A comparison of programming languages for bayesian inference

Filipe Stona[a,b]

[a]*Doutorado em Economia Aplicada - Artigo para a disciplina de Econometria III (Prof. Flávio Ziegelmann)*
[b]*Department of Economics, Federal University of Rio Grande do Sul (UFRGS)*

Abstract

This paper aims to compare the performance of three scripting programming languages –`Matlab`, `R` and `Julia` – for Bayesian inference, more precisely using a Random Walk Metropolis-Hastings algorithm.

Keywords: MCMC, Metropolis-Hastings, Programming Languages.

1. Introduction

Computational statistics has become fundamental in different fields of studies and economics is one of them. Despite the development of computing power, statistical methods are also getting computationally intensive. The memory efficiency and time-consuming of such exercises are close related to the programming language used to solve them. It is widely known that lower-level languages, such as `C`, `C++` and `Fortran`, are faster than interpreted or script languages, however, there is a trade-off between the time written codes and computing time. As Prechelt (2000) highlights, scripting languages are reasonable alternative to these conventional languages, since their higher run time is compensated with programmer productivity.

This paper aims to compare the performance of three scripting programming languages –`Matlab`, `R` and `Julia` – for Bayesian inference, more precisely using a Random Walk Metropolis-Hastings algorithm. For Bayesian inference, the computational burden is one of its major drawbacks, turning some problems prohibitively. The Random Walk Metropolis-Hastings Algorithm, our main focus, is based on a loop process, which used to be slow in interpreted languages. For this reason, it is interesting to compare performance of these languages on the Bayesian estimation.

There are few studies with this purpose in the literature. Prechelt (2000) compares seven programming languages in several aspects of each language, including program length, programming

Email address: `fstona@live.com` (Filipe Stona)

effort, runtime efficiency, memory consumption, and reliability. More specifically, we follow the main idea of Aruoba and Fernández-Villaverde (2015), who compare programming languages for macroeconomics, however focusing on fewer languages. All codes for this paper are available at https://github.com/fstona/bayesian-econ3.

2. A Simulation Exercise

We follow Koop (2003, ch. 5) example for this simulation exercise. Considering a nonlinear function such as a Constant Elasticity of Substitution (CES) function, we applied a nonlinear regression model of the form:

$$y_t = \left(\sum_{j=1}^{k} \gamma_j x_{t,j}^{\gamma_{k+1}} \right)^{\frac{1}{\gamma_{k+1}}} + \varepsilon_t, \tag{2.1}$$

where $x_{t,j} = 1 \quad \forall t = 1, \ldots, T,$ and $j = 1$ represents the intercept, $\varepsilon \sim N(0_T, h^{-1} I_T)$, and $h = 1/\sigma^2$ is the error precision. A generic notation for this nonlinear regression can be introduced as:

$$y_t = f(X, \Gamma) + \varepsilon_t, \tag{2.2}$$

given that $\gamma_j \in \Gamma$ and $x_{t,j} \in X$.

Considering a multivariate Normal distribution, the likelihood function of this model is

$$p(y|\Gamma, h) = \frac{h^{T/2}}{(2\pi)^{T/2}} \left\{ \exp\left[-\frac{h}{2} \{y - f(X,\Gamma)\}'\{y - f(X,\Gamma)\} \right] \right\}. \tag{2.3}$$

Recalling $p(\Gamma, h)$ as the prior density, we can write the posterior density as

$$p(\Gamma, h|y) \propto p(\Gamma, h) \frac{h^{T/2}}{(2\pi)^{T/2}} \left\{ \exp\left[-\frac{h}{2} \{y - f(X,\Gamma)\}'\{y - f(X,\Gamma)\} \right] \right\}. \tag{2.4}$$

Assuming prior independence among parameters and since we cannot analytical posterior results, the Metropolis-Hastings algorithm is a applicable solution to access posterior simulations.

2.1. The Random Walk Metropolis-Hastings Algorithm

As described in Robert and Casella (2004), a Metropolis-Hastings algorithm starts with the objective density. A conditional density, or candidate generating density, $q(\theta^{i-1}|\theta)$ defined with respect to the dominating measure for the model, is then chosen. The Metropolis-Hastings algorithm can be implemented in practice when $q(\cdot|\theta)$ is easy to simulate from and is either explicitly available or symmetric.

The Random Walk chain is a common option and practical implementation of Metropolis-Hastings algorithm when you cannot find a good approximation density for the posterior. In this specification, rather than attempt to approximate the posterior, the candidate generating density is chosen to take draws proportionally in various regions of the posterior.

