bar{\sigma})(\sigma - \bar{\sigma}) + \frac{1}{2}g_{xx}(\bar{x}, \bar{\sigma})(x - \bar{x})^2 \\
 & + \frac{1}{2}g_{x\sigma}(\bar{x}, \bar{\sigma})(x - \bar{x})(\sigma - \bar{\sigma}) \\
 & + \frac{1}{2}g_{\sigma x}(\bar{x}, \bar{\sigma})(\sigma - \bar{\sigma})(x - \bar{x}) + \frac{1}{2}g_{\sigma\sigma}(\bar{x}, \bar{\sigma})(\sigma - \bar{\sigma})^2 + \dots
 \end{aligned}$$

and

$$\begin{aligned}
 h(x, \sigma) = & h(\bar{x}, \bar{\sigma}) + h_x(\bar{x}, \bar{\sigma})(x - \bar{x}) + h_\sigma(\bar{x}, \bar{\sigma})(\sigma - \bar{\sigma}) + \frac{1}{2}h_{xx}(\bar{x}, \bar{\sigma})(x - \bar{x})^2 \\
 & + \frac{1}{2}h_{x\sigma}(\bar{x}, \bar{\sigma})(x - \bar{x})(\sigma - \bar{\sigma}) + \frac{1}{2}h_{\sigma x}(\bar{x}, \bar{\sigma})(\sigma - \bar{\sigma})(x - \bar{x}) + \frac{1}{2}h_{\sigma\sigma}(\bar{x}, \bar{\sigma})(\sigma - \bar{\sigma})^2 + \dots
 \end{aligned}$$

and with a more involved notation (usually tensor notation) these expressions can be generalized to any  $n_x > 1$  and  $n_y > 1$  as shown in Schmitt-Grohé and Uribe (2004).

Finding the coefficients of these expressions would allow us to approximate them up to any arbitrary order. The first step would be to find the steady state values of predetermined and no-predetermined variables  $\bar{x} = h(\bar{x}, 0)$  and  $\bar{y} = g(\bar{x}, 0)$ . Then, exploiting the implicit function theorem, we need to find the coefficients associated to the first order expansion  $h_x(\bar{x}, 0)$ ,  $g_x(\bar{x}, 0)$ . Schmitt-Grohé and Uribe (2004) show that  $h_\sigma(\bar{x}, 0) = 0$ ,  $g_\sigma(\bar{x}, 0) = 0$ . This is indeed the most time-consuming part of the solution algorithm as it requires solving a quadratic function and is done with a Generalized Schur Decomposition, see Klein (2000). Solving higher order terms is straightforward and only requires solving linear systems of equations.

In order to implement the solution in Julia, we consider an off-the-shelve parametrization for both models, shown in Table 1. As can be seen in the table, the parametrization is standard.



**Table 2** Step-by-step comparison of routines for solution

	RBC			NK		
	Julia ( $\mu$ s)	Python ( $\mu$ s)	Matlab ( $\mu$ s)	Julia ( $\mu$ s)	Python ( $\mu$ s)	Matlab ( $\mu$ s)
Schur Dec & 1st order	14	225	128	67	340	219
Schur Dec, 1st & 2nd order	88	1075	2487	1188	78385	16713

$\mu$ s denotes microseconds. Software specifications: Julia compiler 1.9.3. and Python compiler 3.11.5 in VSCode 1.82.2.; Matlab R2020b. Hardware specifications: AMD(R) Ryzen 7 4800H CPU@2.90 GHz with 32 GB RAM. Times in Matlab are computed as the average of 100 runs using tic-toc command, in Python as the average of 100 runs measured with the macro timeit() and in Julia with the macro @btime

Table 2 presents the computing time required to solve the models in Julia 1.9.3, Python 3.11.5 and Matlab R2020b. We run Julia and Python using Visual Studio Code 1.82.2. As seen in the table, Julia is between three to ten times faster than Matlab in order to solve the first and several orders of magnitude faster than Matlab for second order approximations. The gains do not seem to disappear when increasing the number of predetermined variables. Python is in between Julia and Matlab.

