DSGE in Julia

Gilberto Boaretto & Daniel Coutinho

December, 2019

1 Introduction

We implement a solver and estimation procedure for Dynamic Stochastic General Equilibrium (DSGE) models using Julia, a modern programming language focused on scientific computing. We show all of this working for a toy model, from Galí (2009). This allows us to test throughout every step of the code and show that it is sound. This is a rather large enterprise and the final code relies on a number of packages and files. To help the reader, we give an overview of what we did before dwelling into details:

1. We need to take a log linearised DSGE, which depends on expectations of future values, and write it in a form that allow us to take it into the data. *Gensys*, as described in Sims (2002) allow us to do that and is therefore the first step.

2. We need to write, given a set of parameters and some data, the likelihood. This is not trivial since the model is a VARMA. The Kalman Filter allows us to recover the fundamental shocks from the set of observables.

3. The last step is the estimation itself. Due to the large number of parameters and the many restrictions in the parameter space, DSGE are usually estimated by Bayesian methods. We follow this approach and implement the Random Walk Metropolis Hasting (RWMH) algorithm to sample from the posterior. We also estimate using a more modern method for Markov Chain Monte Carlo (MCMC), namely Hamiltonian Monte Carlo (HMC).

Although there is a large literature on solving and estimating DSGEs, tricks that are important from the computational point of view - many regarding floating point arithmetic - are usually not covered. We discuss a number of them and many "work around" that make algorithms work smoothly.

This paper is composed as follows: the first section is this short introduction; the second one explains the model; the third explains the implementation and its various sub components. This section also show each of the components at work with the toy model and provides a quick explanation of HMC and some simulation results comparing RWMH and HMC. The last section concludes.

Before we start, let us give a short note on developing something like this: this is a large project, with many pieces that can potentially go wrong for a number of reasons - the first is, of course, human error. We are careful in implementing each step and testing it throughout

before adding a new layer. Our approach is to always write one of the parts as a function, test it comparing the results with analytical results or whatever the answer is reasonable. An example of the former is our implementation of gensys, in which we can compare the impulse response functions (IRFs) given by the algorithm with the analytical IRFs; an example of the latter is computing the likelihood over a grid: we would expect that there is a local maximum around the true value of the parameters, fixing all other parameters in their true values.

2 The Model

Galí (2009) gives a standard three equations New Keynesian DSGE model:

$$\pi_t = \beta E_t(\pi_{t+1}) + \kappa \tilde{y}_t \tag{1}$$

$$\tilde{y}_t = -\frac{1}{\sigma}(i_t - E_t(\pi_{t+1})) + E_t(\tilde{y}_{t+1}) \tag{2}$$

$$i_t = \phi_\pi \pi_t + \phi_{\tilde{y}} \tilde{y}_t + v_t \tag{3}$$

$$v_t = \rho_v v_{t-1} + \varepsilon_t^v \tag{4}$$

We refer the reader to the book for understanding how to go from the optimization problem to this log linear equations and what each parameter means. The calibrated parameters by Galí are the following:

```
bet = 0.99
sig = 1
phi = 1
alfa = 1/3
epsilon = 6
theta = 2/3
phi_pi = 1.5
phi_y = 0.5/4
rho_v = 0.5

true_pars = (alfa = alfa, bet = bet, epsilon = epsilon, theta = theta, sig = sig, s2 =
1, phi = phi, phi_pi = phi_pi, phi_y = phi_y, rho_v = rho_v)

THETA = (1-alfa)/(1-alfa+alfa*epsilon)
lamb = (1-theta)*(1-bet*theta)/theta*THETA
kappa = lamb*(sig+(phi+alfa)/(1-alfa))
```

Galí (2009) gives an analytical solution to the model above. We implement this and it will be useful for checking whatever we implemented Gensys right, and it gives us some idea of the size of the numerical errors in the algorithm, if any.

3 The software

We implement all of this in Julia. Julia balances speed with a relatively easy to understand language: it might be slower than Fortran or C++, however it is much easier to learn. Julia is shipped with many facilities for linear algebra that will be useful. Last, but not least, there is a package system for Julia just like R and a rich ecosystem. We will use many of the

packages available to skip some boring but important steps, and just focus on the algorithms that are central to solving and estimating a DSGE. One exception is the Kalman Filter, which we use the implementation by Thomas Sargent.

3.1 Gensys

Gensys is a solver of linear rational expectation systems. It requires that we write the model using expectational errors, i.e. $\eta_y = E_t(y_{t+1}) - y_{t+1}$. The cannonical form of a DSGE model for Gensys is:

$$\Gamma_0 y_t = \Gamma_1 y_{t-1} + \Psi \varepsilon_t + \Pi \eta_t$$

In which ε_t are fundamental shocks and η_t are expectational shocks. Gensys takes a model in this form and finds matrices Θ_1 and Θ_2 such that the data follows a VAR(1):

$$y_t = \Theta_1 y_{t-1} + \Theta_2 \varepsilon_t$$

We can rewrite the model from Galí's book using $\eta_t^{\pi} = \pi_t - E_{t-1}\pi_t$ and $\eta_t^{\tilde{y}} = \tilde{y}_t - E_{t-1}\tilde{y}_t$, and we get:

$$\pi_{t-1} - \kappa \tilde{y}_{t-1} + \beta \eta_t^{\pi} = \beta \pi_t \tag{5}$$

$$\sigma \tilde{y}_{t-1} + i_{t-1} + \eta_t^{\pi} + \sigma \eta_t^{\tilde{y}} = \pi_t + \sigma \tilde{y}_t \tag{6}$$

$$i_{t-1} - \phi_{\pi} \pi_{t-1} - \phi_{\tilde{y}} \tilde{y}_{t-1} - v_{t-1} = 0 \tag{7}$$

$$v_t = \rho v_{t-1} + \varepsilon_t^{v} \tag{8}$$

And it is straight foward to write the matrices Γ_0, Γ_1, Ψ, Π, as follows:

```
GAMMA_0 = [bet    0     0  0;
           1      sig   0  0;
           0      0     0  0;
           0      0     0  1]

GAMMA_1 = [1      -kappa  0  0;
           0       sig    1  0;
           -phi_pi  -phi_y  1 -1;
           0       0      0  rho_v]

PSI = [0; 0; 0; 1]

PI = [bet  0;
      1    sig;
      0    0;
      0    0]
```

Here are LaTeX versions of the matrices generated automatically with the package *Latexify*, from the inputed matrices for Julia:

```
using Latexify, Random

print(string("\$\$\\Gamma_0 = \$\$",latexify(GAMMA_0)))
```

$$\Gamma_0 = \begin{matrix} 0.99 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{matrix} \tag{9}$$

```
print(string("\$\$\\Gamma_1 = \$\$",latexify(GAMMA_1)))
```

$$\Gamma_1 = \begin{matrix} 1.0 & -0.12750000000000006 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 0.0 \\ -1.5 & -0.125 & 1.0 & -1.0 \\ 0.0 & 0.0 & 0.0 & 0.5 \end{matrix} \tag{10}$$

```
print(string("\$\$\\Psi = \$\$",latexify(PSI)))
```

$$\Psi = \begin{matrix} 0 \\ 0 \\ 0 \\ 1 \end{matrix} \tag{11}$$

```
print(string("\$\$\\Pi = \$\$",latexify(PI)))
```

$$\Pi = \begin{matrix} 0.99 & 0.0 \\ 1.0 & 1.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{matrix} \tag{12}$$

Gensys allows Γ_0 to be singular. To obtain a system without Γ_0, it relies on the generalized Schur decomposition for both Γ_0 and Γ_1, simultaneously. The **LinearAlgebra** package that comes with Julia has a function called `schur` that computes the (egneralized) Schur decomposition. To test whatever the system has a solution and if it is unique, Gensys *does not* relies on the usual Blanchard Khan condition i.e. whatever the number of explosive eigenvalues is equal to the number of jumping variables - in fact, Gensys does not require that one defines which variables are jumping and which are pre determined. The existence and unity of the solution is determined by the number of singular values that are different from zero. This comes from the Singular Value Decomposition (SVD), another decomposition from numerical linear algebra. Given the role these decompositions will play, it is necessary to give some explanation about them.

3.1.1 Numerical Linear Algebra

We will use two decompositions of matrices: the Schur Decomposition and the Singular Value Decomposition (SVD). Both are closely related to the more well known eigenvalue decomposition:

$$A = P^{-1}\Lambda P$$

In which Λ is a diagonal matrix. In general, P is not orthonormal. The eigenvectors matrix, P, is orthonormal, i.e. $P^{-1} = P'$ in some cases, for example, if A is normal (Symmetric matrices are normal).

The Schur Decomposition tries to find a decomposition of the same form of the eigenvalue decomposition, but instead of imposing that Λ is diagonal, we now want that P be orthonormal. So the Schur Decomposition then finds:

$$A = Q'SQ$$

Such that Q is orthonormal. Remarkably S is always an upper triangular matrix.

We will be actually interested in the Generalized Schur Decomposition. To understand this one, lets first understand what is the generalized eigenvalue. In the usual eigenvalue problem, we want to find a vector $\mathbf{x}$ and a value λ such that:

$$Ax = \lambda x$$

Rewritting this in matricial terms, we want:

$$(A - \lambda I)x = 0$$

In which I stands for the identity matrix. Now, what if we want to solve the following problem:

$$Ax = \lambda Bx$$

We can find matrices such that $B = P^{-1}P$ and $A = P^{-1}\Lambda P$, in which Λ is a diagonal matrix.

The idea for the Generalized Schur Decomposition will be the same, but now we want to find matrices Q, Z, S, T such that:

1. Q and Z are orthonormal
2. S and T are upper triangular
3. $A = QSZ$
4. $B = QTZ$
5. $\forall i S_{ii}, T_{ii}$ are not zero
6. The pairs (S_{ii}, T_{ii}) can be arranged in any order

Item 6 will be particularly useful in what follows

The SVD also follows the idea from the eigenvalue problem. Now the idea is to decompose A as USV', in which S is a diagonal matrix and both U and V are orthonormal. The diagonal values in S are known as singular values.

3.1.2 The function

We present the function as is. Our implementation follows closely Miao (2014):

```
function gensys(G0,G1,Psi,Pi;verbose = true, tol = 1e-12)
    n = size(G0,1)
    decomp_1 =
    try
        schur(G0,G1)
    catch
        eu = (-1,-1)
        Theta1 = zeros(n,n)
        Theta2 = zeros(n,n)
        Theta3 = zeros(n,n)
        ans = Sims(Theta1,Theta2,Theta3,eu)
        return ans
        if verbose
            @warn "Unknown error. Probably LAPACK Exception 2. Skipping."
        end
    end
    gen_eigen = abs.(decomp_1.beta ./ decomp_1.alpha)
    ordschur!(decomp_1, gen_eigen .< 1)

    ns = findfirst(sort(gen_eigen) .> 1) #finding the number of stable roots: find
first unstable root
    if isnothing(ns)
        ns=1
    else
        ns = ns -1
    end
    nu = n - ns
    S11 = decomp_1.S[1:ns,1:ns]
    S12 = decomp_1.S[1:ns,(ns+1):n]
    S22 = decomp_1.S[(ns+1):n,(ns+1):n]

    T11 = decomp_1.T[1:ns,1:ns]
    T12 = decomp_1.T[1:ns,(ns+1):n]
    T22 = decomp_1.T[(ns+1):n,(ns+1):n]

    Qt = decomp_1.Q'

    Q1 = Qt[1:ns,:]
    Q2 = Qt[(ns+1):n,:]

    Q2Pi = Q2*Pi #This is equation 2.25 in p. 46 Miao (2014)

    m = size(Q2Pi,2)

    svd_Q2Pi = svd(Q2Pi)
    svd_aux = svd_Q2Pi.S #S is a vector
    r = sum(abs.(svd_aux) .> tol)

    #Checking existence and uniqueness see p. 46-47, Miao (2014)
    if m > r
        eu = [1;0]
        if verbose == true
            @warn "No Unique Solution"
        end
        Theta1 = zeros(n,n)
        Theta2 = zeros(n,n)
```

```
        Theta3 = zeros(n,n)
        ans = Sims(Theta1,Theta2,Theta3,eu)
        return ans
    elseif m < r
        eu = [0;0]
        Theta1 = zeros(n,n)
        Theta2 = zeros(n,n)
        Theta3 = zeros(n,n)
        ans = Sims(Theta1,Theta2,Theta3,eu)
        return ans
        if verbose == true
            @warn "No solution"
        end
    else
        eu = [1;1]
        if verbose == true
            @info "Unique and Stable Solution"
        end
        U1 = svd_Q2Pi.U[:,1:r]
        Xi = Q1*Pi*pinv(Q2Pi) #bottom of p 46 #change in 11 dec 2019: pinv instead
of manually multiplying the elements

        Aux1 = S12-Xi*S22

        larg1 = size(Aux1,2)
        Aux2 = [S11 Aux1;zeros(larg1,size(S11,2)) Matrix(I,larg1,larg1)]
        larg2 = size(Aux2,2)
        larg2 = larg2 - size(T11,1)

        ##Matrices on top of p. 46

        Theta1 = [T11 T12-Xi*T22;zeros(larg2,n)]
        Theta1 = decomp_1.Z*inv(Aux2)*Theta1*decomp_1.Z'
        Theta2 = [Q1 - Xi*Q2;zeros(larg2,n)]
        Theta2 = decomp_1.Z*inv(Aux2)*Theta2*Psi
        Theta3 = zeros(n,n)
        ans = Sims(Theta1,Theta2,Theta3,eu)
        return ans
    end
end
```

There are a number of pieces of code that their only function is to catch errors and avoid the MCMC crashing at a given iteration (namely the `catch` and `try`). We also have a check for if a given singular value is bellow a threshold. This is again another attempt to starve floating point arithmetic problems. For example, in the code above, if we change ϕ_π from 1.5 to 0.5 - which should throw a not unique error - the SVD calculates one of the roots equal to a small number that it is not zero. However, due to analytical results, we know that this must be a numerical accuracy issue.

3.1.3 Our example

Galí (2009) gives the analytical expressions for the impulse response functions of this model. We implement those in Julia:

```
LAMBDA_v = ((1-bet*rho_v)*(sig*(1-rho_v)+phi_y)+kappa*(phi_pi-rho_v))^(-1)

y_irf(v) = -(1-bet*rho_v)*LAMBDA_v*v
```

```
pi_irf(v) = -kappa*LAMBDA_v*v
r_irf(v) = sig*(1-rho_v)*(1-bet*rho_v)*LAMBDA_v*v
i_irf(v) = (sig*(1-rho_v)*(1-bet*rho_v) - rho_v*kappa)*LAMBDA_v*v
v_irf(v,uu) = rho_v*v + uu
```

And we compute a sample IRF for a positive 25 basis shock in the interest rate:

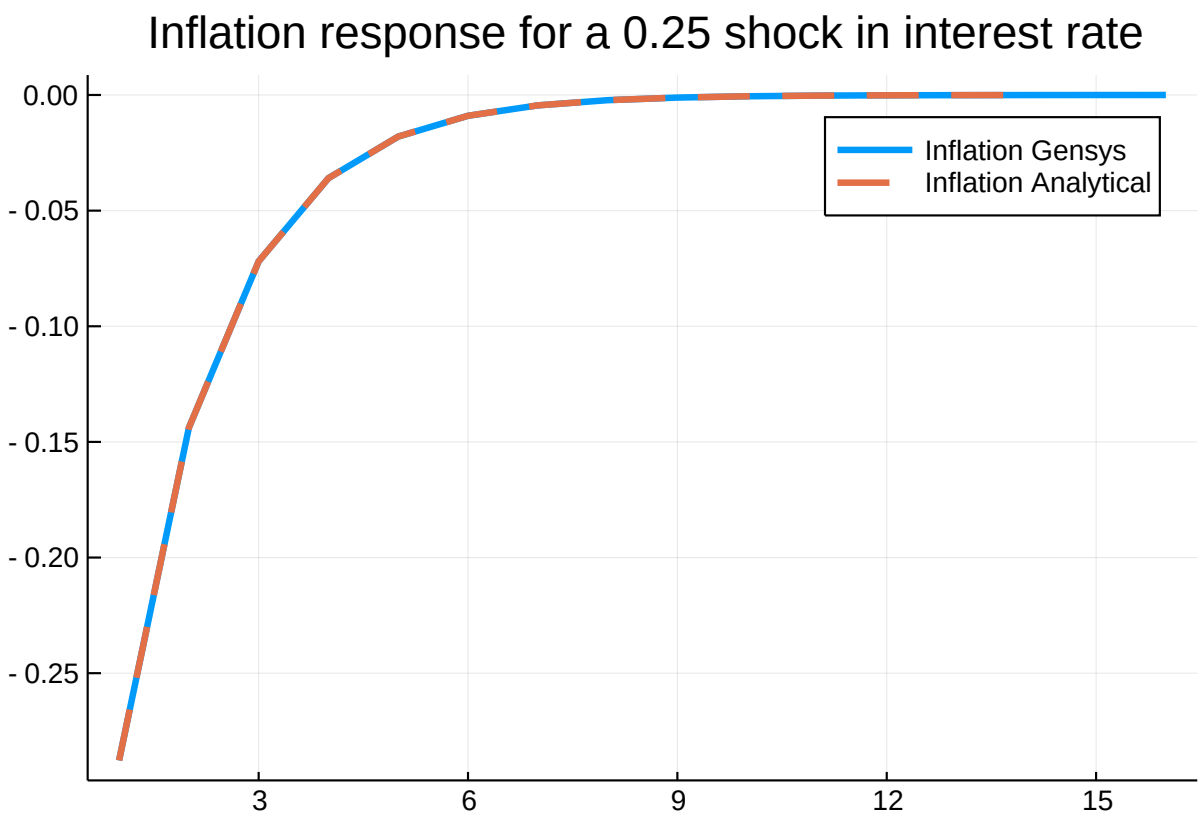
```
irfs_true = zeros(15,5)

irfs_true[1,1] = 0

uu = 0.25

for t in 2:15
    irfs_true[t,4] = v_irf(irfs_true[t-1,4],uu)
    irfs_true[t,1] = pi_irf(irfs_true[t,4])
    irfs_true[t,2] = y_irf(irfs_true[t,4])
    irfs_true[t,3] = i_irf(irfs_true[t,4])
    irfs_true[t,5] = r_irf(irfs_true[t,4])
    global uu = 0
end
```

And we just call the function with this matrices:

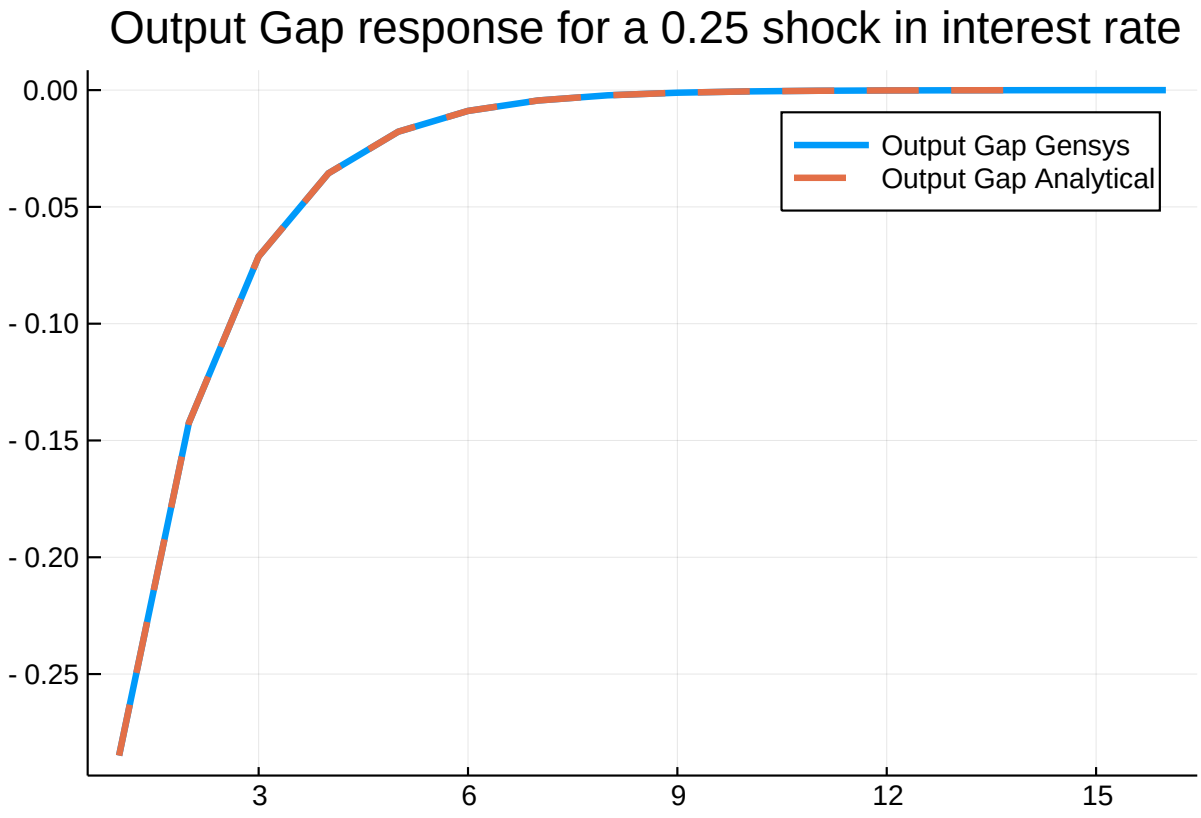
```
include(string(pwd(), "/src/gensys.jl")) #the original codefile, includes the irf
function that generates IRFs from the matrices Theta_1 and Theta_2

sol = gensys(GAMMA_0,GAMMA_1,PSI,PI)

irfs = irf(sol,15,0.25)
```