Formally, taking θ as a vector of parameters and regarding $\Gamma \subseteq \theta$, the Random Walk Metropolis-Hastings algorithm (RWMH) takes the following steps:

1. Chose starting value, θ^0;
2. Take a candidate draw, ϑ from the candidate generating density, $q(\vartheta|\theta^{i-1})$;
3. Calculate the acceptance probability, $\alpha(\theta^{i-1}, \vartheta)$;
4. Set $\theta^i = \vartheta$ with probability $\alpha(\theta^{i-1}, \vartheta)$ and $\theta^i = \theta^{i-1}$ with probability $1 - \alpha(\theta^{i-1}, \vartheta)$;
5. Repeat steps 2 to 4 s times.

The RWMH has a proposal distribution with a random walk form, so it generates candidates draws according to

$$\vartheta = \theta^{i-1} + \eta, \tag{2.5}$$

where η is the increment random variable with mean zero and variance cV. We use a multivariate normal proposal distribution, following great part of the literature. The covariance matrix cV is defined as $V = \widehat{var(\theta_{ML})}$ and $c \in [0.2, 3]$ is adjusted to ensure a "reasonable" acceptance rate.

The acceptance probability $\alpha(\cdot, \cdot)$ ensure that the chain moves in the correct direction. Giving the symmetric nature of the proposal distribution the acceptance probability takes the form

$$\alpha(\theta^{i-1}, \vartheta) = \min\left\{\frac{p(\vartheta|y)}{p(\theta^{i-1}|y)}, 1\right\}, \tag{2.6}$$

so, a draw ϑ is accepted with probability one if the posterior at ϑ has higher value than the posterior at θ^{i-1}. Considering the ratio between accepted draws and total draws, we have the acceptance rate. As Gamerman and Lopes (2006) and Herbst and Schorfheide (2015) mention, there is no consensus on the literature for the optimal acceptance rate, suggesting, as a generic rule, an acceptance rate around 24%. The is conditioned to the cV matrix, so before we run the RWMH algorithm, we test different c values during a warm up period $s_w = 0.25s$ until we have a plausible acceptance rate between 0.17 and 0.25, considering each c in a sequence $c_0 = \{0.2, 0.4, \ldots, 3\}$.

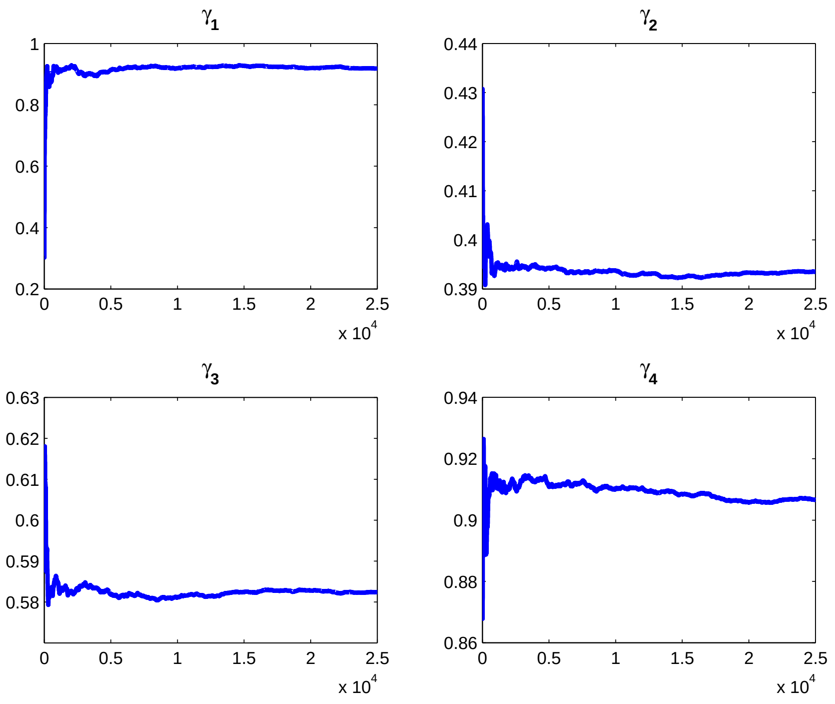
3. Selection of Programming Languages

As noted in the TIOBE Index of programming language popularity and the IEEE Spectrum Ranking, there are hundred of programming languages used for a wide variety of purposes. Since we do not have a specific ranking by field of study, so we could select the most used languages in economics, our decision is based on common knowledge and experience. First, we decide to chose among scriting languages, because, even though complied languages, such as `C++` and `Fortran` are remarkably powerful options, they are also harder to code, and only used in highly computationally complex studies. Scripting languages are easier to code and learn, being usual entrance languages for students of economics. Both `R` and `Matlab` are typical languages for statistical computation and are in the Top 10 Programming Languages for scientific applications by IEEE Spectrum.