According to this table, we can expect that estimating DSGE models, both linear and nonlinearly, in Julia may be significantly faster. We turn to documenting this fact in the next section.

## 4 Sequential Montecarlo Methods

Even though computing power made Bayesian estimation one of the main instruments for DSGE analysis, several tools can still benefit substantially from better/more novel computational techniques and languages. Herbst and Schorfheide (2014) introduced algorithms based on Sequential Montecarlo (SMC) Methods for the estimation of DSGE models. Similarly, to the Random-Walk Metropolis-Hastings (RWMH), the SMC is a method that allows the researcher to generate draws from posterior distributions of DSGE models. The authors show that the SMC may work better than standard MCMC methods when estimating large scale DSGE models, but they are also more time demanding. Here, we discuss the general algorithm and show the gains of implementing in Julia a simple variant of the SMC combined with a first order approximation to the solution of the RBC and the NK model presented in the previous sections.

The objective of a model estimation in the Bayesian framework is to learn about the shape of the posterior distribution of the parameters of the model using the prior information and the information contained in the data, that is, to learn about  $p(\theta|Y)$  using a model

$$p(\theta|Y) = \frac{p(Y|\theta)p(\theta)}{p(Y)}$$

where  $p(Y) = \int p(Y|\theta)p(\theta)d\theta$  is the marginal density of the data,  $\theta$  is a vector of parameters of interest,  $p(\theta)$  is a prior distribution and  $p(Y|\theta)$  is a likelihood function.

**Table 3** Comparison of estimation routines (3 observables)

	RBC			NK		
	Julia	Python	Matlab	Julia	Python	Matlab
$N_{part}=500, N_{obs}=200$	30.8	1030.0	331.7	73.9	2810.3	360.3
$N_{part}=8192, N_{obs}=200$	427.9	19,820.9	4885.7	1159.4	48,170.5	5959.2
$N_{part}=8192, N_{obs}=500$	1533.1	42,464.2	9018.8	2081.1	985,557.8	9810.9

Measured in seconds. Software specifications: Julia compiler 1.9.3. and Python compiler 3.11.5 in VSCode 1.82.2.; Matlab R2020b. Hardware specifications: AMD(R) Ryzen 7 4800H CPU@2.90 GHz with 32 GB RAM. Times in Matlab are computed as the average of 100 runs using tic-toc command, in Python as the time of a run measured with the macro `timeit()` and in Julia with the macro `@btime`

The main idea behind the SMC is the same as for an importance sampler, where the researcher does not know the right distribution,  $p(\theta|Y)$ , from where to take random draws, but instead generate random draws from a candidate distribution,  $g(\theta)$ , and define weights of the draws by  $w(\theta)$  that approximates the right one. Theoretically this reasoning is based on the following derivations:

$$\mathbb{E}_{\pi}[h(\theta)] = \int h(\theta) \frac{p(Y|\theta)p(\theta)}{p(Y)} d\theta = \frac{1}{p(Y)} \int h(\theta) \frac{p(Y|\theta)p(\theta)}{g(\theta)} g(\theta) d\theta,$$

which is clearly an identity. Here  $\pi = \frac{p(Y|\theta)p(\theta)}{p(Y)}$ , then  $w(\theta) = \frac{p(Y|\theta)p(\theta)}{g(\theta)}$  operate as weights of the particles (draws). Hence a collection of particles and weights allow us to obtain an approximation to the right distribution.

In practice, finding a good density from where to take draws,  $g(\theta)$ , is very hard and as can be seen, it is the key step of the method. The SMC is a method to construct an important sampling function in a recursive fashion. The algorithm starts by producing particles from an easy to sample initial distribution and finishes with a distribution that produces posterior draws that approximate those of the model of interest. Each particle has an associated importance weight that is ultimately used to construct posterior distributions. The algorithm operates such that the initial set of particles are mutated in each step using a Metropolis–Hastings algorithm.