The sol object is a structure of type ”Sims” (that we created) and we write the `IRF` function such that it knows which fields to query from this object to compute the impulse response. Lets compare the IRFs generated analytically and by our implementation of Gensys:

```
using Plots

plot(4*irfs[:,1], label = "Inflation Gensys", w =3)
plot!(4*irfs_true[2:15,1], label = "Inflation Analytical", line = :dash, w =3)
title!("Inflation response for a 0.25 shock in interest rate")
```

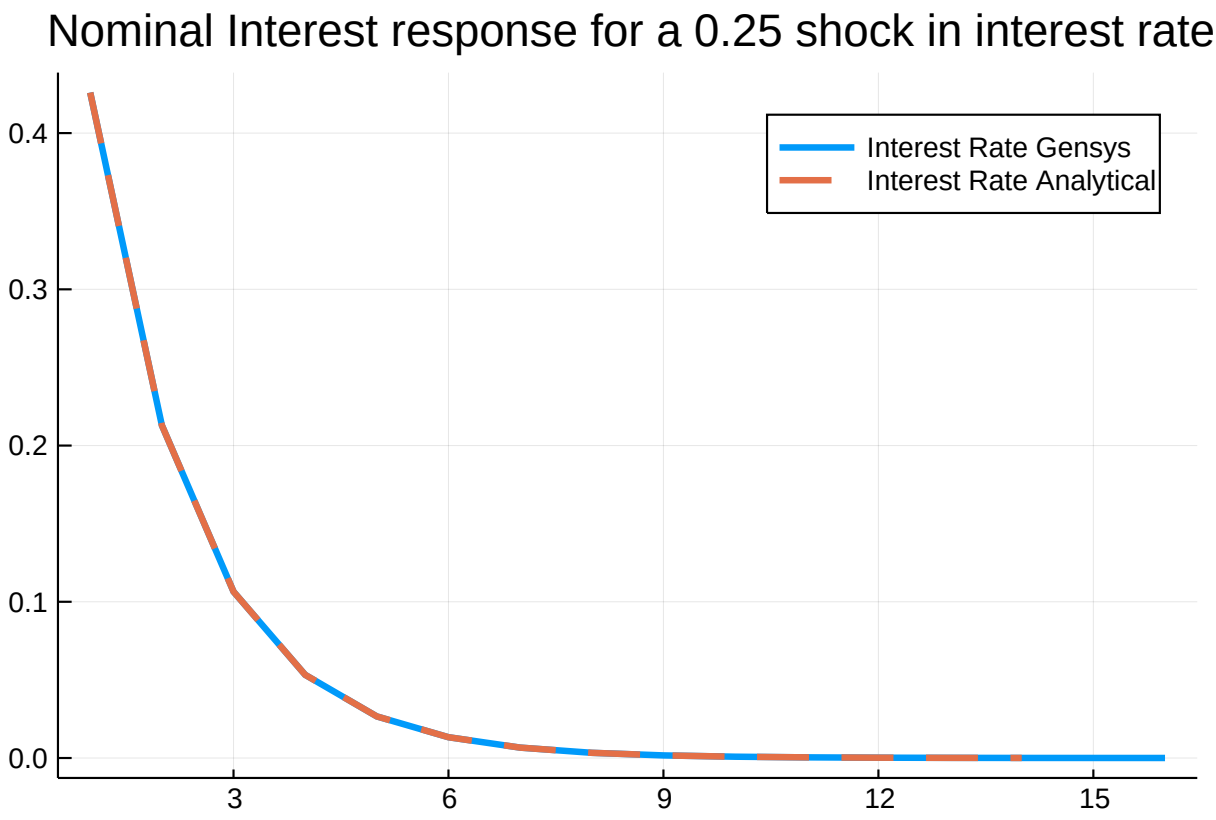



```
plot(irfs[:,2], label = "Output Gap Gensys", w = 3)
plot!(irfs_true[2:15,2], label = "Output Gap Analytical", line = :dash, w = 3)
title!("Output Gap response for a 0.25 shock in interest rate")
```

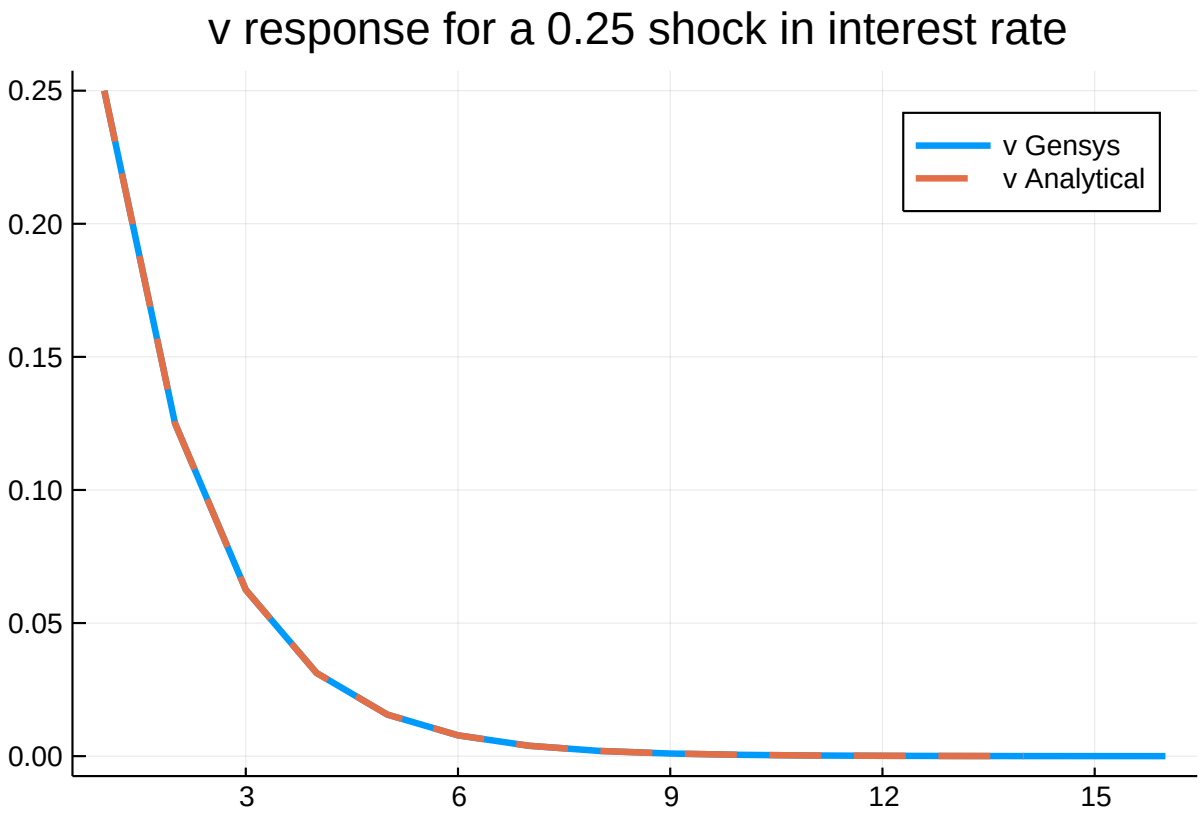



```
plot(4*irfs[:,3], label = "Interest Rate Gensys", w = 3)
plot!(4*irfs_true[2:15,3], label = "Interest Rate Analytical", line = :dash, w = 3)
```

```
title!("Nominal Interest response for a 0.25 shock in interest rate")
```




```
plot(irfs[:,4], label = "v Gensys", w = 3)
plot!(irfs_true[2:15,4], label = "v Analytical", line = :dash, w = 3)
title!("v response for a 0.25 shock in interest rate")
```



The impulse response are identical between the two methods, which is a strong evidence of the reliability and precision of our implementation of Gensys, Gensys itself and Julia capabilities to handle linear algebra routines.

3.2 Likelihood

The likelihood function is the next step of our code. The inputs are a set of parameter values and data regarding the economy. For a set of parameters, we use gensys to find the VAR representation of the model. We could estimate it as an usual VAR, however:

1. In general, the model assumes that the shock is an autocorrelated process. This means that the true process is a VARMA model and the coefficient of the autocorrelation and the fundamental shocks must be estimated as well.

2. For estimation, one can only include as many data series as there are shocks in the model. Otherwise, we incur in a stochastic singularity, i.e. the whole model is determined for any set of parameters. In this extremely simple example, this forces us to work with a single series. Since the dynamic of the model depends on the evolution of all the series, we have to estimate the remaining series from the data.

The fact that the model is linear with Gaussian shocks allows us to work with the Kalman Filter, that addresses both points above. The Kalman Filter works from the state space representation of the model:

$$y_t = Gx_t + v_t \tag{13}$$
$$x_t = Ax_{t-1} + w_t \tag{14}$$

In which v_t $N(0, R)$ and w_t $N(0, Q)$. The first equation is the *observation equation* and the second one is the *state equation*. The idea is that we observe y_t and that the behaviour of the system is governed by x_t - an idea closely related to the factor model in economics. This assumes asd well that we know G,A,R and Q. This allows us to recover x_t in a recursive manner.

What we do is to set A equals to Θ_1 and Q equals to $\sigma^2\Theta_2\Theta_2'$, in which Θ_1 and Θ_2 are the matrices from gensys and σ^2 is the variance from the monetary policy shock. So our state space is the whole system. G is just a selection matrix, i.e. which of the series from the state space we observe - this is filled with a value of 1 and everything else is just 0.

The value of R is *not* set to zero. The usual interpretation of the shock v_t is that it is a measurement error, i.e. we do not observe the exactly true GDP that our model prescribes because it has measurement errors. However, in our setup there are no such things: we simulate the values from the system and observe them exactly. So, we should set it to zero.

However, it is well known (see, for example, Keesman (2011)) that if there are no measurement errors the Kalman filter suffers from serious numerical instability problems. One that we found is that it often happens that the forecast variance matrix will have a determinant close to zero or even negative if there are no measurement errors. To avoid this, we add a measurement error with variance 10^{-8}. This creates a really small error and avoids the numerical instability problems completely. We could push it even closer to zero, however

due to the finite precision of floating point arithmetic we do not do it. There are ways of avoiding this - Keesman (2011) suggests a root Kalman filter that is more stable from the numerical point of view.

Although this trick helps preventing errors, it does not solve all the problems. Another problem - that is related to taking the inverse of the forecast error variance - is that the matrix might be *ill-condition*. This is related to the condition number of a matrix, $||A||||A^{-1}||$. Large condition numbers means that the inverse is an ill posed problem, from the numerical point of view; condition numbers close to one means that the problem is well behaved. We follow Dynare's code in skipping the cases in which the forecast variance matrix is ill conditioned (they use the inverse of the condition number, so numbers closer to zero means that the problem is ill posed. We use the same threshold they use).

Each step of the Kalman filter uses the current distribution about the state to forecast the state in the future. The error from the difference of the forecasted state to the real value of the state gives the mean of the normal. The forecast error variance is $P = G\Sigma_s G' + R$, in which Σ_s is the variance matrix of the states. This gives us all we need to evaluate the likelihood at each observation. With the likelihood at each point, one can then *multiply* all of the values.

This is problematic because, in many cases, the value of the density at the point is smaller than 1. Since the computer has a finite decimal precision (known as the *eps*), multiplying several numbers bellow one can quickly become messy. Just as an example, taking 0.0001^{100} (this would be just a hundred observations with density 0.0001) is bellow the eps of most CPUs. To avoid this, is better to work with the log of the likelihood and sum them over. That is the procedure we follow.

We also throw away the first nine likelihoods from the state. One needs to initialize the state distribution at some value - since we are working with the Normal distribution, this means the initialize the mean and the variance of the distribution. It takes a while to make this values converge, and in some cases this throws non sense values that destroy the loglikelihood (like 10^{18}). For initialization, we use the mean and the variance of the observed equation.

Here is the function for the (log)likelihood:

```
using QuantEcon
using Statistics
using LinearAlgebra

include(string(pwd(),"/src/gensys.jl"))

function log_like_dsge(par,data;kalman_tol = 1e-10)
    #order to par
    #alfa
    #beta
    #epsilon
    #theta
    #sig
    #sigma: this is the std dev of the innovation
    #phi
    #phi_pi
    #phi_y
    #rho_v

    #data will have t x p dimension: lines are periods p are variables

    alfa = par[1]
```

```
bet = par[2]
epsilon = par[3]
theta = par[4]
sig = par[5]
#par 6 is coded bellow see line 56
phi = par[7]
phi_pi = par[8]
phi_y = par[9]
rho_v = par[10]

THETA = (1-alfa)/(1-alfa+alfa*epsilon)
lamb = (1-theta)*(1-bet*theta)/theta*THETA
kappa = lamb*(sig+(phi+alfa)/(1-alfa))

nobs = size(data,1)

GAMMA_0 = [bet  0    0  0;
           1    sig  0  0;
           0    0    0  0;
           0    0    0  1]

GAMMA_1 = [ 1       -kappa  0   0;
            0        sig    1   0;
           -phi_pi  -phi_y  1  -1;
            0        0      0   rho_v]

PSI = [0; 0; 0; 1]

PI = [bet  0;
      1    sig;
      0    0;
      0    0]

p = size(GAMMA_1,1) #number of endogenous vars

sol = gensys(GAMMA_0,GAMMA_1,PSI,PI; verbose = false)
if sum(sol.eu) != 2
    return -NaN
end

#Sig = zeros(p,p)
#Sig[4,4] = par[6]

G = zeros(1,p)
G[1,2] = 1

A = sol.Theta1
R = [0] .+ 1e-8
Q = par[6]^2*sol.Theta2*sol.Theta2'

kalman_res = Kalman(A,G,Q,R) #create a Kalman filter instance

y_mean = mean(data,dims=1)
y_var = var(data,dims=1)
y_var = reshape(y_var,size(y_var,2))
#y_var = repeat([1],p)

x_hat = repeat(y_mean,p) #initial mean of the state
x_var = diagm(repeat(y_var,p))#variance initial of state
```

```
    #x_var = x_var*x_var'
    set_state!(kalman_res,x_hat,x_var)

    fit = zeros(nobs,4)

    llh = zeros(nobs)

    for j in 1:(nobs-1)
        med = kalman_res.cur_x_hat
        fit[j,:] = med
        varian = kalman_res.cur_sigma
        #println(det(varian))
        eta = data[j+1,:] - kalman_res.G*med #mean loglike
        P = kalman_res.G*varian*kalman_res.G' + kalman_res.R
        teste_cond = 1/cond(P)
        if teste_cond < kalman_tol
            llh[j] = -500
        #elseif det(P) <= 0
        #    llh[j] = -500
        else
            llh[j] = -(p*log(2*pi) + logdet(P) .+ eta'*inv(P)*eta)/2
            QuantEcon.update!(kalman_res,data[j+1,:]) #updating the kalman estimates
        end
        #println(kalman_res.cur_sigma)
        #println(det(varian))
        #println(P)
    end
    llh = llh[10:length(llh)]
    return sum(llh)
end
```

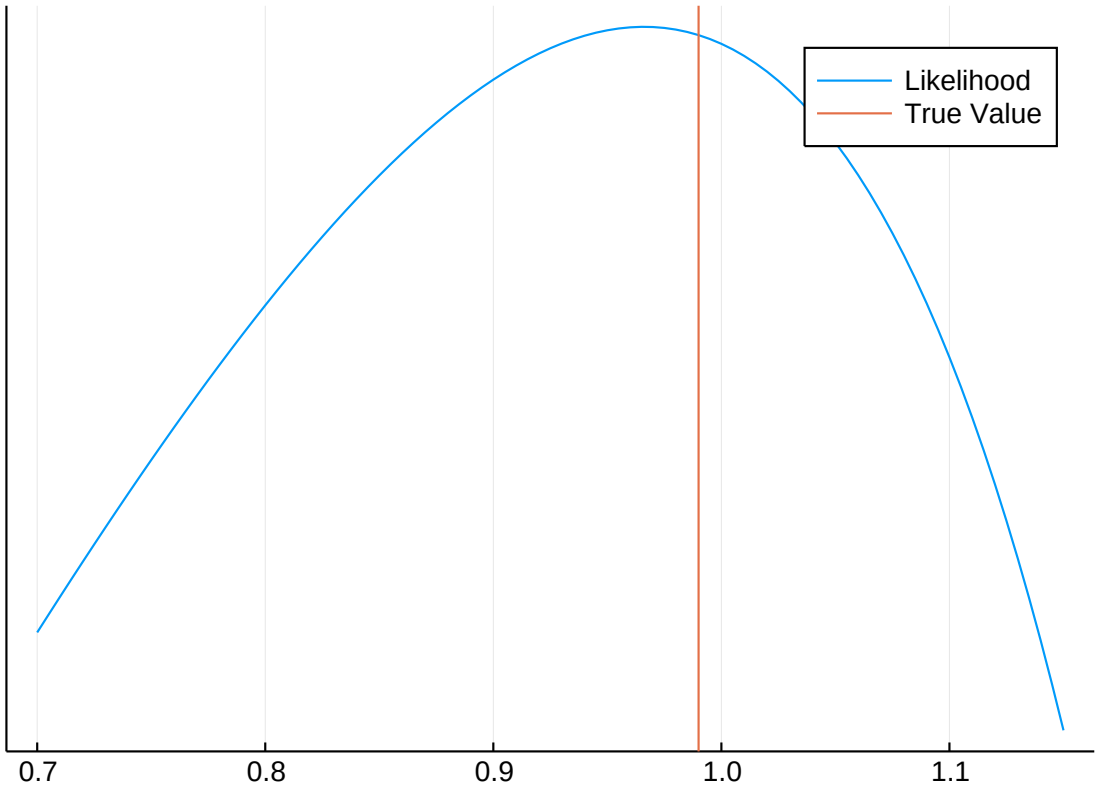
3.2.1 Example

Lets simulate an economy using the parameters from Galí's book:

```
Random.seed!(324128)

include(string(pwd(),"/src/simulation.jl"))

dados,choques = simulate_dsge(GAMMA_0,GAMMA_1,PSI,PI,5000)
```

Now compute the likelihood for beta

```
bett = 0.70:0.005:1.15

vals_bet = collect(bett)

for i in 1:length(bett)
    param1 = [alfa,bett[i],epsilon,theta,sig,1,phi,phi_pi,phi_y,rho_v]
    vals_bet[i] = log_like_dsge(param1,dados[:,2])
end

plot(bett,vals_bet, label = "Likelihood")
vline!([0.99], label = "True Value")
```



The shape is what we expected: increasing or decreasing the parameter from some point near the true value should decrease the likelihood. Notice that the maximum of this function is not exactly in the true value of the parameter - even though we have a sample of size 5000, many times the size of the usual macroeconomics database.

3.3 Markov Chain Monte Carlo

Bayesian estimation is based around Bayes's Theorem:

$$P(\Theta|y) = \frac{\ell(y|\Theta)p(\theta)}{P(y)}$$

In which Θ is the vector of parameters, y is the data, ℓ is the likelihood function, $p(\Theta)$ is the prior over the parameters and $P(y)$ is the marginal over the data. $P(y)$ actually depends of integrals over all the parameters, and so we suffer the curse of dimensionality. Markov Chain Monte Carlo (MCMC) solve this by avoiding estimating $P(y)$ and instead relying on the reason $\frac{\ell(y|\Theta)p(\theta)}{\ell(y|\Theta')p(\theta')}$ to build the posterior. The algorithm is rather simple. Besides the prior and the likelihood, we will also need a kernel distribution, that we will call $\mathcal{K}(\Theta|\Theta')$. Here is the algorithm for a special and rather useful algorithm for MCMC, the Metropolis-Hasting:

For j in 1:ndraws:

1. Draw a vector of parameters Θ^*, from the distribution $\mathcal{K}(\Theta|\Theta_j)$.
2. Compute $r = \min\left\{\frac{\ell(y|\Theta^*)p(\Theta^*)\mathcal{K}(\Theta_{j-1}|\Theta^*)}{\ell(y|\Theta_{j-1})p(\Theta_{j-1})\mathcal{K}(\Theta^*|\Theta_{j-1})}, 1\right\}$
3. Draw a random number from a $\alpha = U(0,1)$

4. If $r < \alpha$, set $\Theta_j = \Theta_{j-1}$; otherwise, set $\Theta_j = \Theta^*$

end

In the end we have $\{\Theta_j\}_{j=1}^{ndraws}$, a set of ndraws of the parameter vector.This algorithm maps the posterior, and is quite smart: every time you draw a parameter vector that makes the likelihood times the prior increase a lot compared to the point you are now, you accept. However, even when it does not increase the likelihood, this does not means that the vector will be excluded: it has some probability of being included. This allows the algorithm to map the distribution and to not get stuck in some points of high probability, like the mode.

The choice of the kernel is important. Usually, the kernel is a Normal distribution that has mean in the last parameter vector accepted (therefore the random walk part of the name that we used in the introduction). The choice of the variance is of central importance: the optimal choice for the variance of the kernel is $c\Sigma_\Theta$, in which Σ_Θ is the variance matrix for the vector Θ evaluated on the mode. c is a scale constant that plays a fundamental role to ensure that we will map the whole distribution. The idea is simple: if the variance is too low, you will visit only a small part of the distribution in a given number of iterations; set it too high and the algorithm will start to wander in regions far away from the mode, that contribute little to the overall density. Therefore setting c in the right way ensures that our iterations won't be wasted.