Furthermore, it is hard to say that exists a "common language" for computing economics problems. Just in the October/2017 edition of American Economic Review, i.e., we have five different softwares used in the papers (including `R` and `Matlab`). On the other hand, `Julia` is a new open-source high-performance scripting programming language with a syntax very close to `Matlab`'s. Beyond being a language with few users in economics, Aruoba and Fernández-Villaverde (2015) highlight `Julia`'s outstanding performance in the solution of a stochastic neoclassical growth model, being the fastest scripting language test by the authors.

4. Results

We use simulated data, in a way we know the correct parameters value. Regarding our CES nonlinear regression, Eq. 2.1, we chose k = 3, T = 200 and $\Gamma = vec(0.90, 0.40, 0.60, 0.85)$. For $j = 2, 3$, $x_{t,j}$ commonly denotes production factors (capital and labor, i.e.) and cannot assume negative values. So we generate them from a Gamma distribution, $x_{2,t} \sim Gamma(10, 1)$ and $x_{3,t} \sim Gamma(5, 1)$. Finally, we simulate $\varepsilon_t \sim N(0, 1)$. After generate simulated data, we construct the candidate density for MH Algorithm. We compute the OLS estimates of parameters $\gamma_{1,2 \text{ and } 3}$ and consider $\gamma_4 = 1$ as initial values for the optimization of the likelihood. Than, the resulting parameters and inverse Hessian are used to initialize the RWMH algorithm. We simulate $s = 50,000$ draws of the RWMH algorithm, with a burn-in period of $s_0 = 25,000$, and after s simulations we take the average draw for each parameter and compare with the actual Γ. We also compute the running time of the simulation process until the ML estimation and the total time for the MH



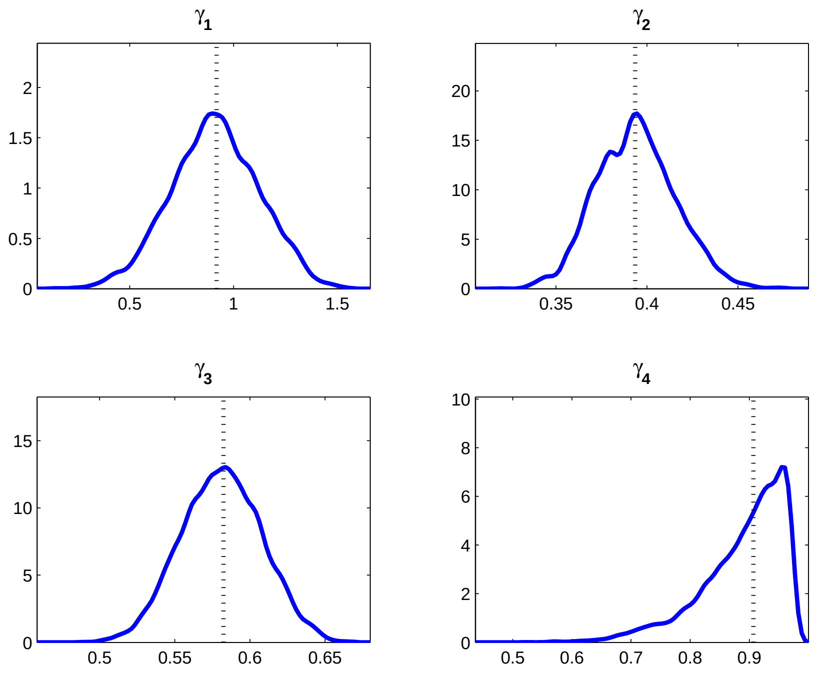
Note: Recursive mean after the burn-in period for the four parameters.

Figure 1: Recursive Averages

algorithm. Considering that we are using simulated data with different random number simulators, we repeat this procedure 20 times.

Before the comparison of performance, we conduct an example exercise with the same simulated variables, so we could verify that all languages are with correct results. Figure 1 depicts the recursive average $\bar{\Gamma}_{s|s_0} = \frac{1}{s-s_0}\sum_{i=s_0+1}^{s}\Gamma^i$. For all parameters, the recursive mean seems to approach the a limit point with stationarity. Figure 2 illustrates the posterior marginal densities for $s - s_0$ draws. It is important to highlight that we impose a boundary condition on the γ_4 parameters, refusing draws with $\gamma_4 > 1$, justifying the asymmetric density observed in the plot.