Our exercise is to estimate both models presented in the previous sections: the RBC and the NK model. To this end, with the calibration presented in Table 1, we simulate the models for 10,000 periods and keep the last 500 periods (or 200 periods) as an estimation sample for three observables. Then, using those observables, we estimate the persistence and volatility of the shock of the mode using different numbers of particles: 500 and  $2^{13}$ . Our priors are a Truncated Normal for the persistence of the technology shock and an Inverse Gamma for the standard deviation. We do the estimation imposing measurement errors as the number of observables is larger than the number of shocks.

In Table 3 we present the time required to estimate both the RBC and the NK models using SMC with different sample sizes and different number of particles.

As seen in the table, and expected from the previous section, the gains from Julia are substantial. We must point out that significant gains are attained when we use the

MKL package for Linear Algebra which allows Julia to handle matrix objects in an efficient way.

The table shows that Julia takes 10% of the time that Matlab takes, which is also better than Python, for the case of low number of particles and observations.

## 5 Discussion, Packages and Technicalities

### 5.1 Evaluation and Numerical Implementation

A few words about the implementation of the routine in Matlab and Julia are important at this point. First, the solution in Matlab does not need the use of the *eval()* command, the approximation package by Schmitt-Grohé and Uribe (2004) prints the matrices required for the QZ decomposition and load them directly as scripts, this step makes the algorithm faster than the use of the evaluation command from Matlab, which substantially boosts the estimation times shown in Sect. 4. The *eval()* function in Julia also relents the algorithm substantially and we managed to reduce it to one evaluation only.

Following the implementation in Matlab of the package by Schmitt-Grohé and Uribe (2004), we make extensive use of the symbolic toolbox in order to obtain the steady state expressions and the derivative matrices of each model in terms of the parameters. However, contrary to the quoted package, we store them as strings concatenated with lines declaring a function with the parametrization as input, and then, we immediately parse these strings to convert them into Expr type objects. Thus, after a single evaluation of these objects we obtain two functions specifically tailored for the model which compute the steady state values and the derivative matrices for any given parametrization. The recurring use of this function prevents calling to the *eval()* function or loading a script in every step, significantly reducing the execution time of the algorithm. We follow the same procedure to equivalently accelerate the execution in Python.

Second, to our best knowledge Matlab deals efficiently with matrices using MKL. For the solution stage, Julia is faster than Matlab even without this package, but the package is also available in Julia, including it -that also boosts matrix handling in Julia- allow us to attain the gains depicted in the table, and will be responsible for the gains of the estimation stage. We have also included this package in Python for comparability, obtaining also significant gains in time for this language.

### 5.2 Parallel Implementation

We consider the estimation of the RBC parallelizing in the CPU. An advantage of SMC methods vis-a-vis MH is that at each step of the estimation process, the algorithm needs to evaluate the model for many particles, a task that can be parallelized. The package “Threads” is the simplest to parallel the SMC estimation. We find that estimation with 200 periods, for  $n_\varphi = 100$ ,  $n_{part} = 500$  with one thread takes 39.9 s,

with 11 threads, the time required for estimation is 19.9 s in a Julia compiler 1.9.3. in Visual Studio Code 1.82.2; Matlab R2020b with the following hardware specifications: AMD(R) Ryzen 7 4800H CPU@2.90 GHz with 32 GB RAM.

## 6 The Smets and Wouters (2007) Model

As a final test we consider a larger model that is along the lines of the models used in Central Banks for policy analysis, the model in Smets and Wouters (2007). This model is a large scale DSGE model for the US economy that incorporates many types of real and nominal frictions. Particularly, the authors consider a closed economy with nominal price stickiness, wage setting frictions, habit formation in consumption, capital utilization, production subject to fixed costs and investment adjustment costs. These frictions are meant to help the model fit the data and have hump shaped responses consistent with empirical studies. On the drivers of the business cycle, the model includes seven shocks. Real “supply side shocks” include a total factor productivity shock and wage markup shock; “demand side” shocks include government spending shock, risk premium shock, investment specific technological shock; in turn, nominal shocks include price mark-up shocks and a monetary policy shock. A full presentation of the model is beyond the scope of this paper and will not contribute to our objectives. We recommend those interested in the modelling details to look at the Smets and Wouters (2007) article.