Notice that our previous discussion about multiplying small numbers also applies when taking the ratio of two (potentially) small numbers. Therefore, although we could get the likelihood from the log likelihood (just apply the exponential), we do not do it. Instead, step 3 becomes:

$$r = \min\{0, \log(\ell(y|\Theta^*)) + \log(p(\Theta^*)) + \log(\mathcal{K}(\Theta_{j-1}|\Theta^*)) - \log(\ell(y|\Theta_{j-1})) + \log(p(\Theta_{j-1})) + \log(\mathcal{K}(\Theta^*|\Theta_{j-1}))\}$$

And we use $\log(\alpha)$ instead of α in step 4. We tried working with the exponential of the loglikelihood. However, this was not entirely sucessful. On a side note, this trick is not mentioned in any of the books we checked, for example Canova (2011).

Here are the priors. We create the function `gamma_mv` that receives the mean and the standard deviation we want the prior to have, instead of the usual shape and scale parameters:

```
using Distributions

gamma_mv(mu,std) = Gamma(mu^2/std^2,std^2/mu)
gammainv_mv(mu,std) = Gamma(mu^2/std^2,(std^2/mu)^(-1))

prior_bet = Beta(100,1.01010101)
prior_epsilon = Normal(6,1.5)
prior_theta = Beta(3,1.5)
prior_sig = gamma_mv(1,0.5)
prior_phi = gamma_mv(1,0.5)
prior_phi_pi = gamma_mv(1.5,0.4)
prior_phi_y = gamma_mv(0.5/4,0.2)
prior_rho_v = Beta(0.88,0.88)
prior_s2 = InverseGamma(3.76,2.76)
```

Notice that we have a potential problem using a Normal kernel: some parameters are constrained in some interval of the reals. An example would be the β, the intertemporal discount parameter: this is constrained between 0 and 1. We can use some transformation of the parameters so they are paremetrized in the whole real line - e.g. take log of the

parameters that are positive. This requires an adjusment in the likelihood function by the determinant of the jacobian of this transformations. This is bothersome. Fortunately, Julia has a package for handling this transformations, called **TransformVariables**. All we have to do is tell it what interval of the real line the parameters are constrained. This is easier than writing the code for the jacobian, however it requires some steps:

```
using Parameters, TransformVariables

struct DSGE_Model{data <: AbstractArray, alfa <: Float64}
    "Data"
    data::data
    "Alpha"
    alfa::alfa
end

function (problem::DSGE_Model)(pars) #this is the numerator of the posterior
    @unpack alfa, data = problem
    @unpack bet, epsilon, theta, sig, s2, phi, phi_pi, phi_y, rho_v = pars
    llh = log_like_dsge([alfa,bet,epsilon,theta,sig,s2,phi,phi_pi,phi_y,rho_v],data)
        return
llh+logpdf(prior_bet,bet)+logpdf(prior_epsilon,epsilon)+logpdf(prior_theta,theta)+logpdf(prior_sig,sig)
end

function problem_transform(p::DSGE_Model) #this makes the changes in parameters
    as((bet = as_unit_interval, epsilon=as_positive_real,theta = as_unit_interval, sig =
as_positive_real, s2 = as_positive_real, phi = as_positive_real, phi_pi =
as_positive_real, phi_y = as_positive_real, rho_v = as_unit_interval))
end
```

The previous code requires a bit more of explanation. The first thing we do is define a *structure* - a relative of R's list - that receives the data series and the parameter α, that we are calibrating. We are defining this type of *structure* to be of type `DSGE_Model`. The second function takes an object of type `DSGE_Model` to then generate a function that is the numerator of the posterior, the (log)likelihood times (plus, since we are using log) the (log) of the prior of each parameter. This is a function that generates a function, which is a bit unusual. The last function is responsible to carrying out the transformation.

Lets start with the basic MCMC algorithm code, assuming that we know the scale parameter c:

```
coef_escala = 0.24

pars_aceitos[1,2:10] = rand(MvNormal(hes_inv),1)

j = 2
rejec = 0

while j <= num_iter
    kernel_velho = MvNormal(pars_aceitos[j-1,2:10],coef_escala*hes_inv)
    novo_par = rand(kernel_velho)
    #pars_aceitos[j,2:10] = novo_par
    kernel_novo = MvNormal(novo_par,coef_escala*hes_inv)

    teste = LogDensityProblems.logdensity(P,novo_par)
    if isnan(teste)
        #pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
        #global rejec += 1
        #global j += 1
        continue
```

```
    else
        num = teste + logpdf(kernel_novo,pars_aceitos[j-1,2:10])
        #println(string(teste," ",logpdf(kernel_novo, pars_aceitos[j-1,2:10])))
        dem = LogDensityProblems.logdensity(P,pars_aceitos[j-1,2:10]) +
logpdf(kernel_velho,novo_par)
        alpha = exp(min(0,num - dem))
        p = rand(unif)
        if alpha < p
            pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
            global rejec += 1
        else
            pars_aceitos[j,2:10] = novo_par
        end
        acc = 1 - rejec/j
        if j % 50 == 0
            println("Iteração ", j, " taxa de aceitação ", acc)
            sleep(0.4)
        end
        global j += 1
    end
end
```

We have a trick here as well: every time we draw a set of parameters that generates an equilibrium that is not unique or if there is no equilibrium, then we disregard the whole iteration and repeat it. It does not counts as a value of the parameter that was rejected. It simply never happened. This avoids getting stucked in a region and getting the same set of values of that region in the set of parameters, which could distort the distribution.

If we do not know the scale parameter - as it happens most of times - we need to tune the parameter. Gelman *et al.* (2014) suggests the following procedure: run the MCMC for a while, changing the scale coefficient and updating the covariance matrix to the empirical covariance matrix. Change the scale parameter to achieve around 23% of acceptance rate. When it is lower than this, reduce the scale parameters; if it is larger than this, increase. Then, after we settle for a scale coefficient and covariance matrix, we run the MCMC for real. From this last run we get the parameters that will be used to build the posterior distribution.

In pratice, we implement this in a three step process:

1. Start at some random point of the parameter space. For 5000 times, run the MCMC and update the covariance matrix at each multiple of 1000, using just the last 500 estimate of the parameter vector. This attempts to isolate possible ”transition dynamics” due to the change in the distribution when we changed the covariance matrix

2. Take the last parameter vector from the previous procedure. Starting from this point, run the MCMC for more 5000 times. This time we will choose the scale parameter following the procedure of Gelman *et al.* (2014). They do not give a rule by how much the scale coefficient should change and we settle with a reduction of 20% in the value and an increase of 10% - this is completely arbitrary.

3. Using the covariance matrix and the scale parameter, run the MCMC for n iterations to obtain the posterior distribution.

Step 2 requires a lot of care. First, we don’t want to contaminate the acceptance ratio by the acceptance ratio of the previous scale parameter. This requires that we reset the

acceptance ratio every time we change the scale parameter. Moreover, it takes a while before the acceptance ratio settles. This forces us to discard the first 900 iterations for a new scale parameter and just the final 300 to compute the acceptance ratio.

Hitting exactly 23% of the acceptance ratio is impossible. We settle for any acceptance ratio between 18% and 30%, based on the values suggested by Dynare's documentation.

After we have finally a scale value that generates an acceptance in this range, we compute further 500 steps in the MCMC procedure just to check that we have a acceptance ratio in the right range. This sometimes generates acceptance ratio well outside the range establish and we go back to the iterative procedure, reducing or increasing the scale parameter based on the acceptance ratio of this last 500 iterations.

Here is the code for the first step:

```
rejec = 0

k = 2

while k <= adapt_warm_up
    kernel_velho = MvNormal(warm_up_pars[k-1,2:10],coef_escala*hes_inv)
    novo_par = rand(kernel_velho)
            #pars_aceitos[j,2:10] = novo_par
    kernel_novo = MvNormal(novo_par,coef_escala*hes_inv)

    teste = LogDensityProblems.logdensity(P,novo_par)
    if isnan(teste)
            #pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
            #global rejec += 1
            #global j += 1
        continue
    else
        num = teste + logpdf(kernel_novo,warm_up_pars[k-1,2:10])
        #println(string(teste," ",logpdf(kernel_novo, pars_aceitos[j-1,2:10])))
        dem = LogDensityProblems.logdensity(P,warm_up_pars[k-1,2:10]) +
logpdf(kernel_velho,novo_par)
        alpha = exp(min(0,num - dem))
        p = rand(unif)
        if alpha < p
            warm_up_pars[k,2:10] = warm_up_pars[k-1,2:10]
            global rejec += 1
        else
            warm_up_pars[k,2:10] = novo_par
        end
    end

    if k % 500 == 0
        acc = 1 - rejec/k
        println("Warm up phase, iteration " ,k, " acceptance ", acc*100,"%")
    end

    if k % 1000 == 0
        pars_space = warm_up_pars[(k-499):k,2:10]
        global hes_inv = cov(pars_space)
        global hes_inv = (hes_inv + hes_inv' )/2
        global hes_inv = cholesky(Positive,hes_inv)
        global hes_inv = hes_inv.L*hes_inv.L'
        global rejec = 0
    end
```

```
    global k += 1
end
```

And for the second step, the algorithm has too many ifs and whiles:

```
hes_inv_backup = hes_inv

j = 2
rejec = 0
test = true

coef_escala = 2.4/sqrt(npar)

final_countdown = 0

pars_aceitos[1,2:10] = warm_up_pars[adapt_warm_up,2:10]

while j <= adapt && test
    kernel_velho = MvNormal(pars_aceitos[j-1,2:10],coef_escala*hes_inv)
    novo_par = rand(kernel_velho)
            #pars_aceitos[j,2:10] = novo_par
    kernel_novo = MvNormal(novo_par,coef_escala*hes_inv)

    teste = LogDensityProblems.logdensity(P,novo_par)
    if isnan(teste)
            #pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
            #global rejec += 1
            #global j += 1
        continue
    else
        num = teste + logpdf(kernel_novo,pars_aceitos[j-1,2:10])
        #println(string(teste," ",logpdf(kernel_novo, pars_aceitos[j-1,2:10])))
        dem = LogDensityProblems.logdensity(P,pars_aceitos[j-1,2:10]) +
logpdf(kernel_velho,novo_par)
        alpha = exp(min(0,num - dem))
        p = rand(unif)
        if alpha < p
            pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
            if j % 1200 > 900
                global rejec += 1
            end
        else
            pars_aceitos[j,2:10] = novo_par
        end
    end
    if j % 1200 == 0
        acc = 1 - rejec/300
        #pars_space = pars_aceitos[(j-299):j,2:10]
        # global hes_inv = cov(pars_space)
        # global hes_inv = (hes_inv + hes_inv' )/2
        # global hes_inv = cholesky(Positive,hes_inv)
        # global hes_inv = hes_inv.L*hes_inv.L'
        if round(acc;digits=4) > 0.3
            rescale = 1.1#acc/0.23
            global coef_escala = coef_escala*rescale
            println("Adaptação: iteração ", j, " taxa de aceitação ", acc*100, "% new c
= ", coef_escala)
            sleep(0.4)
            #pars_aceitos[j,2:10] = pars_aceitos[1,2:10]
            global j += 1
```

```
            global rejec = 0
        elseif round(acc;digits=4) < 0.18
            rescale = 0.8#acc/0.23
            global coef_escala = coef_escala*rescale
            println("Adaptação: iteração ", j, " taxa de aceitação ", acc*100, "% new c
= ", coef_escala)
            sleep(0.4)
            #pars_aceitos[j,2:10] = pars_aceitos[1,2:10]
            global j += 1
            global rejec = 0
        else
            acc = 1 - rejec/300
            println("Adaptação: iteração ", j, " taxa de aceitação ", acc*100, "% c = ",
coef_escala)
            global rejec = 0
            while final_countdown <= 500
                kernel_velho =
MvNormal(pars_aceitos[j+final_countdown-1,2:10],coef_escala*hes_inv)
                novo_par = rand(kernel_velho)
                        #pars_aceitos[j,2:10] = novo_par
                kernel_novo = MvNormal(novo_par,coef_escala*hes_inv)

                teste = LogDensityProblems.logdensity(P,novo_par)
                if isnan(teste)
                        #pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
                        #global rejec += 1
                        #global j += 1
                    continue
                else
                    num = teste +
logpdf(kernel_novo,pars_aceitos[j+final_countdown-1,2:10])
                    #println(string(teste," ",logpdf(kernel_novo,
pars_aceitos[j-1,2:10])))
                    dem = LogDensityProblems.logdensity(P,pars_aceitos[j-1,2:10]) +
logpdf(kernel_velho,novo_par)
                    alpha = exp(min(0,num - dem))
                    p = rand(unif)
                    if alpha < p
                        pars_aceitos[j+final_countdown,2:10] =
pars_aceitos[j+final_countdown-1,2:10]
                        global rejec += 1
                    else
                        pars_aceitos[j+final_countdown,2:10] = novo_par
                    end
                    acc = 1 - rejec/final_countdown
                end
                println("Adaptação: iteração ", final_countdown, " taxa de aceitação ",
acc*100, "% c = ", coef_escala)
                global final_countdown += 1
            end
            if 0.3 > round(acc;digits=4) > 0.18
                global test = false
            else
                global final_countdown = 0
                if round(acc;digits=4) > 0.3
                    rescale = 1.1#acc/0.23
                    global coef_escala = coef_escala*rescale
                    println("Adaptação: iteração ", j, " taxa de aceitação ", acc*100,
"% new c = ", coef_escala)
```

```
                    sleep(0.4)
                    #pars_aceitos[j,2:10] = pars_aceitos[1,2:10]
                    global j += 1
                    global rejec = 0
                elseif round(acc;digits=4) < 0.18
                    rescale = 0.8#acc/0.23
                    global coef_escala = coef_escala*rescale
                    println("Adaptação: iteração ", j, " taxa de aceitação ", acc*100,
"% new c = ", coef_escala)
                    sleep(0.4)
                    #pars_aceitos[j,2:10] = pars_aceitos[1,2:10]
                end

                global rejec = 0
                global j += 1
            end
        end
    else
        global j += 1
    end
end
```

Finally, the step that generates the distribution, the third one:

```
j = 2
rejec = 0

pars_aceitos[1,2:10] = rand(MvNormal(hes_inv),1)

#coef_escala = 0.5

while j <= num_iter
    kernel_velho = MvNormal(pars_aceitos[j-1,2:10],coef_escala*hes_inv)
    novo_par = rand(kernel_velho)
        #pars_aceitos[j,2:10] = novo_par
    kernel_novo = MvNormal(novo_par,coef_escala*hes_inv)

    teste = LogDensityProblems.logdensity(P,novo_par)
    if isnan(teste)
        #pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
        #global rejec += 1
        #global j += 1
        continue
    else
        num = teste + logpdf(kernel_novo,pars_aceitos[j-1,2:10])
        #println(string(teste," ",logpdf(kernel_novo, pars_aceitos[j-1,2:10])))
        dem = LogDensityProblems.logdensity(P,pars_aceitos[j-1,2:10]) +
logpdf(kernel_velho,novo_par)
        alpha = exp(min(0,num - dem))
        p = rand(unif)
        if alpha < p
            pars_aceitos[j,2:10] = pars_aceitos[j-1,2:10]
            global rejec += 1
        else
            pars_aceitos[j,2:10] = novo_par
        end
        acc = 1 - rejec/j
        if j % 50 == 0
            println("Iteração ", j, " taxa de aceitação ", acc)
            sleep(0.4)
```

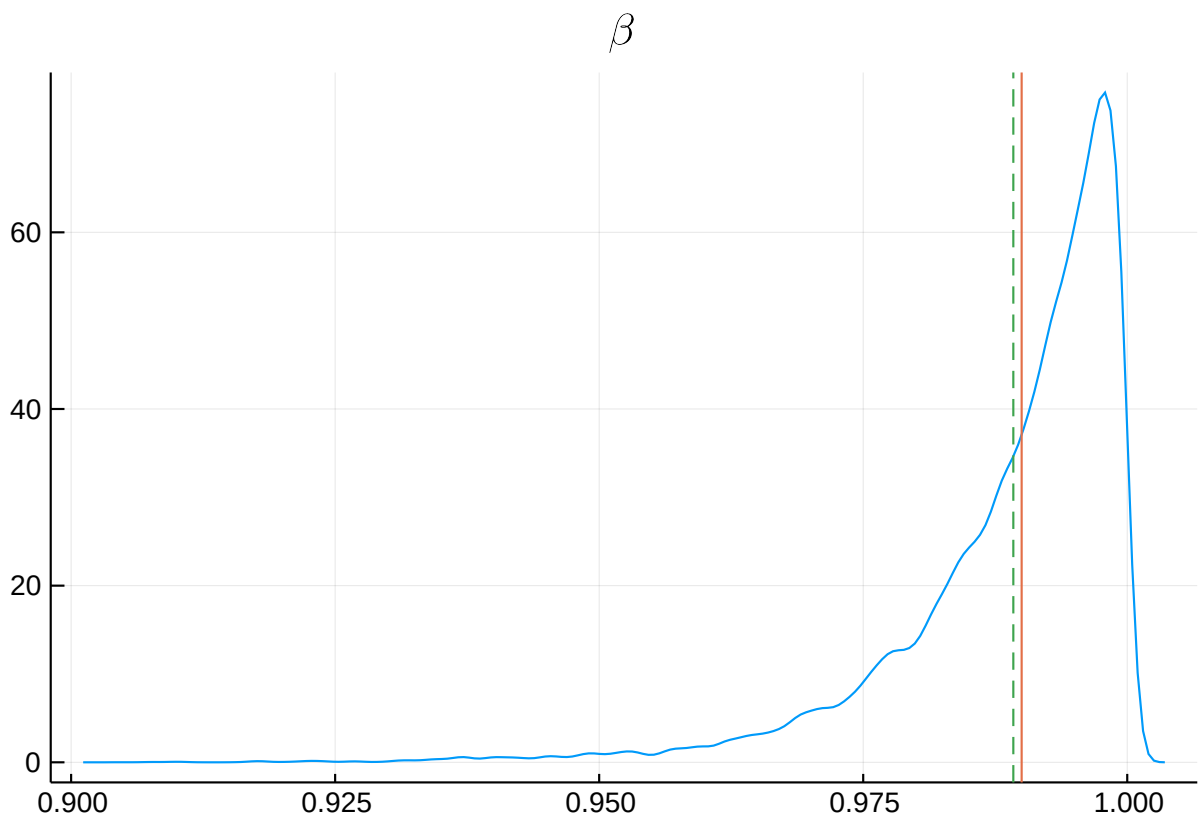
```
        end
        global j += 1
    end
end
```

We don't run this code when compiling this document because it takes a while to run it. Let's show the results of this estimation by plotting the densitites using the *StatsPlots* package. Notice that the parameters that we receive at the end of the MCMC are the transformed ones, not the parameters that interest us. We need to tranform them back to their domain:

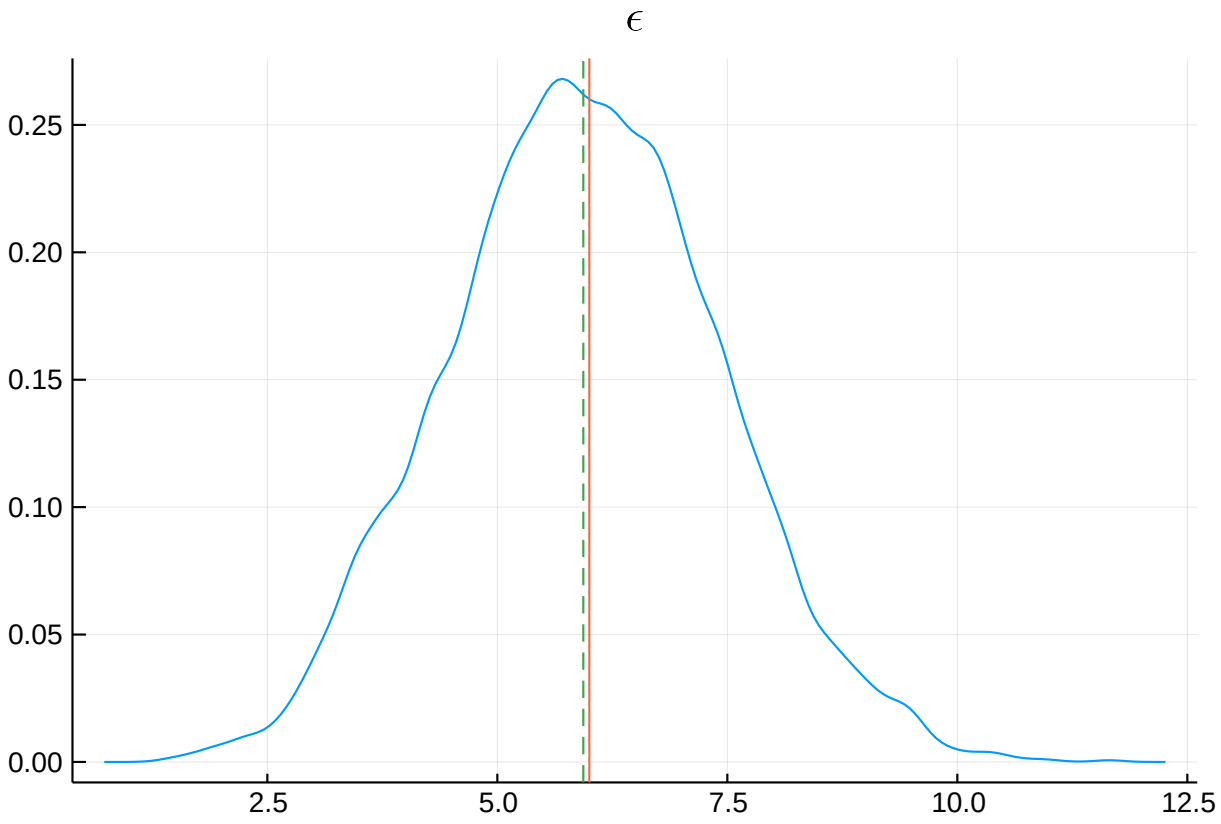
```
using JLD, StatsPlots, LaTeXStrings

tt = load("data/mcmc2.jld")

yy,shocks = simulate_dsge(GAMMA_0,GAMMA_1,PSI,PI,500)

pars_aceitos = tt["pars"]

prob = DSGE_Model(yy[:,2],1/3)
t = problem_transform(prob)

pars_convertidos = zeros(100000,9)

for i in 50001:100000
    @unpack bet,epsilon,theta,sig,s2,phi,phi_pi,phi_y,rho_v =
TransformVariables.transform(t,pars_aceitos[i,2:10])
    pars_convertidos[i,:] = [bet,epsilon,theta,sig,s2,phi,phi_pi,phi_y,rho_v]#(bet =
pars_aceitos[i,2], epsilon = pars_aceitos[i,3], theta = pars_aceitos[i,4],sig =
pars_aceitos[i,5],s2 = pars_aceitos[i,6],phi = pars_aceitos[i,7],phi_pi =
pars_aceitos[i,8],phi_y = pars_aceitos[i,9],rho_v = pars_aceitos[i,10]))
end
```