The main result for time performance is represented in Table 1. The simulation time is how long the program took to simulate data and optimize, while th MH time is the total time in seconds to run the algorithm for s draws simulations. In both cases, simulation and MH time, `Julia` demonstrates better performance than `R` and `Matlab`. The RWMH estimation in `Julia` results a five times fester computation than `Matlab` and times comparing with `R`. The resulting



Note: Dotted line represents the mean of each parameter posterior draws.

Figure 2: Posterior Marginal Densities

performance of `Julia` compared with `Matlab` is similar with Aruoba and Fernández-Villaverde (2015) results. However, the authors found that `Julia` performed around 100x faster than `R`, while this discrepancy is tighter in our estimation, since `R` perform closed to `Matlab`.

Finally, we present accuracy results in Table 2. We compare the estimated parameters with their actual values, and after repeat this process 20 times we compare the average error for each parameter both in the ML estimation and the MH estimation. Table xx also presents the mean number of iterations of the optimization routine[1]. We need to be careful analyzing this table, since, as we already mention, each simulated dataset is different. The estimation accuracy of the MH algorithm tend to be grater than to one estimate with ML, which is expected, since the ML results initialize the MH algorithm. However, it is intereseting to observe that results for $\hat{\gamma}_1$ are inaccurate for all languages, getting away of the actual result during the MH algorithm in some cases.

[1] `optim` in `R`, `fminunc` in `Matlab`, and `Optim.jl` in `Julia`.

	simulation time (s)				mean MH time (s)			
	Mean	St. Dev.	Min	Max	Mean	St. Dev.	Min	Max
Matlab	0.1004	0.0786	0.0399	0.3497	33.9087	7.0895	20.4313	49.0214
R	0.03893	0.01144	0.03002	0.08305	32.0463	2.57439	27.2064	35.7411
Julia	0.06626	0.95988	0.02613	0.17072	6.61259	0.62956	6.11333	8.57663

Table 1: Running Time

	Matlab				**R**				**Julia**			
	ML estim.		MH estim.		ML estim.		MH estim.		ML estim.		MH estim.	
	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
$\hat{\gamma_1} - \gamma_1$	0.149	0.230	0.136	0.171	1.165	0.256	1.197	0.276	-0.196	1.321	-0.394	1.443
$\hat{\gamma_2} - \gamma_2$	-0.026	0.033	-0.016	0.018	0.006	0.026	-0.011	0.034	0.120	0.953	0.065	0.961
$\hat{\gamma_3} - \gamma_3$	0.013	0.058	0.001	0.032	0.004	0.031	0.019	0.037	0.154	0.921	0.096	0.929
$\hat{\gamma_4} - \gamma_4$	0.104	0.199	0.038	0.045	-0.021	0.069	0.082	0.173	0.148	0.936	0.044	0.984
niter	21.00	4.31			8.00	1.38			11.00	2.50		

Table 2: Estimation Errors

5. Final Remarks

In this short paper we have taken a first step at a comparison of programming languages in Bayesian inference. We could observe that `Julia` is a new option for the solution of this problems, since it has a close syntax to `Matlab`'s and `R`'s with faster computing time. Further research might include other languages, such as `Python`, and explore more complex exercises, such as the estimation of a small scale New-Keynesian Model.

References

Aruoba, S. B., and J. Fernández-Villaverde (2015): "A comparison of programming languages in macroeconomics," *Journal of Economic Dynamics and Control*, 58, 265 – 273.

Gamerman, D., and H. F. Lopes (2006): *Markov Chain Monte Carlo: stochastic simulation for Bayesian inference.* Chapman & Hall/CRC, Boca Raton, FL, 2 edn.

Herbst, E. P., and F. Schorfheide (2015): *Bayesian Estimation of DSGE Models.* Princeton University Press.

Koop, G. (2003): *Bayesian Econometrics.* John Wiley & Sons.

Lubin, M., and I. Dunning (2015): "Computing in Operations Research Using Julia," *INFORMS Journal on Computing*, 27(2), 238–248.

Prechelt, L. (2000): "An empirical comparison of seven programming languages," *Computer*, 33(10), 23–29.

Robert, C., and G. Casella (2004): *Monte Carlo Statistical Methods*, Springer Texts in Statistics. Springer, New York, 2 edn.