For this model we follow the standard practice of introducing the equilibrium conditions in a log-linearized form, which only allows for comparison of the solution up to a first order approximation. Nevertheless, remember that the first order approximation is the costliest as it is the step that requires solving a quadratic matrix function, higher order terms are obtained by solving linear systems of equations.

As in the previous sections, we estimate this model using SMC with different number of particles and different sample size using a simulated dataset. Table 4 presents the solution and estimation times of Julia, Python and Matlab.

As Table 4 shows, even for large models Julia outperforms Matlab. For the QZ decomposition and the first order terms, Julia is more than 20% faster than Matlab and twice as fast as Python. Moreover, the estimation of this model is about twice as fast in Julia than in Matlab both for small and large numbers of particles.

## 7 Concluding Remarks

This paper shows that, even though Perturbation methods are fast and accurate approximations to the policy functions of non-linear models, the implementation in Julia language can reduce solution times in several orders of magnitude, not only for linear approximations, but also for non-linear terms. We also show that Julia attains large gains when we use perturbation method combination with frontier estimation methods.

Moreover, using Julia, computing in parallel and integration with CUDA is straightforward, potentially leading to further reductions in estimation times.

Our paper includes a companion repository with the codes in Julia Language. The codes are general for the solution of first, second and third order Perturbation, simulation, and production of impulse response functions. Additionally, we include the codes for SMC estimation. We implement the solution in Visual Studio Code, Julia 1.9.3. To build this package we translated to package by Schmitt-Grohé and Uribe (2004) for the first and second order perturbation and extended them to third order using Andreasen et al. (2018). For the SMC algorithm we translated the simple example algorithm in the replication package of Herbst and Schorfheide (2014). The package for solving and estimating DSGE models in Julia is available at: <https://github.com/HernanSeo/JuliaPerturbation>

## Appendix 1: New-Keynesian Model

The equilibrium conditions of the New-Keynesian model in its standard form with CRRA preferences are given by,

Use  $\tilde{p}_t = \frac{\tilde{P}_t}{P_t}$  and  $\Pi_t = \frac{P_t}{P_{t-1}}$ . Collect optimality conditions:

$$\begin{aligned}
 s_t &= (1 - \alpha)\tilde{p}_t^{1-\eta} + \alpha\Pi_t^\eta s_{t-1} \\
 z_t h_t^{1-\alpha} k_t^\alpha &= s_t(c_t + i_t) \\
 1 &= (1 - \alpha)\tilde{p}_t^{1-\eta} + \alpha\Pi_t^{\eta-1} \\
 x_t^1 &= \tilde{p}_t^{1-\eta} a_t \frac{\eta-1}{\eta} + \alpha\mathbb{E}_t r_{t,t+1} \left( \frac{\tilde{p}_t}{\tilde{p}_{t+1}} \right)^{1-\eta} \Pi_{t+1}^\eta x_{t+1}^1 \\
 x_t^2 &= \tilde{p}_t^{-\eta} a_t m c_t + \alpha\mathbb{E}_t r_{t,t+1} \left( \frac{\tilde{p}_t}{\tilde{p}_{t+1}} \right)^{1-\eta} \Pi_{t+1}^{\eta+1} x_{t+1}^2 \\
 x_t^2 &= x_t^1 \\
 u_t &= m c_t z_t F_k(h_t, k_t) \\
 w_t &= m c_t z_t F_k(h_t, k_t) \\
 \frac{1}{R_t} &= \mathbb{E}_t r_{t,t+1} \\
 u_c'(c_t, h_t) &= \lambda_t \\
 \frac{u_h'(c_t, h_t)}{u_c'(c_t, h_t)} &= w_t \\
 \lambda_t r_{t,t+1} &= \beta \left[ \lambda_{t+1} \frac{P_t}{P_{t+1}} \right] \\
 \lambda_t &= \beta \mathbb{E}_t [(u_{t+1} + 1 - \delta) \lambda_{t+1}] \\
 k_{t+1} &= (1 - \delta)k_t + i_t
 \end{aligned}$$