We will use the first half of the sample as warm up and the remaining will be used to build the distributions. We do not do any thinning - excluding some observations to generate a sample that is less autocorrelated, like excluding every even observation or just using 1 observation out of three (use, don't don't, use don't, don't...). The red vertical line represents the true line and the green dashed line the mean of the parameter in the distribution:

```
StatsPlots.density(pars_convertidos[50001:100000,1], legend =:none)
title!(L"\beta")
vline!([true_pars[:bet]])
vline!([mean(pars_convertidos[50001:100000,1])], line = :dash)
```

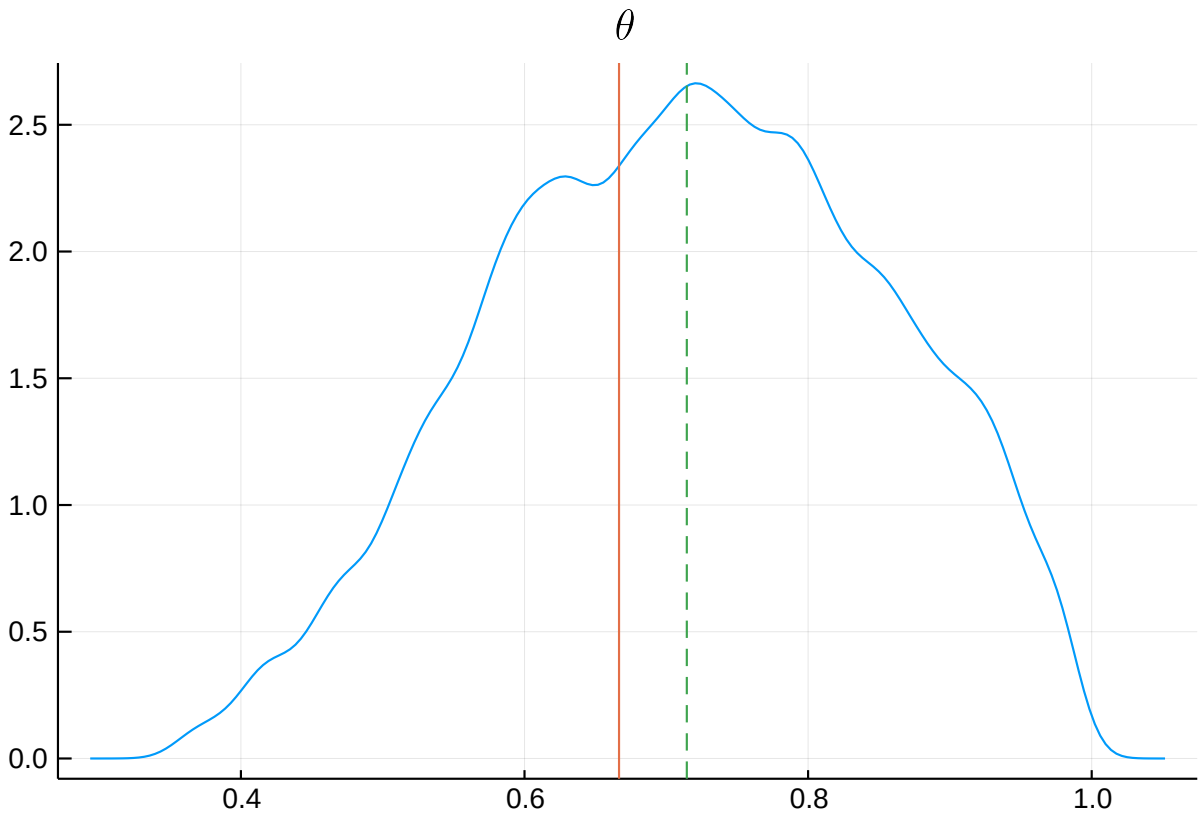



```
StatsPlots.density(pars_convertidos[50001:100000,2], legend =:none)
title!(L"\epsilon")
vline!([true_pars[:epsilon]])
vline!([mean(pars_convertidos[50001:100000,2])], line = :dash)
```

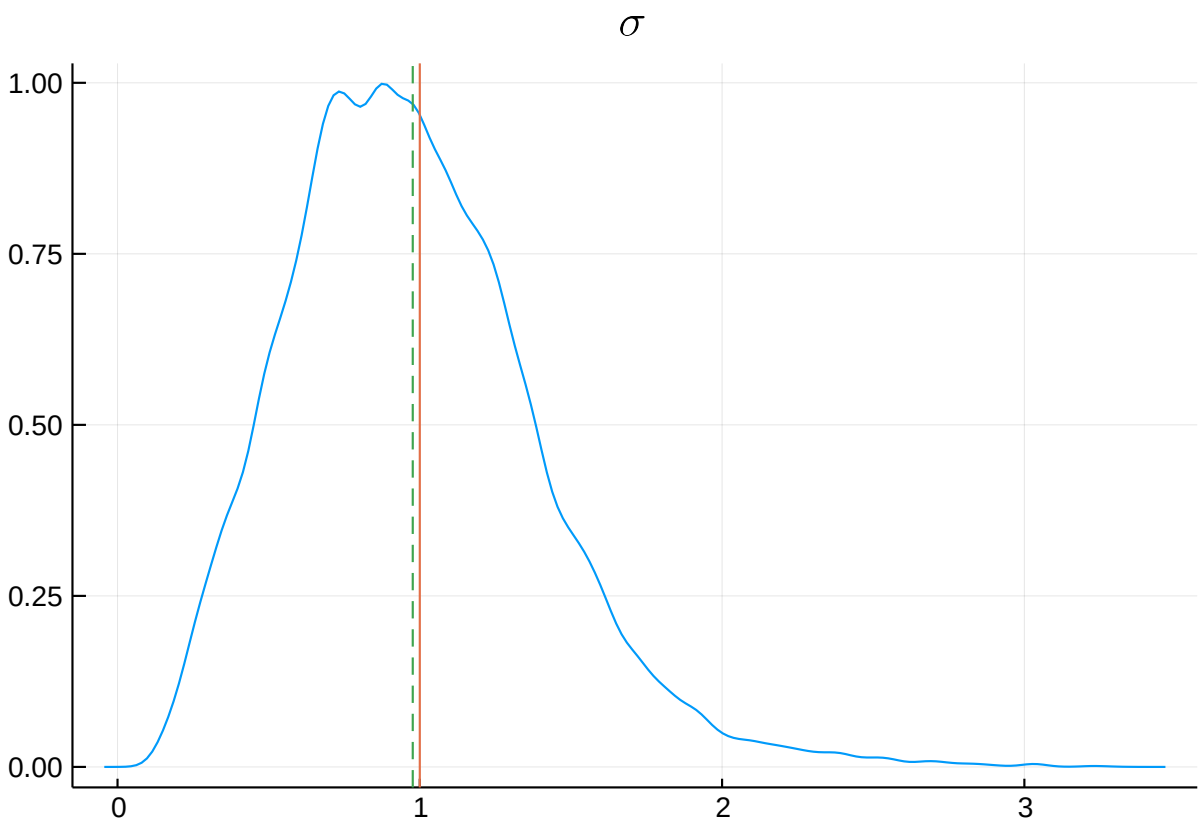



```
StatsPlots.density(pars_convertidos[50001:100000,3], legend = :none)
```

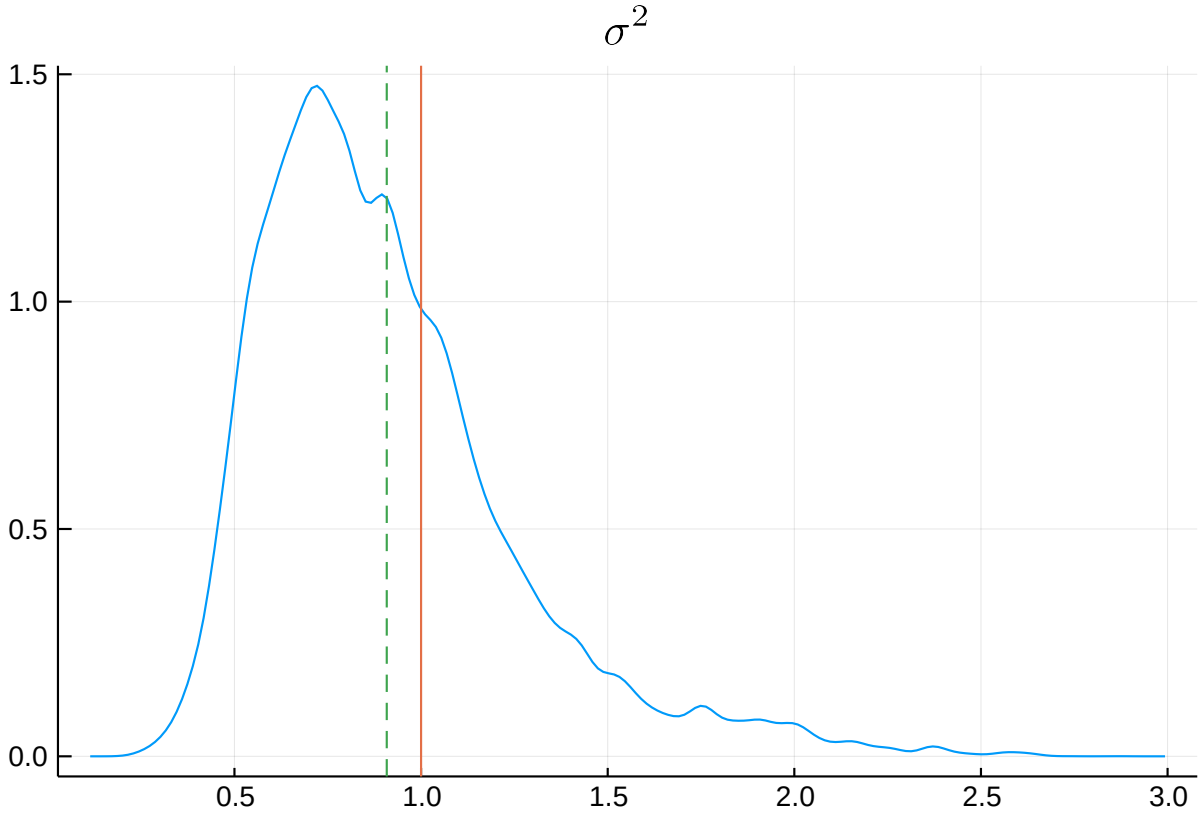
```
title!(L"\theta")
vline!([true_pars[:theta]])
vline!([mean(pars_convertidos[50001:100000,3])], line = :dash)
```




```
StatsPlots.density(pars_convertidos[50001:100000,4], legend = :none)
title!(L"\sigma")
vline!([true_pars[:sig]])
vline!([mean(pars_convertidos[50001:100000,4])], line = :dash)
```

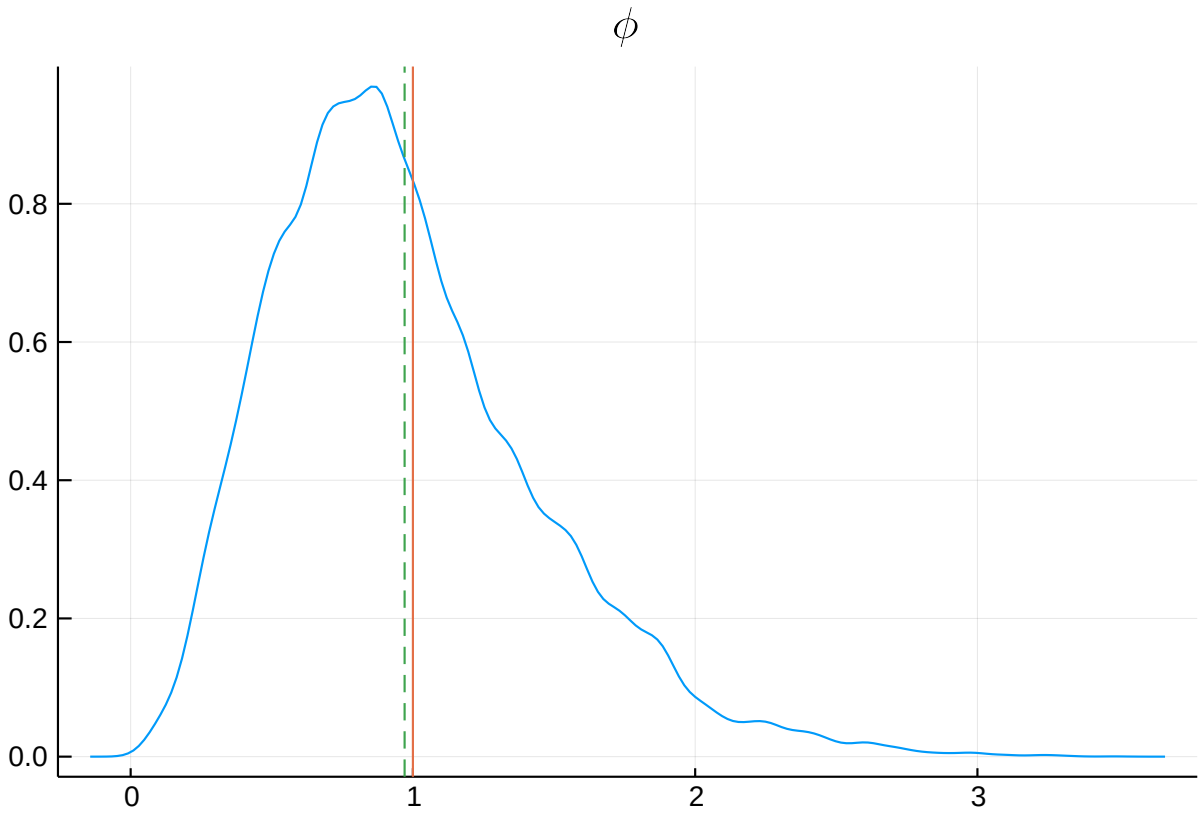



```
StatsPlots.density(pars_convertidos[50001:100000,5], legend = :none)
title!(L"\sigma^2")
vline!([true_pars[:s2]])
vline!([mean(pars_convertidos[50001:100000,5])], line = :dash)
```

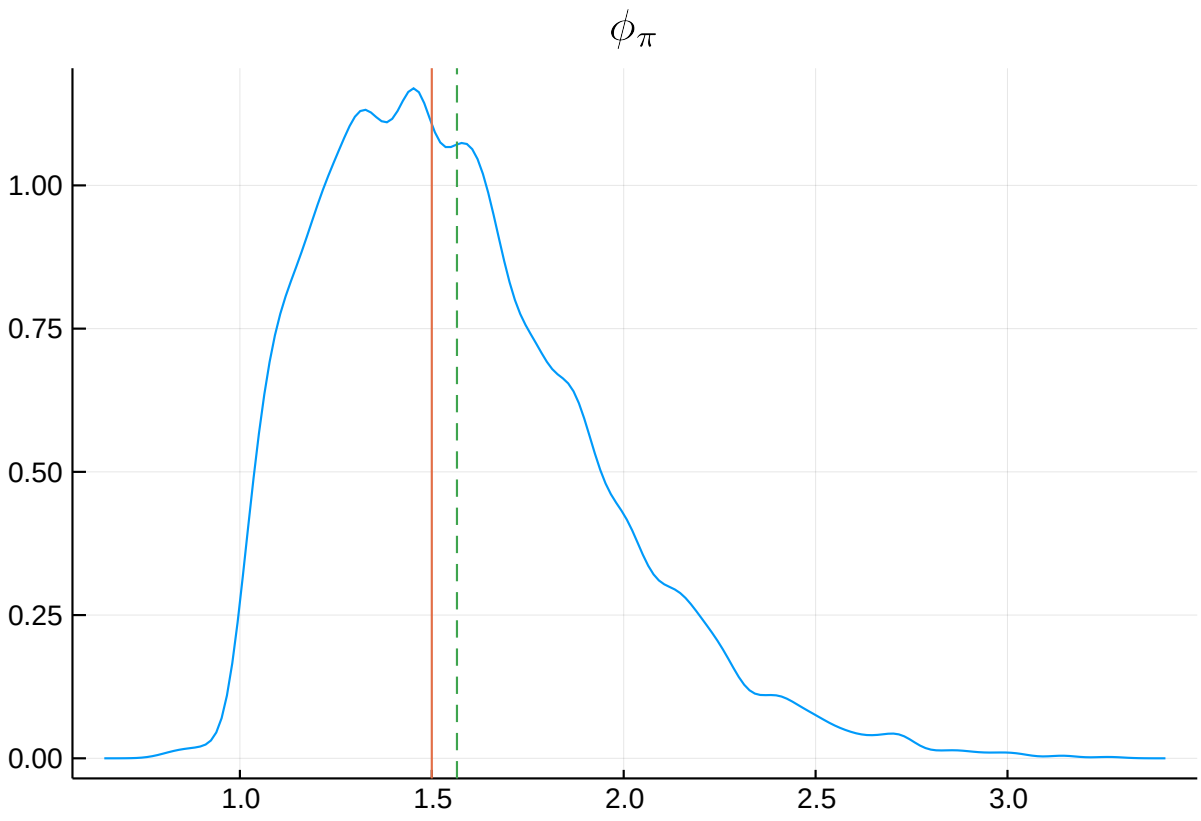



```
StatsPlots.density(pars_convertidos[50001:100000,6], legend = :none)
```

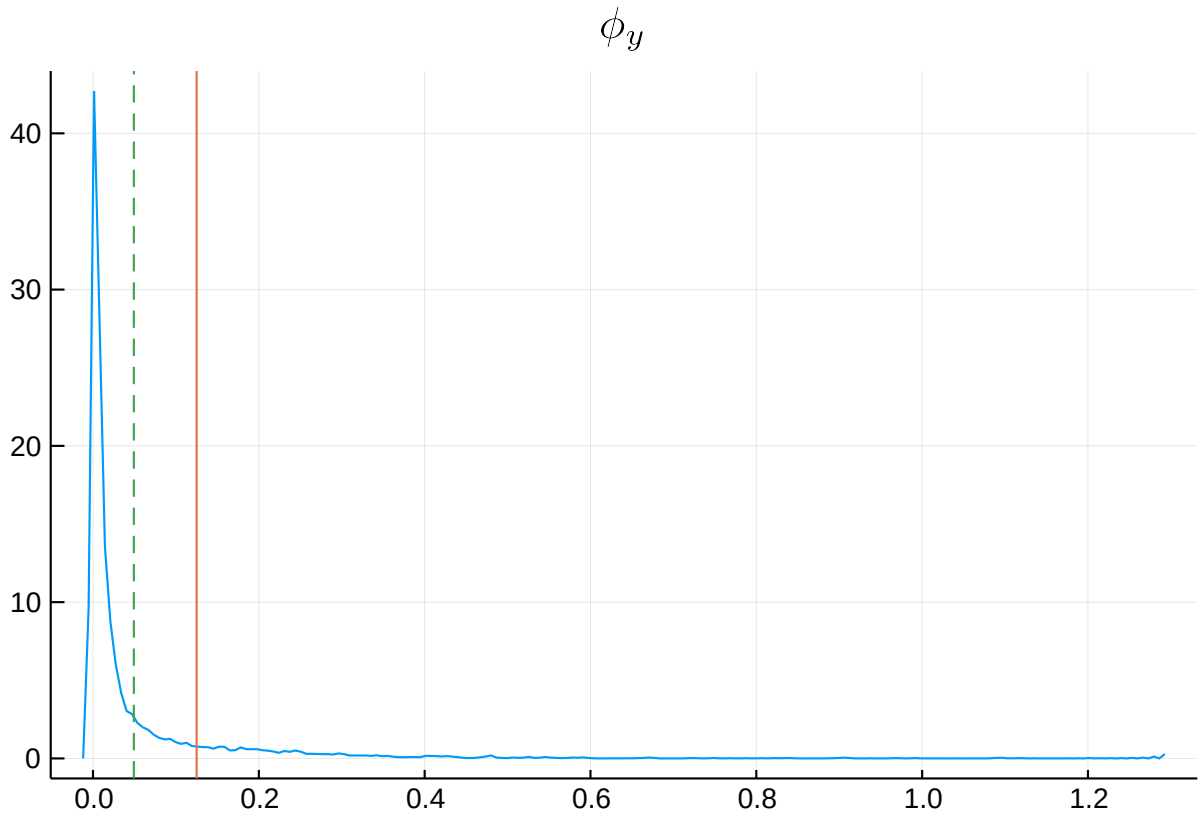
```
title!(L"\phi")
vline!([true_pars[:phi]])
vline!([mean(pars_convertidos[50001:100000,6])], line = :dash)
```




```
StatsPlots.density(pars_convertidos[50001:100000,7], legend = :none)
title!(L"\phi_{\pi}")
vline!([true_pars[:phi_pi]])
vline!([mean(pars_convertidos[50001:100000,7])], line = :dash)
```

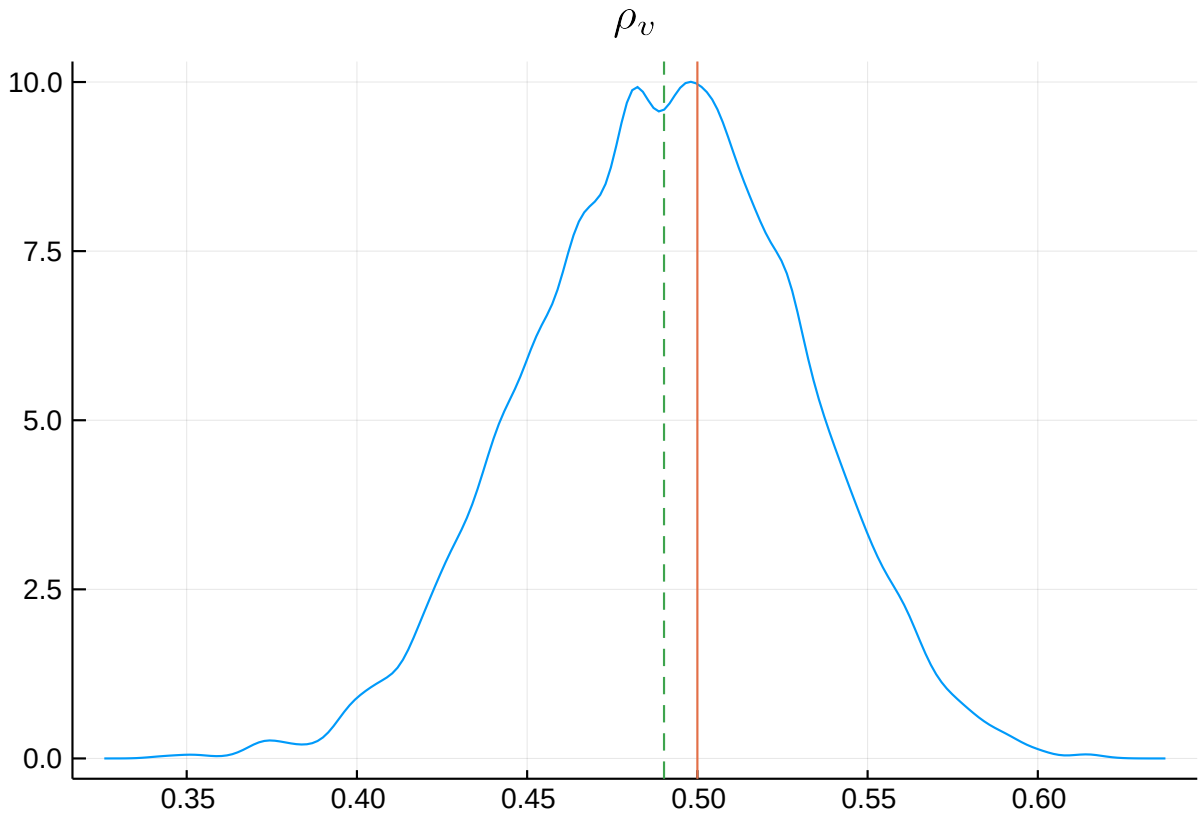



```
StatsPlots.density(pars_convertidos[50001:100000,8], legend = :none)
title!(L"\phi_y")
vline!([true_pars[:phi_y]])
vline!([mean(pars_convertidos[50001:100000,8])], line = :dash)
```




```
StatsPlots.density(pars_convertidos[50001:100000,9], legend = :none)
```

```
title!(L"\rho_v")
vline!([true_pars[:rho_v]])
vline!([mean(pars_convertidos[50001:100000,9])], line = :dash)
```



The algorithm is pretty sucessful. All means are close to the true value and so are the modes. The algorithm has a hard time recovering ϕ_y; however, it is surprisingly sucessful recovering ϵ, which is inside the expression for κ.

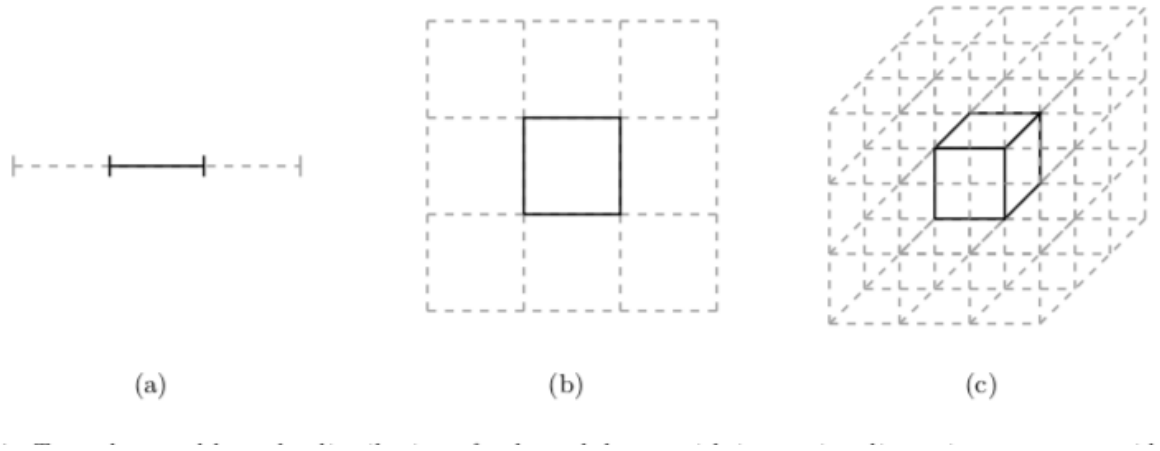
3.4 Hamiltonian Monte Carlo

Although Random Walk Metropolis Hasting (RWMH) is the main method used when estimating DSGEs by Bayesian methods, it is not the only MCMC algorithm. RWMH is a reliable and relatively simple algorithm to implement, and only requires evaluating the likelihood twice per iteration - which in our case is costly - and does not require knowing the marginal distribution like Gibb's Sampler. However, RWMH relies on a random walk to explore the parametric space: it may wander through regions with low density. This is even worse in high dimensional cases.Betancourt (2017) gives an intuitive example of what goes wrong, using Figure **????**:

Betancourt (2017) explains the figure as follows: in one dimension (figure (a)), a division in three parts of the line makes the total mass in the center partition equals 1/3; in $\mathbb{R}^2$ (figure (b)) we have in the center a mass of 1/9; in $\mathbb{R}^3$ (figure (c)), the mass on the center is just 1/27. The volume around the mode is negligible for high dimensional spaces. However, the density function is small outside the region outside mode. this generate a trade off: stay too near to the mode and you will never explore most of the volume; go too far and you will be exploring regions with low density.

The region that we want to explore is called the *typical set*. This is loosely the set that

Figure 1: Changes in volume outside



achieves a balance between the two points above, and it has a formal definition in terms of information theory. The problem is that RWMH wanders around the space of parameters and this does not guarantee that it will reach or stay in the typical set in finite time - the time we usually have for simulations.

Fortunately enough, we have a vector that always points toward the mode: the gradient. The Hamiltonian Monte Carlo (HMC) combines this information with the usual Metropolis Hasting algorithm to make sure that the algorithm explores the typical set. Betancourt (2017) uses as a metaphor putting a satellite in orbit: the satellite is attracted to the point with larger mass (the earth). We want our satellite to orbit earth, so we have to give enough impulse that it does not comes crash down on earth; however we don't want to give it *too* much impulse and start wandering around the solar system.

The heart of the Hamiltonian Monte Carlo is to add an additional variable, that is deterministic, that measures the impulse that is given for each parameter. This has to be choosen - as the scale parameter of the RWMH - to guarantee that we will stay in the typical set. This variable follows the dynamics of a Hamiltonian system of physics, which explains the name.

The implementation is not as easy as the RWMH, since we have to keep track of the impulse and update it accordingly the Hamiltonian of the system. Fortunately, there are plenty of implementations of HMC in Julia. We will use the package **DynamicHMC**, that follows the implementation suggested in Bethancourt's paper. We have to give to it the (numerator of) the posterior and it's gradient, and the package does everything else.

Getting the gradient is tricky. Gelman *et al.* (2014) explicitly acknowledge that "In practice the gradient must be computed analytically; numerical differentiation requires too many function evaluations to be computed analytically". It is hard to imagine *how* to compute the analytical derivate of each parameter, given the amount of computations that go into building the likelihood. Therefore, we have to rely on numerical differentiation. The usual method for differentiating, finite difference, requires a lot of function evaluation (at least two for each variable) and is prone to numerical errors - which can only be reduced by evaluating the function even more times. A first attempt using finite differences led to a running time of at least 12 hours.

However, there are better methods than finite differences. One of them is automatic differentiation. Although it builds from dual numbers and other concepts of algebra, that is not a subject usually dealt by economists, the idea is just to use the chain rule: to differentiate a

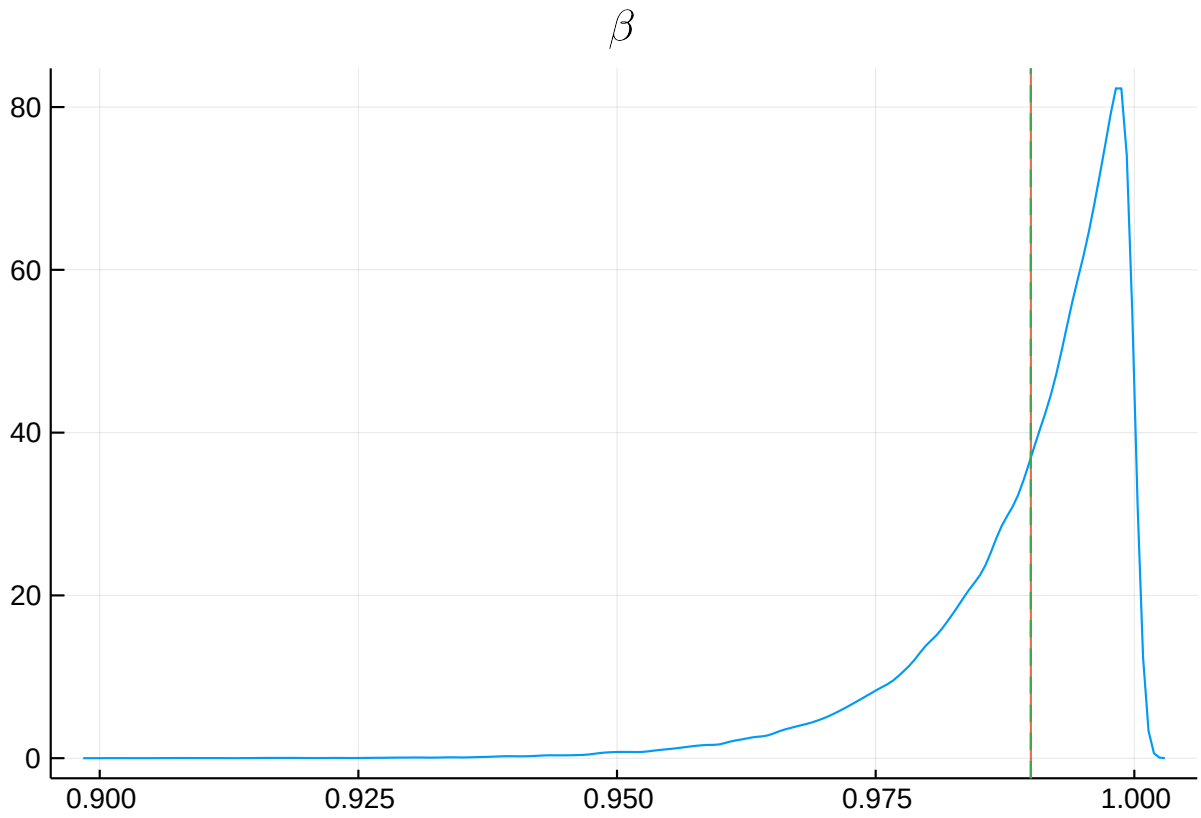
given function, differentiate each component that forms the function. This is both faster and more precise than finite differences and is implemented by a variety of packages in Julia. We use **Tracker**: some other packages for automatic differentiation do not differentiate over the Schur decomposition.

The code that estimates the DSGE using the likelihood we built above is:

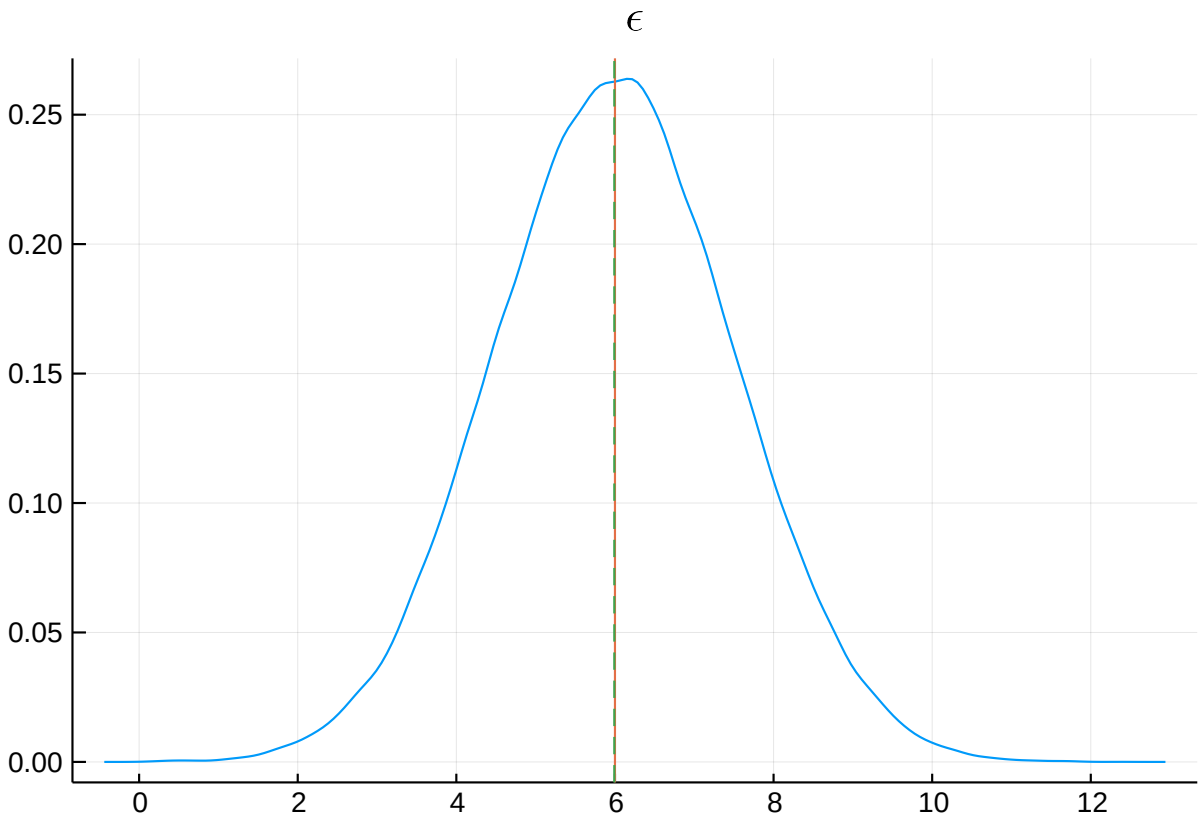
```
using Tracker, DynamicHMC

grad_p = ADgradient(:Tracker,P)

res = mcmc_with_warmup(Random.GLOBAL_RNG, grad_p,100000)
```

And let's show the densities:

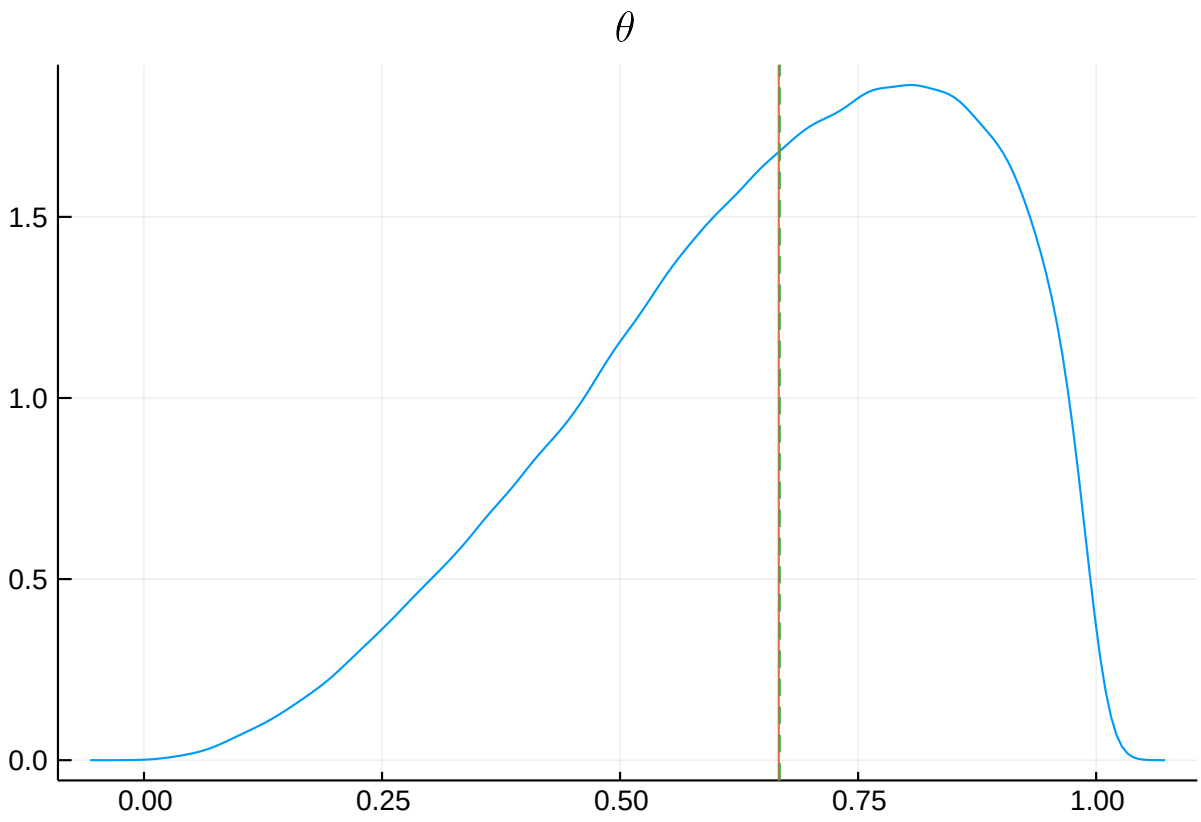
```
pars_hmc = load("data/hmc2.jld")["pars"]

StatsPlots.density(pars_hmc[:,1], legend =:none)
title!(L"\beta")
vline!([true_pars[:bet]])
vline!([mean(pars_hmc[:,1])], line = :dash)
```




```
StatsPlots.density(pars_hmc[:,2], legend =:none)
title!(L"\epsilon")
vline!([true_pars[:epsilon]])
vline!([mean(pars_hmc[:,2])], line = :dash)
```