Additionally, we consider the following productivity process and Taylor Rule:

$$\ln(z_{t+1}) = \rho \ln(z_t) + \sigma \epsilon_{t+1}$$

$$\ln\left(\frac{R_t}{R^*}\right) = \phi_R \ln\left(\frac{R_{t-1}}{R^*}\right) + \phi_\Pi \ln\left(\frac{\Pi_{t-1}}{\Pi^*}\right) + \phi_y \ln\left(\frac{s_t(c_t + i_t)}{s^*(c^* + i^*)}\right)$$

We consider a deterministic steady state, setting  $\Pi^*$  as the inflation target and We have normalized the steady-state value of productivity to  $z^* = 1$ . Therefore, from the price evolution equation:

$$\tilde{p}^* = \left( \frac{1 - \alpha \Pi^{*(\eta-1)}}{1 - \alpha} \right)^{\frac{1}{1-\eta}}$$

And thus, the cost of price dispersion is:

$$s^* = \frac{(1 - \alpha)\tilde{p}^*}{1 - \alpha \Pi^{*\eta}}$$

From the remaining equilibrium conditions and assuming the functional forms  $F(h_t, k_t) = h_t^{1-\theta} k_t^\theta$  and  $u(c_t, h_t) = \frac{[c(1-h)^\gamma]^{1-\xi} - 1}{1-\xi}$ , we get:

$$r^* = \frac{\beta}{\Pi^*}$$

$$R^* = \frac{\Pi^*}{\beta}$$

$$u^* = \frac{1}{\beta} - 1 + \delta$$

$$mc^* = \frac{1 - \alpha r^* \Pi^{*(\eta+1)}}{1 - \alpha r^* \Pi^{*\eta}} \frac{\eta - 1}{\eta} \tilde{p}^*$$

$$h^* = \frac{1 - \theta}{\frac{\gamma}{s^* mc^*} + 1 - \theta - \delta \frac{\theta}{u^*}}$$

$$k^* = \left( \frac{\theta mc^* z^*}{u} \right)^{\frac{1}{1-\theta}} h^*$$

$$w^* = mc^* z^* (1 - \theta) \left( \frac{k^*}{h^*} \right)^\theta$$

$$i^* = \delta k^*$$

$$c^* = \frac{z^* h_t^{1-\theta} k_t^\theta}{s^*} - i^*$$

$$x^{1*} = \frac{1}{1 - \alpha r^* \Pi^{*\eta}} \frac{\eta - 1}{\eta} c^* \tilde{p}^{*(1-\eta)}$$

$$x^{2*} = x^{1*}$$

$$\lambda^* = c^{*- \xi} (1 - h^*)^{1-\xi}$$

**Table 4** Comparison of routines for solution of the Smets and Wouters (2007) model

	Julia	Python	Matlab
Schur Dec & 1st order terms ( $\mu$ s)	746.5	1596.3	911.5
Estimation (Npart=500, Nobs=200) (s)	292.4	16,019.9	419.9
Estimation (Npart=8192, Nobs=200) (s)	4490.4	262,469.8	7206.9

$\mu$ s denotes microseconds. Software specifications: Julia compiler 1.9.3. in VSCode 1.82.2.; Matlab R2020b. Hardware specifications: AMD(R) Ryzen 7 4800H CPU@2.90 GHz with 32 GB RAM. Times in Matlab are computed as the average of 100 runs using tic-toc command, in Python as the time of a run measured with the macro timeit() and in Julia with the macro @btime

## Appendix 2: The codes

The companion package includes the following *.jl* files:

1. *run\_solution\_xxxx*: Launches the solution of a model.
2. *run\_estimation\_xxxx*: Launches the Bayesian estimation of a model through the Sequential Monte Carlo (SMC) method.
3. *solution\_functions*: Repository with all the functions required to solve and simulate the model.
4. *smc\_rwmh*: Repository with all the functions required to estimate the model through the SMC method.