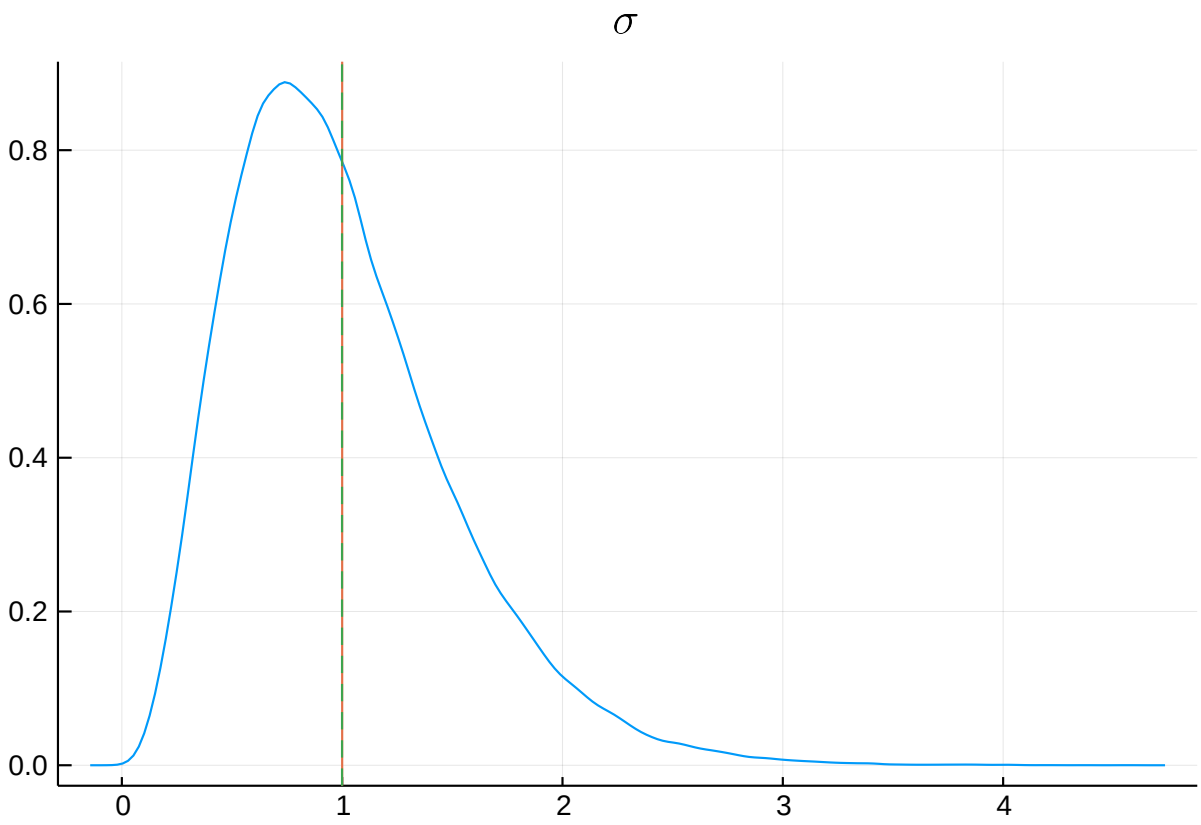



```
StatsPlots.density(pars_hmc[:,3], legend =:none)
title!(L"\theta")
vline!([true_pars[:theta]])
vline!([mean(pars_hmc[:,3])], line = :dash)
```

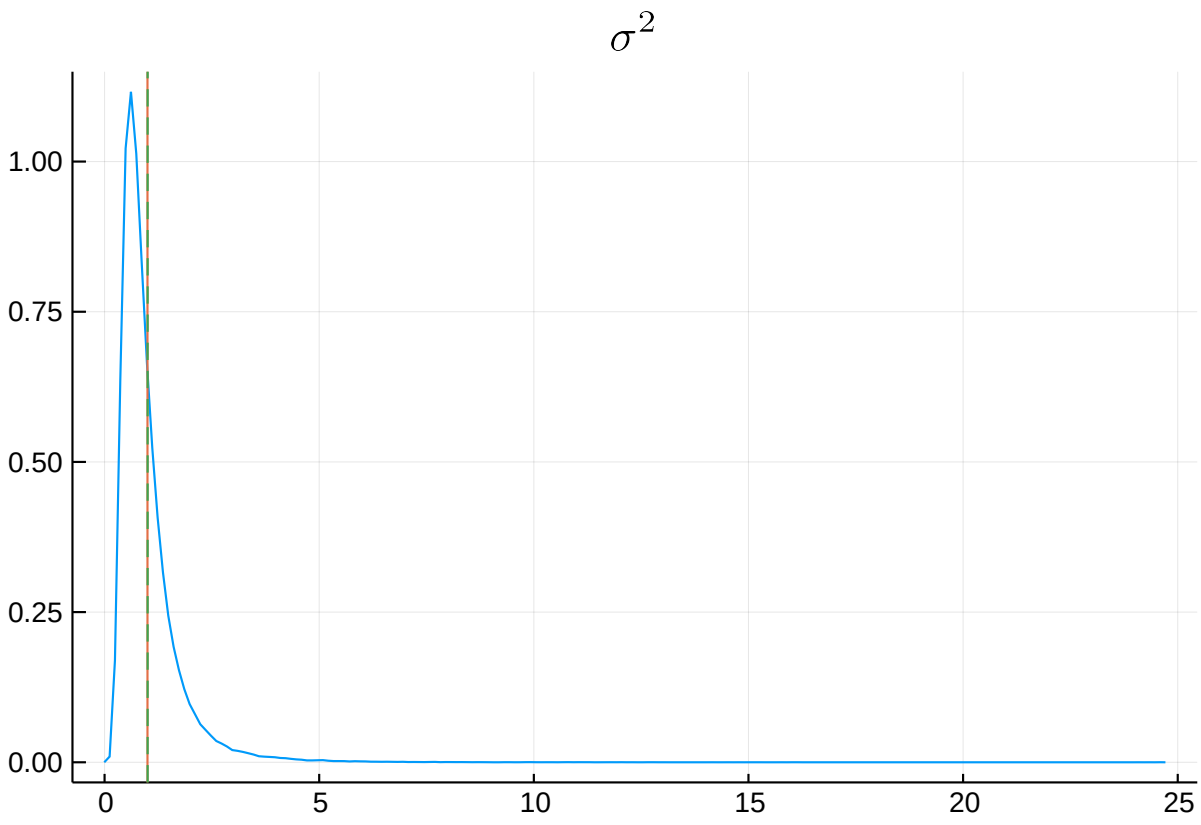



```
StatsPlots.density(pars_hmc[:,4], legend =:none)
```

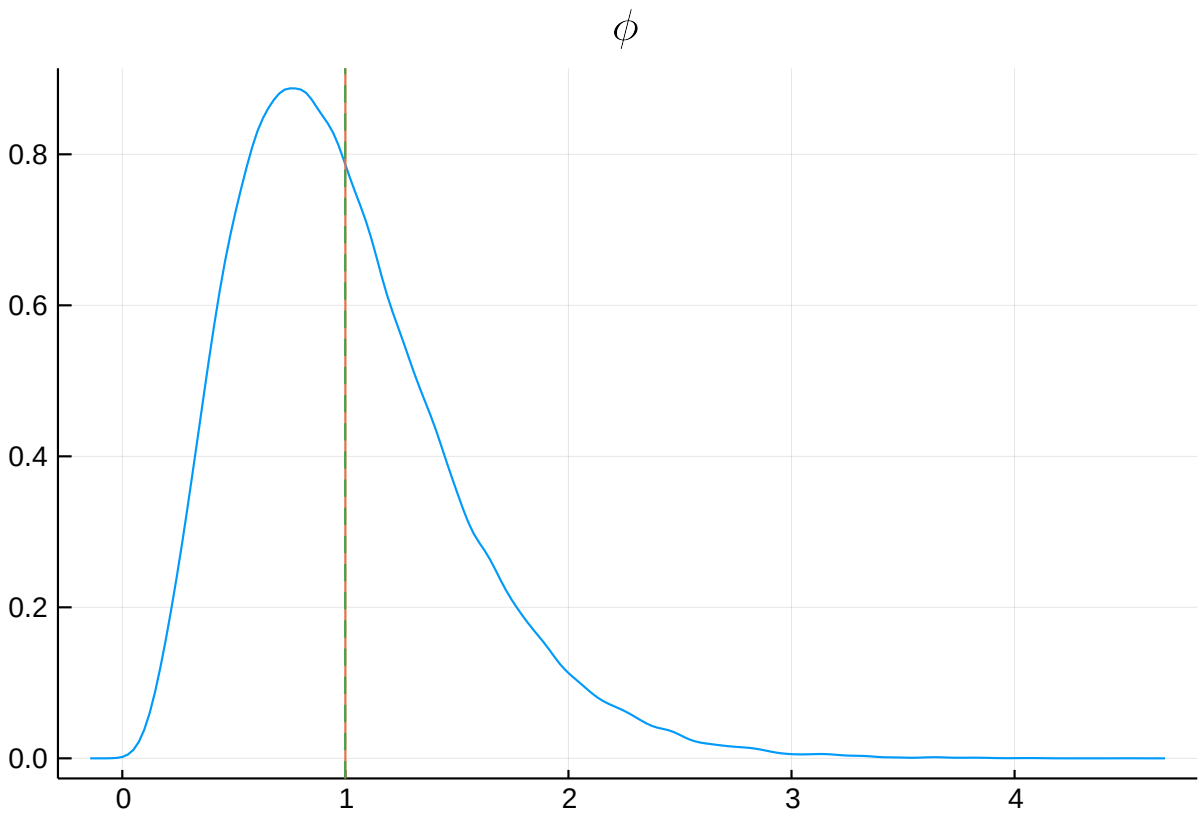
```
title!(L"\sigma")
vline!([true_pars[:sig]])
vline!([mean(pars_hmc[:,4])], line = :dash)
```




```
StatsPlots.density(pars_hmc[:,5], legend =:none)
title!(L"\sigma^2")
vline!([true_pars[:s2]])
vline!([mean(pars_hmc[:,5])], line = :dash)
```

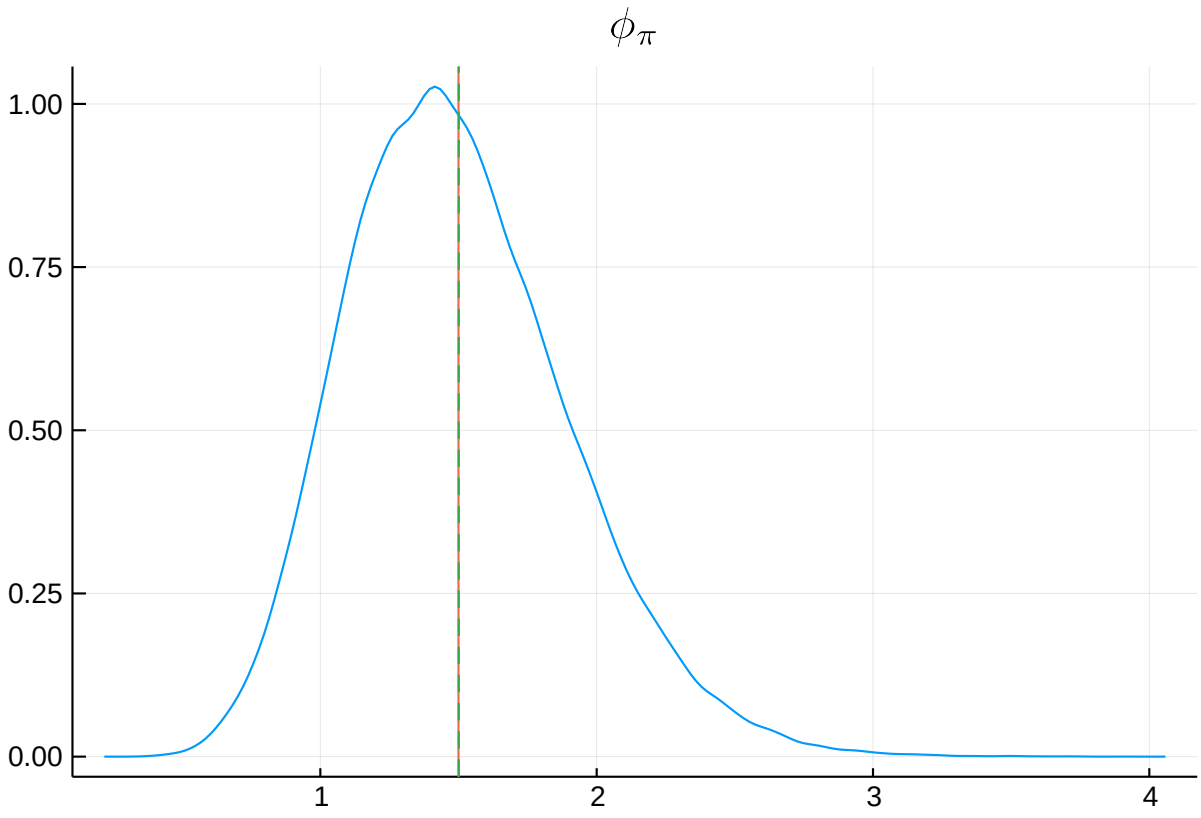



```
StatsPlots.density(pars_hmc[:,6], legend =:none)
title!(L"\phi")
vline!([true_pars[:phi]])
vline!([mean(pars_hmc[:,6])], line = :dash)
```

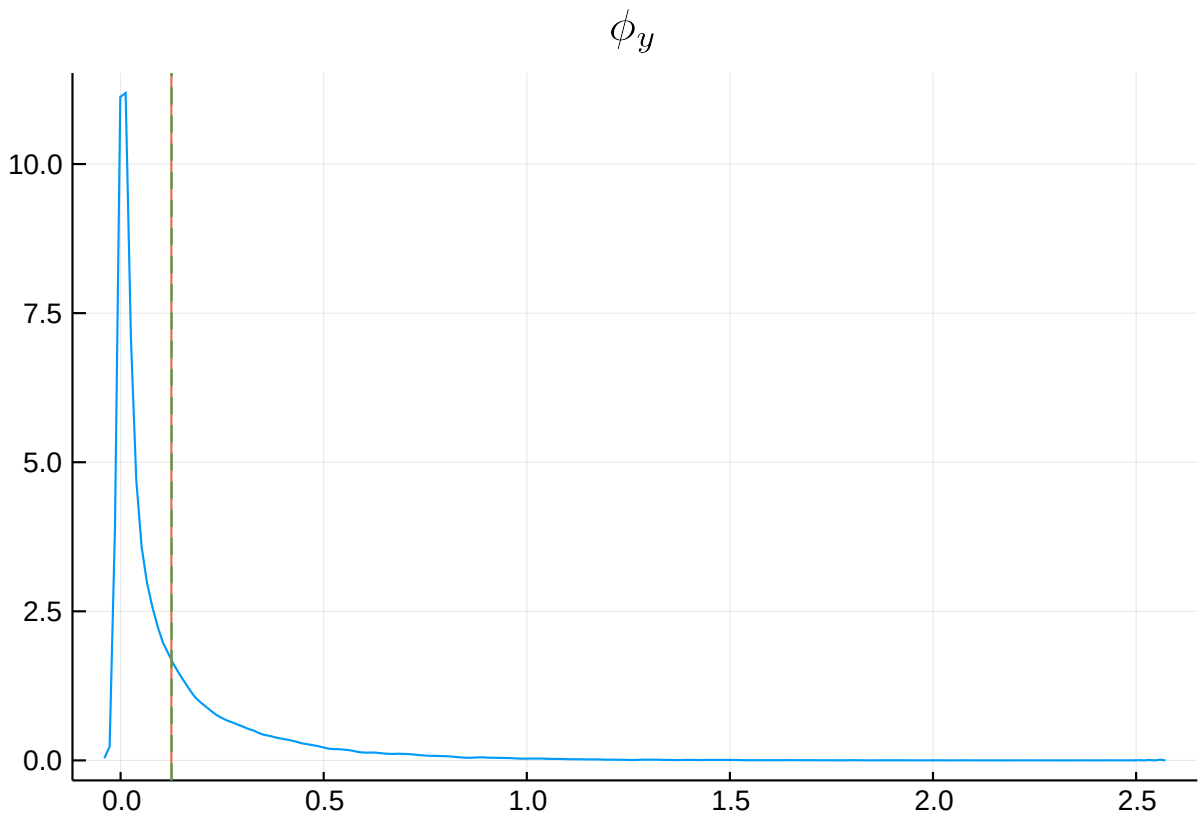



```
StatsPlots.density(pars_hmc[:,7], legend =:none)
```

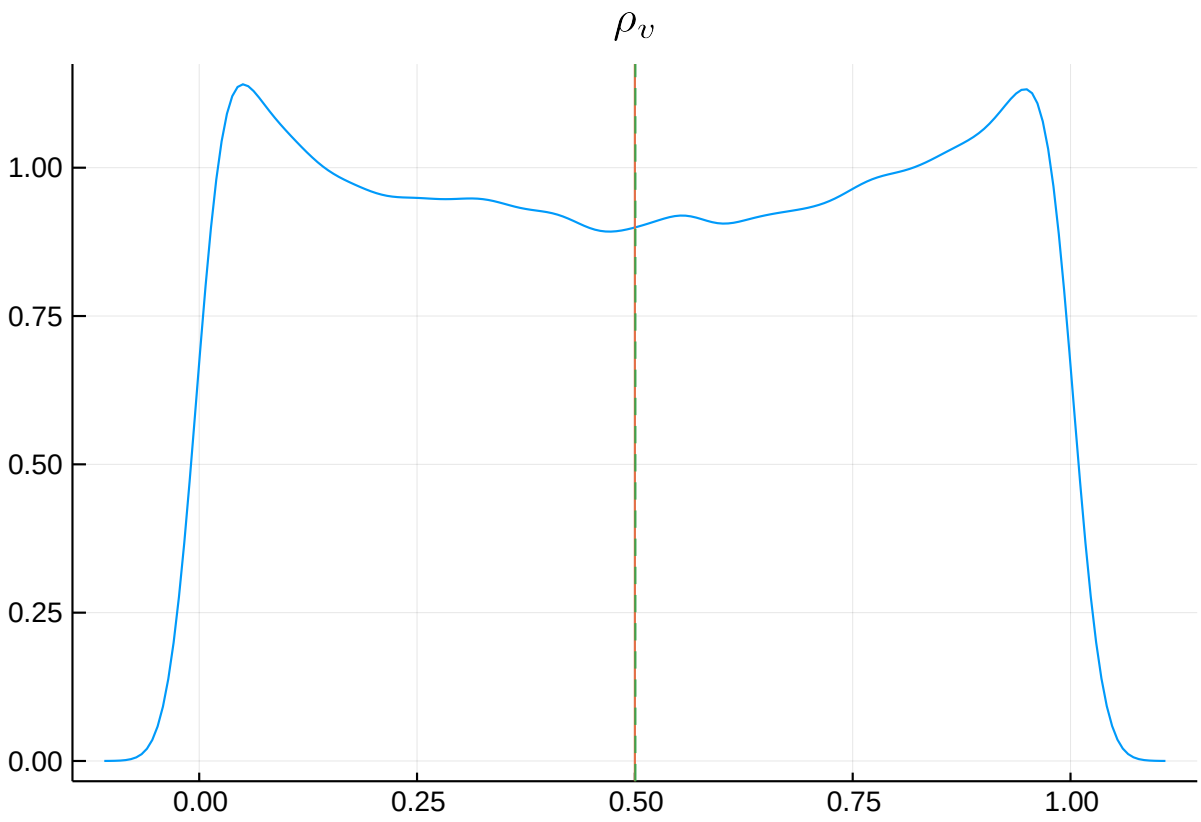
```
title!(L"\phi_{\pi}")
vline!([true_pars[:phi_pi]])
vline!([mean(pars_hmc[:,7])], line = :dash)
```




```
StatsPlots.density(pars_hmc[:,8], legend =:none)
title!(L"\phi_y")
vline!([true_pars[:phi_y]])
vline!([mean(pars_hmc[:,8])], line = :dash)
```




```
StatsPlots.density(pars_hmc[:,9], legend =:none)
title!(L"\rho_v")
vline!([true_pars[:rho_v]])
vline!([mean(pars_hmc[:,9])], line = :dash)
```



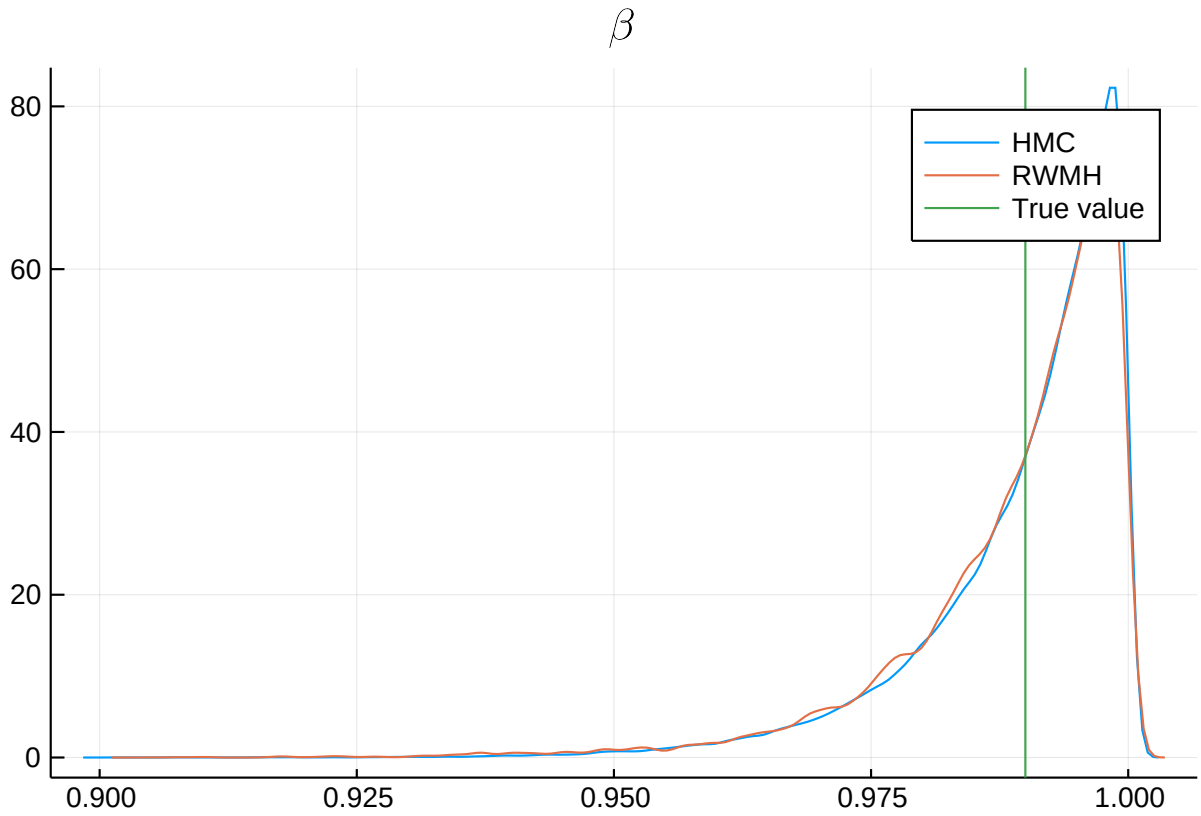
The results are mostly positive, with means always close to the true value of the parameters.