### Solution and Simulation of the Model

In order to adapt the codes to solve and simulate a specific model the following modifications must be conducted in the file *run\_solution\_xxxx.jl*:

---

```
## Model ##
# Settings #
flag_order=          # State the order of the estimation (1,2,3)
flag_deviation=      # State whether the solution of the model should be log – linearized
                      (true) or not (false)

# Parameters #
@vars                # List all the parameters in the model to generate them as symbols
parameters=[:];      # List all the parameters in the model in a column vector
estimate=[]           # Since no parameter should be estimated, leave this field as an empty
                      vector
priors=[]             # Since no parameter should be estimated, leave this field as an empty
                      vector

# Variables #
@vars                # List all the variables in the model in order to generate them as
                      symbols
x=[:];               # List all the states in the model at time t as a column vector
y=[:];               # List all the jumpers in the model at time t as a column vector
```

---

---

<code>xp = [];</code>	# List all the states in the model at time $t + 1$ as a column vector
<code>yp = [];</code>	# List all the jumpers in the model at time $t + 1$ as a column vector
# Shock #	
<code>@vars</code>	# List all the exogenous shocks in the model at time $t$ in order to generate them as symbols
<code>e = [];</code>	# List as a column vector all the exogenous shocks affecting the model at time $t$
<code>eta = Array([])</code>	
# Equilibrium conditions #	
<code>f1 =</code>	# Write each of the equilibrium conditions as a symbolic function
<code>f2 =</code>	
<code>...</code>	
<code>f = [ f1; f2; f3]</code>	# List all the equilibrium conditions as a column vector
# Steady state (OPTIONAL) #	
<code>SS = [];</code>	# If known, provide the symbolic expression of the steady state in terms of the parameters. It must be provided as a column vector, in which the variables should be ordered in the following way: [ $x$ ; $y$ ; $xp$ ; $yp$ ]. If the steady state is unknown, leave it as an empty vector ( $SS = []$ ), so that the program tries to estimate it numerically
## Solution ##	
# Parametrization #	
<code>PAR = [];</code>	# List the value taken by all the parameters of the model as a column vector. The order should coincide with that of the previously declared "parameters" vector

---

## Estimation

In order to adapt the codes of the estimation algorithm to a specific model the following modifications must be conducted in the file *run\_estimation\_xxxx.jl*:

---

## Model ##	
# Settings #	
<code>flag_order =</code>	# State the order of the estimation (1,2,3)
<code>flag_deviation =</code>	# State whether the solution of the model should be log – linearized (true) or not (false)
# Parameters #	
<code>@vars</code>	# List all the parameters in the model to generate them as symbols
<code>Parameters = [];</code>	# List all the parameters in the model in a column vector
<code>Estimate = []</code>	# List the parameters that need to be estimated in the model as a column vector
<code>Priors = []</code>	# List the priors of the parameters that need to be estimated in the model as a column vector
# Variables #	
<code>@vars</code>	# List all the variables in the model in order to generate them as symbols