The distribution of ρ_v is problematic, being a lot more disperse then their RWMH counterpart. We conduct more comparisons in what follows.

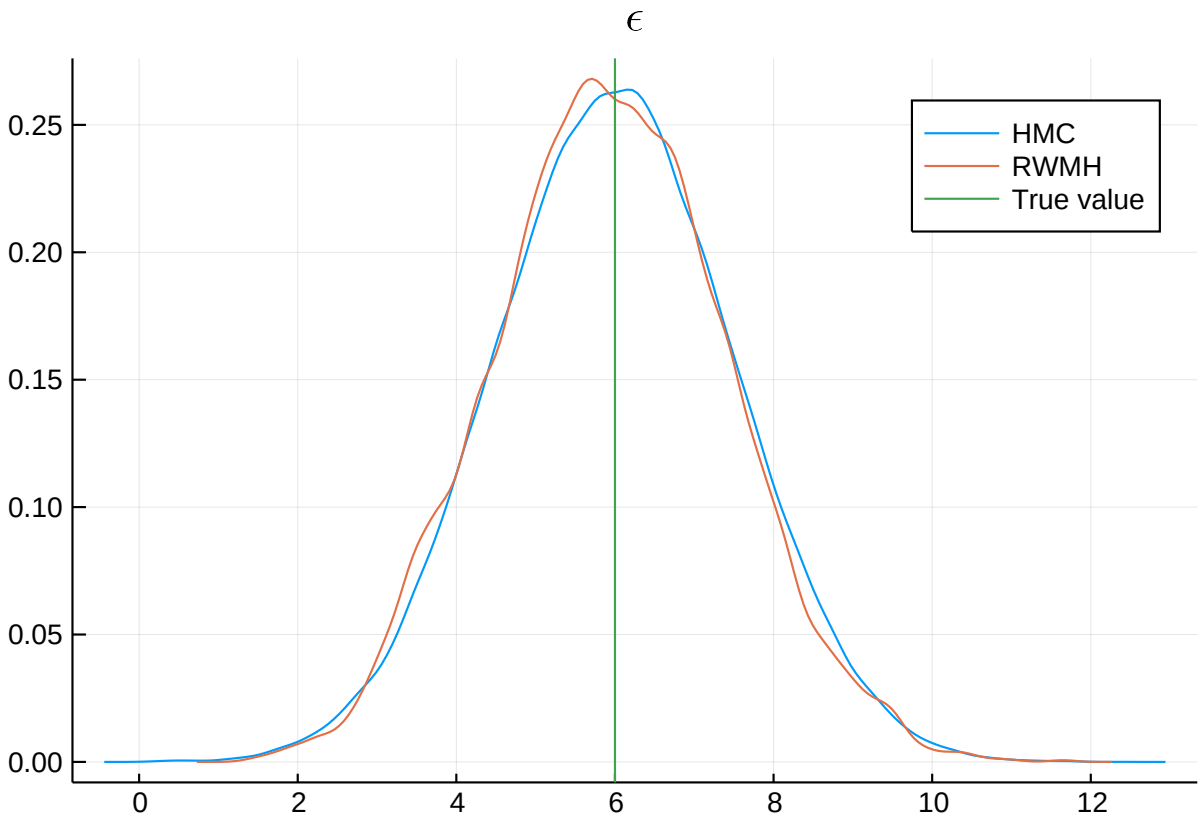
3.4.1 Comparing HMC and RWMH

Two things are important: precision and speed. We plot the distribution of the parameters from both the MCMC and the HMC to get an idea of their accuracy:

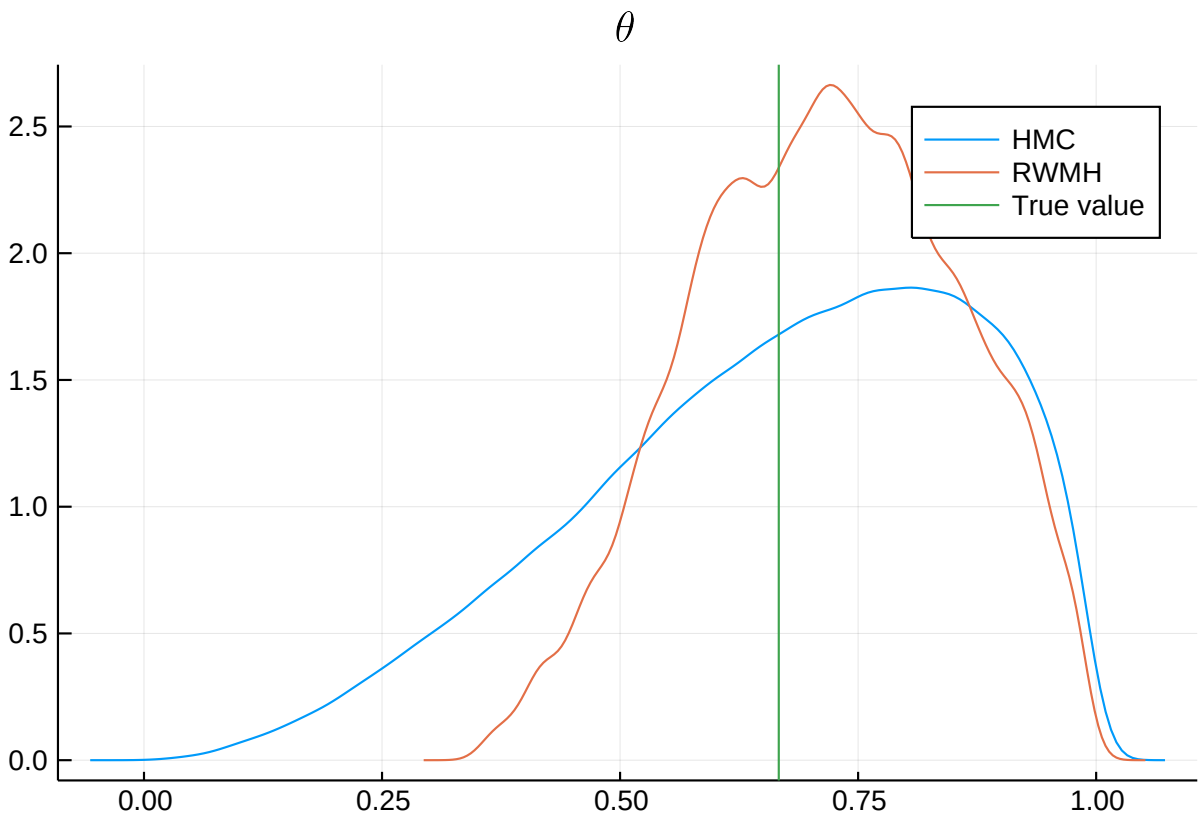
```
pars_hmc = load("data/hmc2.jld")["pars"]

StatsPlots.density(pars_hmc[:,1], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,1], label = "RWMH")
title!(L"\beta")
vline!([true_pars[:bet]], label = "True value")
```




```
StatsPlots.density(pars_hmc[:,2], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,2], label = "RWMH")
title!(L"\epsilon")
vline!([true_pars[:epsilon]], label = "True value")
```

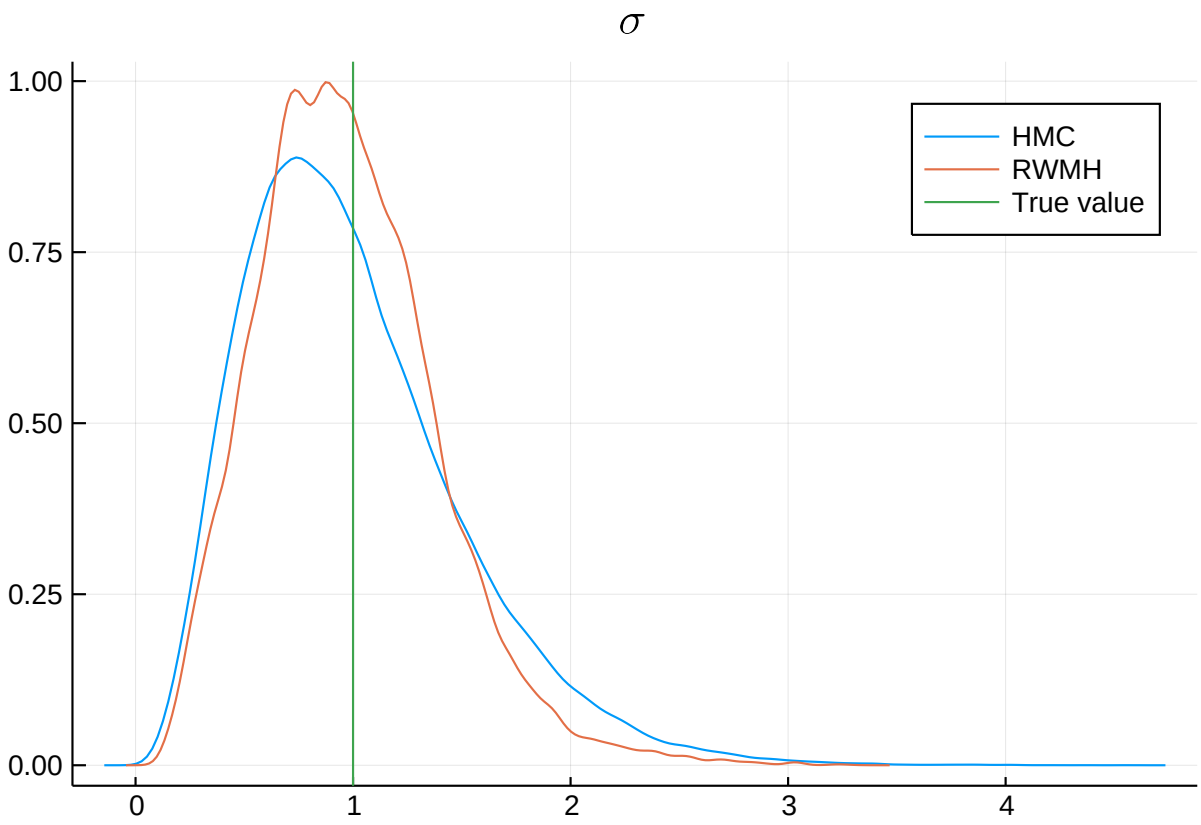



```
StatsPlots.density(pars_hmc[:,3], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,3], label = "RWMH")
title!(L"\theta")
vline!([true_pars[:theta]], label = "True value")
```

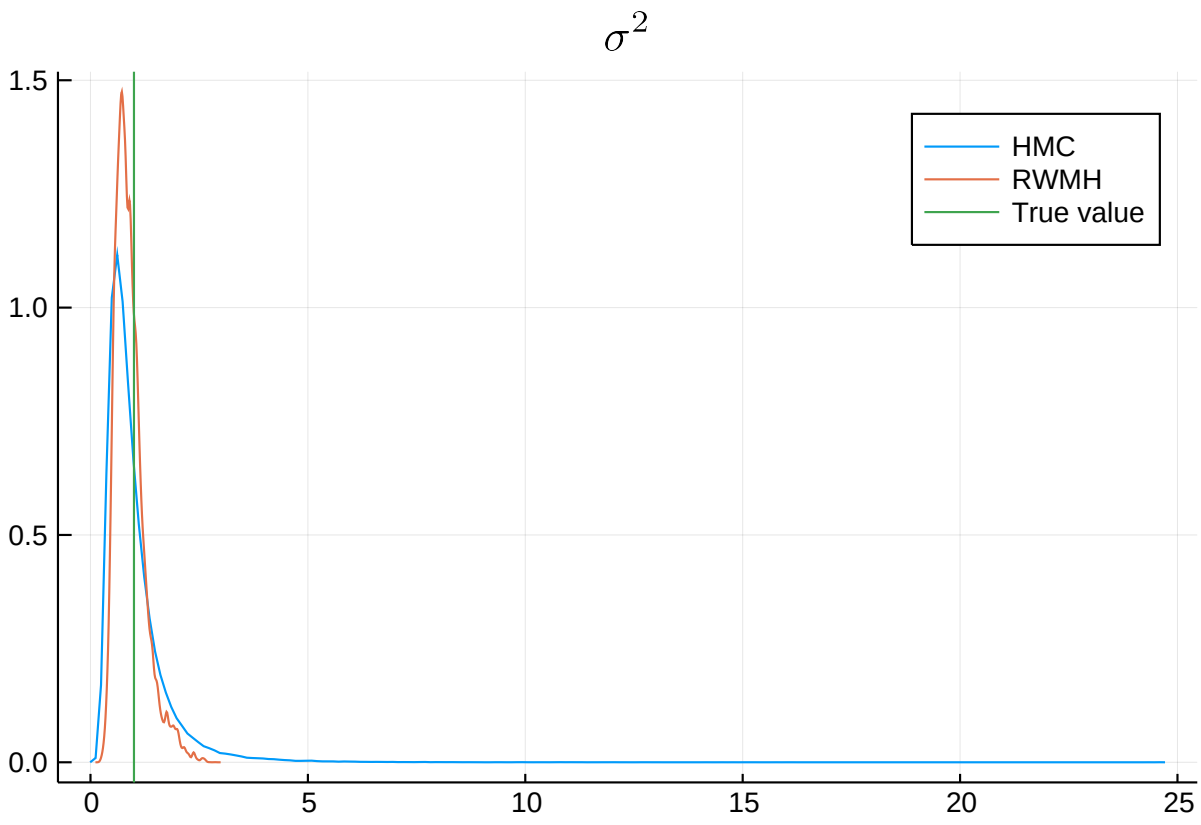



```
StatsPlots.density(pars_hmc[:,4], label = "HMC")
```

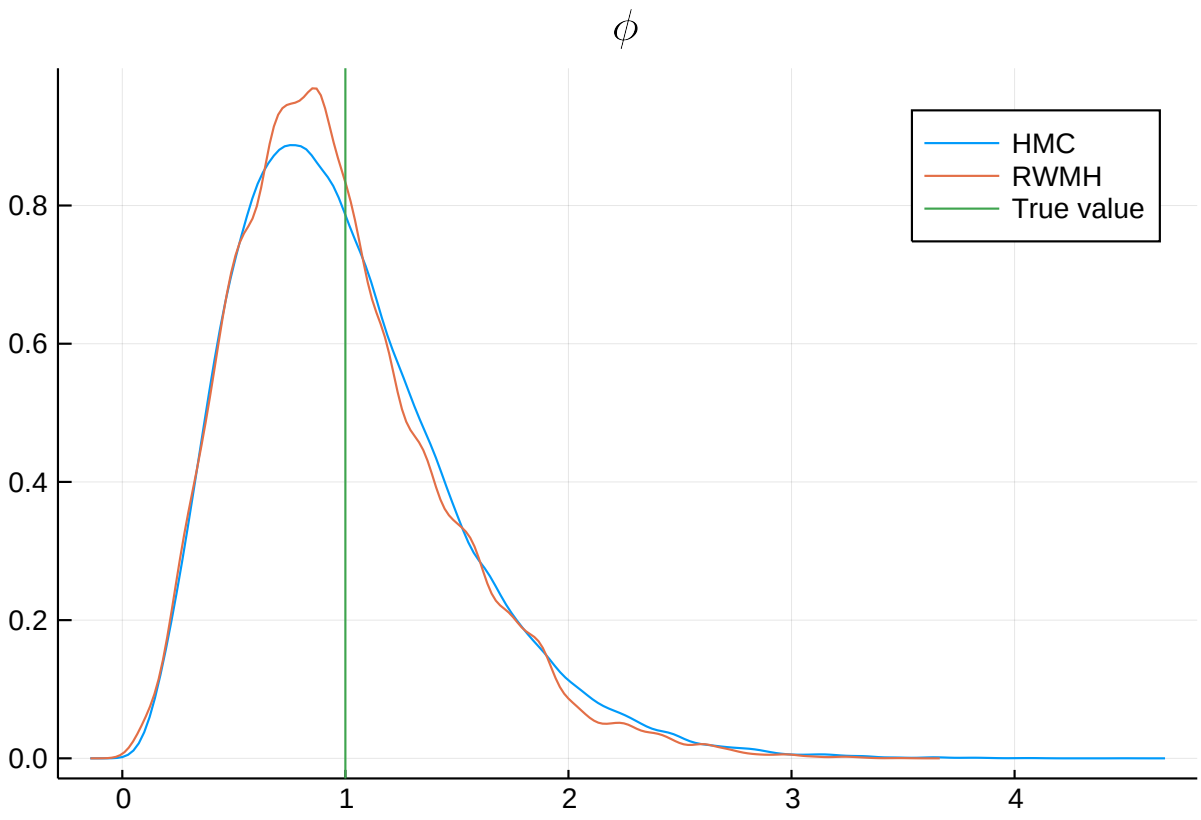
```
StatsPlots.density!(pars_convertidos[50001:100000,4], label = "RWMH")
title!(L"\sigma")
vline!([true_pars[:sig]], label = "True value")
```




```
StatsPlots.density(pars_hmc[:,5], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,5], label = "RWMH")
title!(L"\sigma^2")
vline!([true_pars[:s2]], label = "True value")
```

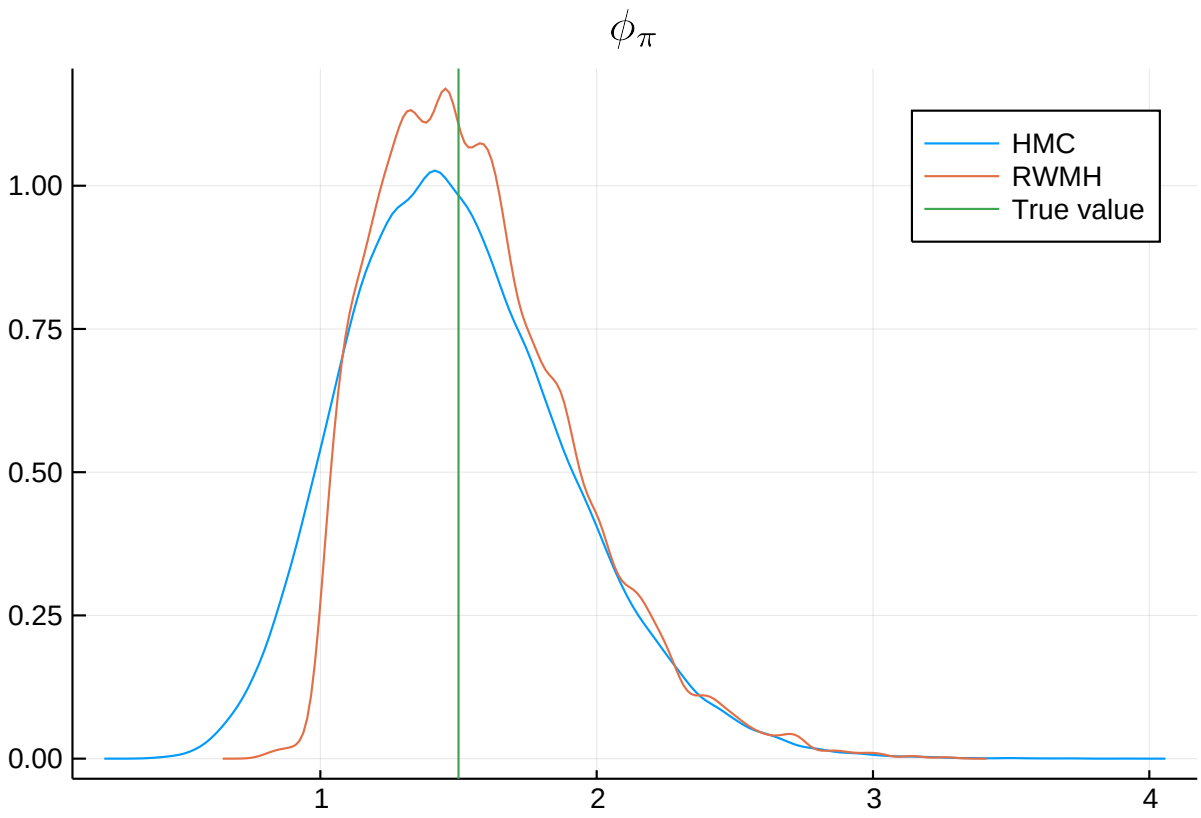



```
StatsPlots.density(pars_hmc[:,6], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,6], label = "RWMH")
title!(L"\phi")
vline!([true_pars[:phi]], label = "True value")
```