---



---

<code>x = []</code>	<code># List all the states in the model at time t as a column vector</code>
<code>y = []</code>	<code># List all the jumpers in the model at time t as a column vector</code>
<code>xp = []</code>	<code># List all the states in the model at time t + 1 as a column vector</code>
<code>yp = []</code>	<code># List all the jumpers in the model at time t + 1 as a column vector</code>
<code># Shock #</code>	
<code>@vars</code>	<code># List all the exogenous shocks in the model at time t in order to generate them as symbols</code>
<code>e = []</code>	<code># List as a column vector all the exogenous shocks affecting the model at time t</code>
<code>eta = Array{ [] }</code>	
<code># Equilibrium conditions #</code>	
<code>f1 =</code>	<code># Write each of the equilibrium conditions as a symbolic function</code>
<code>f2 =</code>	
<code>...</code>	
<code>f = [ f1; f2; f3 ]</code>	<code># List all the equilibrium conditions as a column vector</code>
<code># Steady state (OPTIONAL) #</code>	
<code>SS = []</code>	<code># If known, provide the symbolic expression of the steady state in terms of the parameters. It must be provided as a column vector, in which the variables should be ordered in the following way: [ x; y; xp; yp]. If the steady state is unknown, leave it as an empty vector (SS = []), so that the program tries to estimate it numerically</code>
<code>## Estimation ##</code>	
<code># Parametrization #</code>	
<code>PAR = []</code>	<code># List the initial value of the all the parameters of the model as a column vector. The order should coincide with that of the previously declared "parameters" vector</code>
<code>estimation_results</code>	<code>= smc_rwmh( model, data, PAR, PAR_SS, eta, SS, deriv, sol_mat, c, npart, nphi, lam)</code>
	<code># This line initializes the SMC estimation method. The data should be previously loaded in the variable 'data' as column time series. 'npart' corresponds to the number of particles and 'nphi' to number of periods and can be willingly adjusted</code>

---

**Funding** Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. We thank financial support by PID2019-107161GB-C31/AEI/10.13039/501100011033 and Ministerio de Asuntos Económicos y Transformación Digital through grant Number ECO201676818-C3-1-P. Funding for APC: Universidad Carlos III de Madrid (Agreement CRUE-Madroño 2024).

## Declarations

**Conflict of interest** Alvaro Salazar-Perez and Hernán D. Seoane have neither financial nor non-financial interests conflicting with the current manuscript.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and

your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Aldrich, E. M., Fernández-Villaverde, J., Gallant, A. R., & Rubio-Ramírez, J. F. (2011). Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, 35, 386–393.
- Andreasen, M. M., Fernández-Villaverde, J., & Rubio-Ramírez, J. F. (2018). The pruned state-space system for non-linear DSGE models: Theory and empirical applications. *The Review of Economic Studies*, 85, 1–49.
- Aruoba, S. B., & Fernández-Villaverde, J. (2015). A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58, 265–273.
- Aruoba, S. B., Fernandez-Villaverde, J., & Rubio-Ramirez, J. F. (2006). Comparing solution methods for dynamic equilibrium economies. *Journal of Economic Dynamics and Control*, 30, 2477–2508.
- Caraiani, P. (2018). *Introduction to quantitative macroeconomics using Julia: From basic to state-of-the-art computational techniques*. Academic Press.
- Herbst, E., & Schorfheide, F. (2014). Sequential Monte Carlo sampling for DSGE models. *Journal of Applied Econometrics*, 29, 1073–1098.
- Judd, K. L., & Judd, K. L. (1998). *Numerical methods in economics*. MIT Press.
- Klein, P. (2000). Using the generalized Schur form to solve a multivariate linear rational expectations model. *Journal of Economic Dynamics and Control*, 24, 1405–1423.
- Peri, A. (2020). A hardware approach to value function iteration. *Journal of Economic Dynamics and Control*, 114, 103894.
- Schmitt-Grohé, S., & Uribe, M. (2004). Solving dynamic general equilibrium models using a second-order approximation to the policy function. *Journal of Economic Dynamics and Control*, 28, 755–775.
- Smets, F., & Wouters, R. (2007). Shocks and frictions in US business cycles: A Bayesian DSGE approach. *American Economic Review*, 97, 586–606.
- Uhlig, H. (1999). A toolkit for analysing nonlinear stochastic models easily. In *Computational methods for the study of dynamic economics*, Marimon and Scott eds.

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