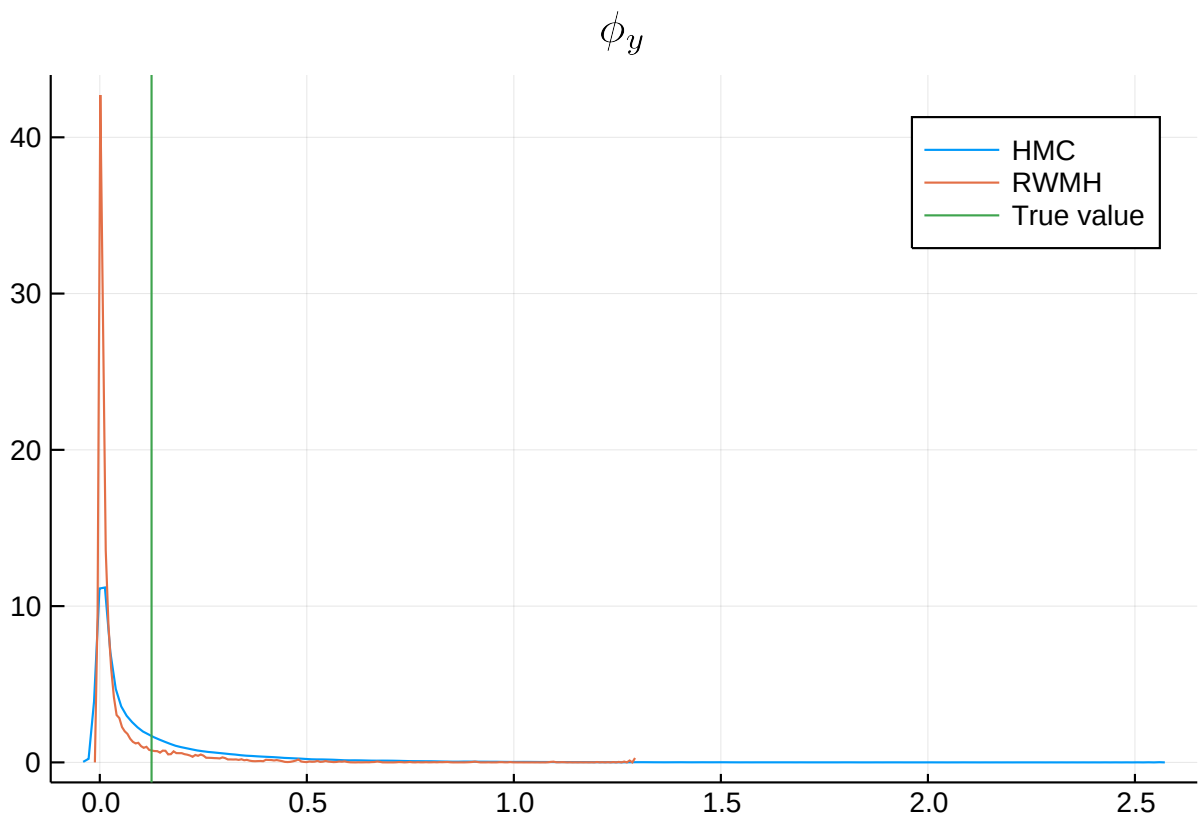



```
StatsPlots.density(pars_hmc[:,7], label = "HMC")
```

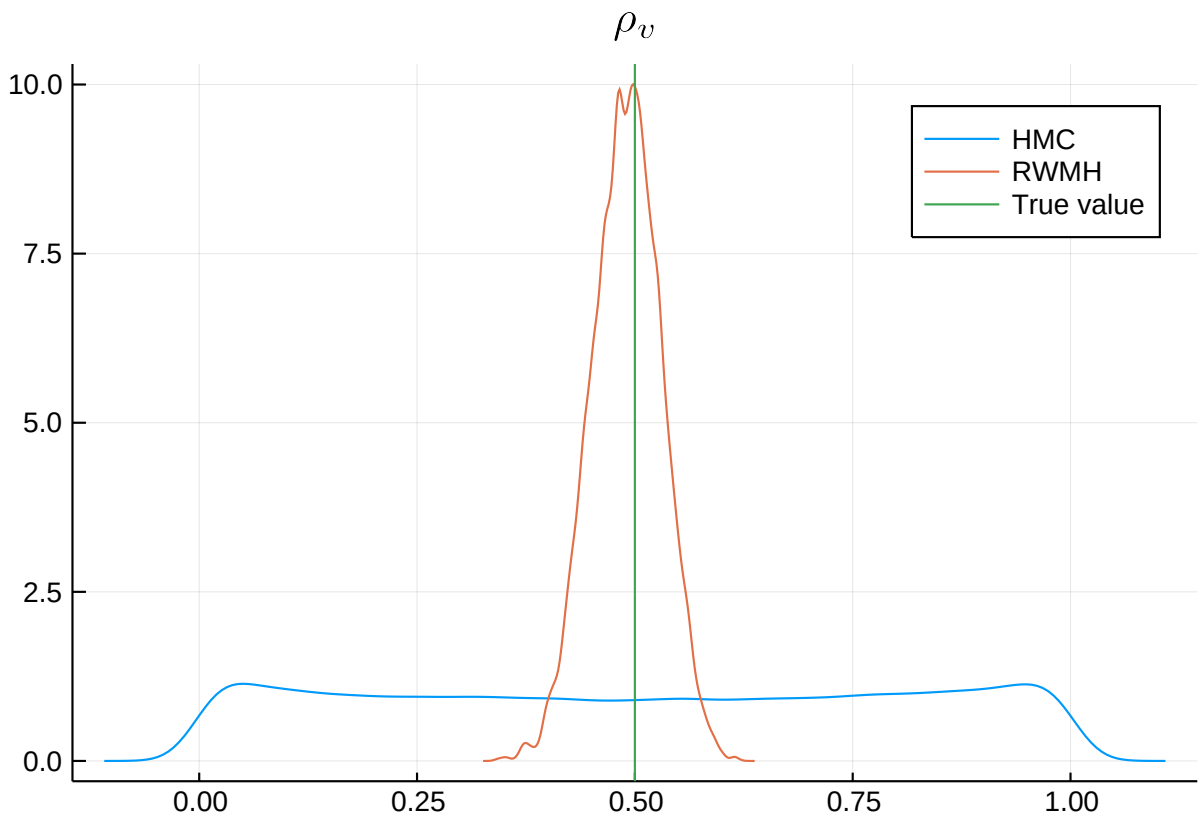
```
StatsPlots.density!(pars_convertidos[50001:100000,7], label = "RWMH")
title!(L"\phi_{\pi}")
vline!([true_pars[:phi_pi]], label = "True value")
```




```
StatsPlots.density(pars_hmc[:,8], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,8], label = "RWMH")
title!(L"\phi_y")
vline!([true_pars[:phi_y]], label = "True value")
```




```
StatsPlots.density(pars_hmc[:,9], label = "HMC")
StatsPlots.density!(pars_convertidos[50001:100000,9], label = "RWMH")
title!(L"\rho_v")
vline!([true_pars[:rho_v]], label = "True value")
```



On the time front, we will do two sets of simulations: one with 10 thousand replications

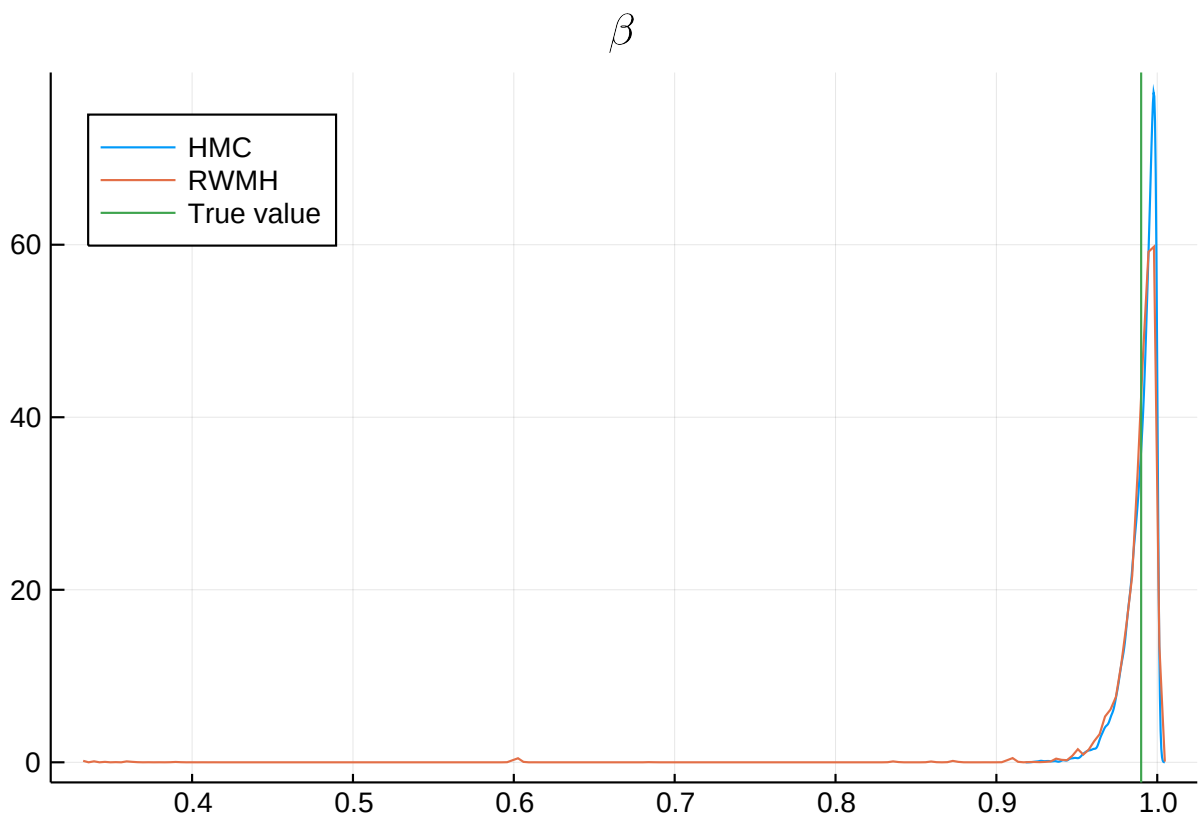
and another with 100 thousand. We would expect that the time difference would be around 10 times, and it is in the ballpark. While we let the HMC procedure choose the optimal parameter, and therefore the time before the actually iteration begins enters in our time figure, we feed the optimal parameter to the MCMC without taking this time into account in the table bellow. The time takes into account the warm up in both case. We conduct this simulations in a Dell Vostro with an Intel dual core i5-3210 at 2.5 Ghz, and with 6 GB of ram. The OS is a Linux flavour. Since we are more interested in the relative performance, this facts are by themselves of little importance. The times are, in seconds:

Table 1: Time, in seconds, for estimating the models using RWMH and HMC

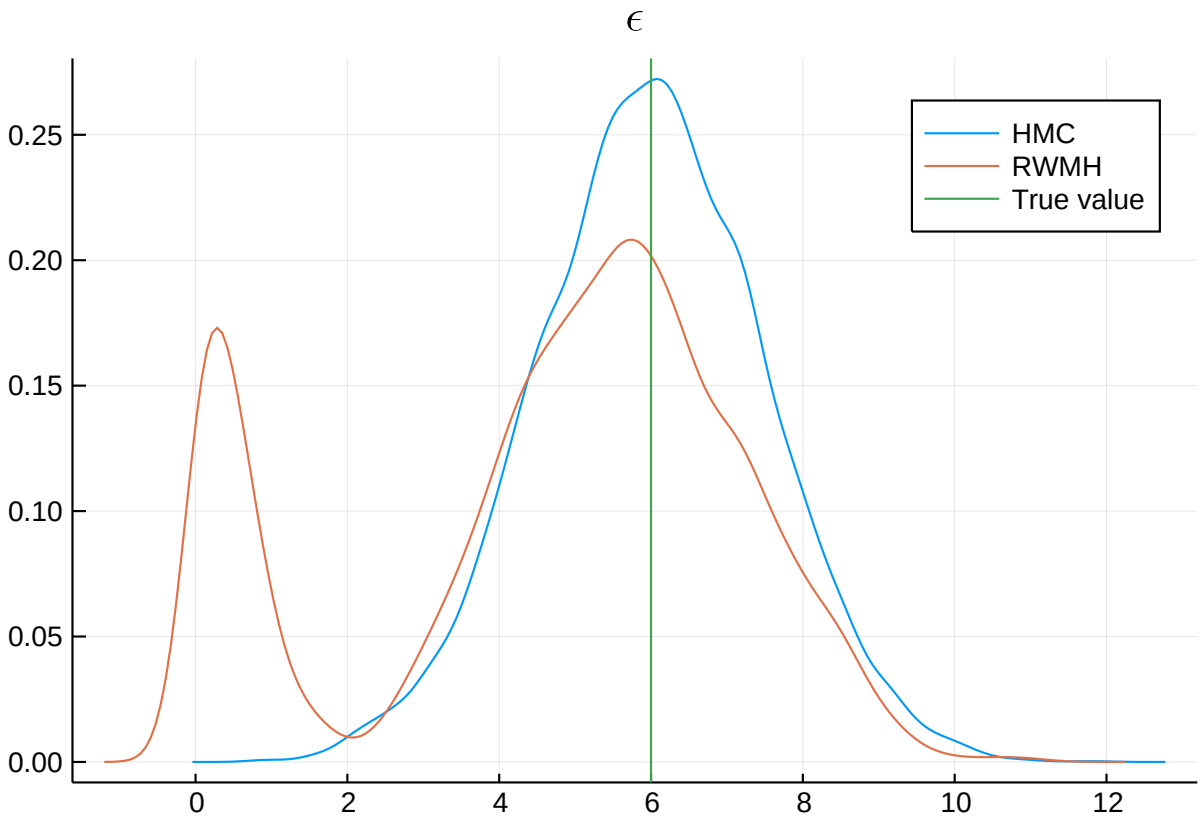
Replications	RWMH	HMC
10,000	252	46
100,000	2632	400

Hamiltonian Monte carlo is not just faster: it is *a lot faster*. This hides fundamental differences between the two methods: the number of iterations necessary to obtain convergence. To illustrate this point, we show the results of only ten thousand iterations of each algorithm:

```
pars_aceitos = load("data/mcmc10k.jld")["pars"]

yy,shocks = simulate_dsge(GAMMA_0,GAMMA_1,PSI,PI,500)

pars_convertidos10k = zeros(10000,9)

for i in 1:10000
    @unpack bet,epsilon,theta,sig,s2,phi,phi_pi,phi_y,rho_v =
TransformVariables.transform(t,pars_aceitos[i,2:10])
    pars_convertidos10k[i,:] = [bet,epsilon,theta,sig,s2,phi,phi_pi,phi_y,rho_v]#(bet =
pars_aceitos[i,2], epsilon = pars_aceitos[i,3], theta = pars_aceitos[i,4],sig =
pars_aceitos[i,5],s2 = pars_aceitos[i,6],phi = pars_aceitos[i,7],phi_pi =
pars_aceitos[i,8],phi_y = pars_aceitos[i,9],rho_v = pars_aceitos[i,10]))
end

pars_hmc10k = load("data/hmc10k.jld")["pars"]

StatsPlots.density(pars_hmc10k[:,1], label = "HMC", legend = :topleft)
StatsPlots.density!(pars_convertidos10k[1:10000,1], label = "RWMH")
title!(L"\beta")
vline!([true_pars[:bet]], label = "True value")
```

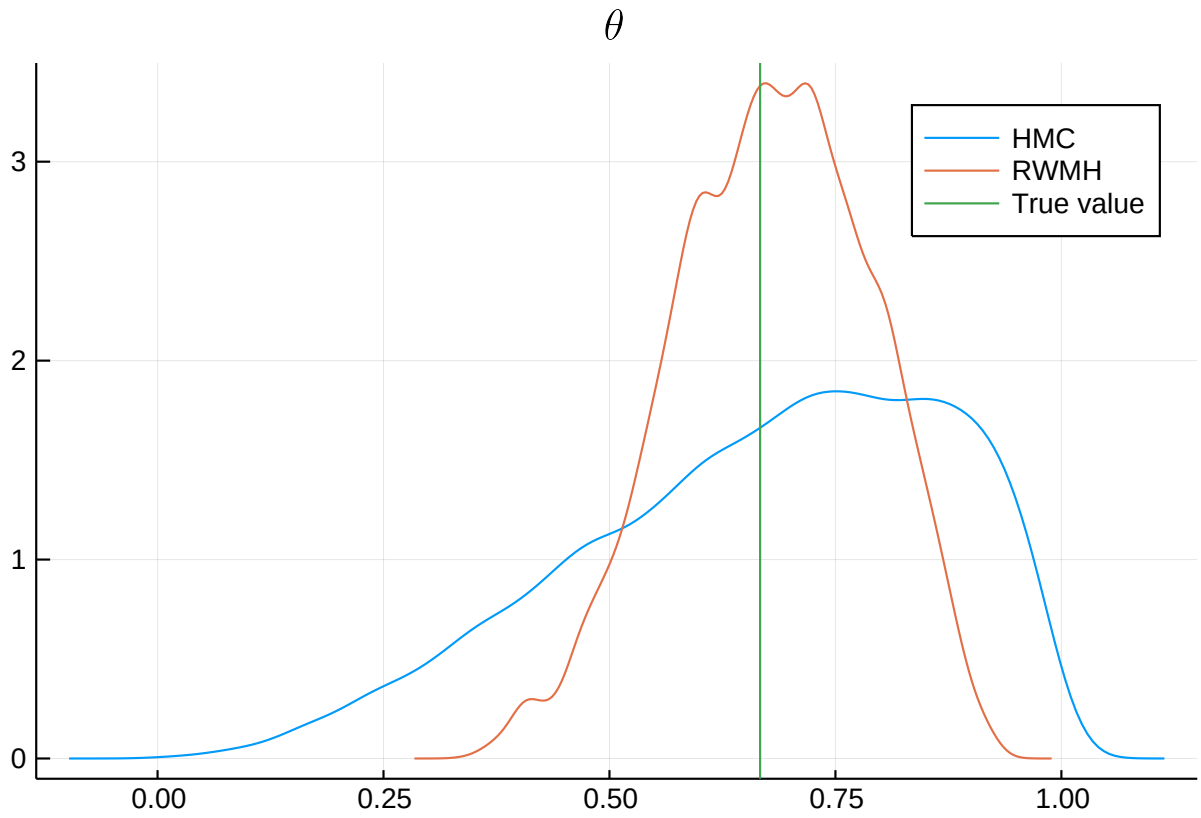



```
StatsPlots.density(pars_hmc10k[:,2], label = "HMC")
StatsPlots.density!(pars_convertidos10k[1:10000,2], label = "RWMH")
title!(L"\epsilon")
vline!([true_pars[:epsilon]], label = "True value")
```

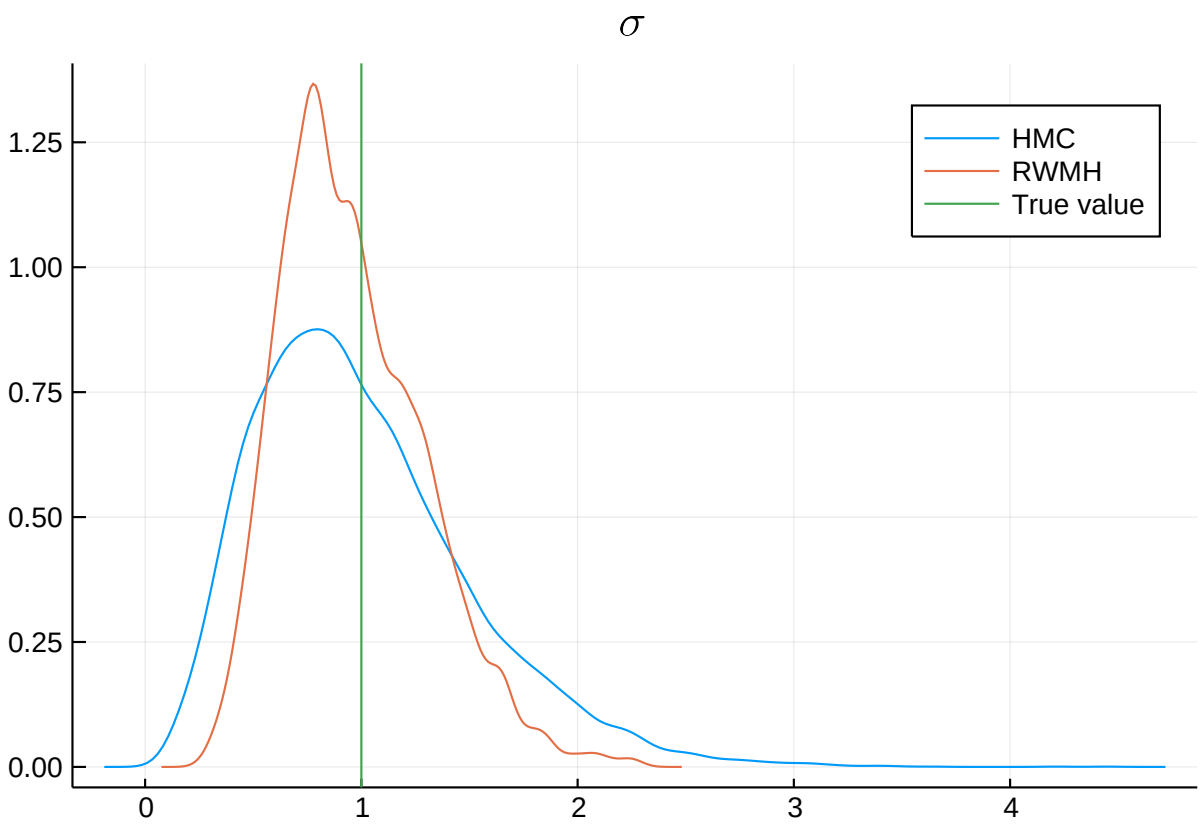



```
StatsPlots.density(pars_hmc10k[:,3], label = "HMC")
```

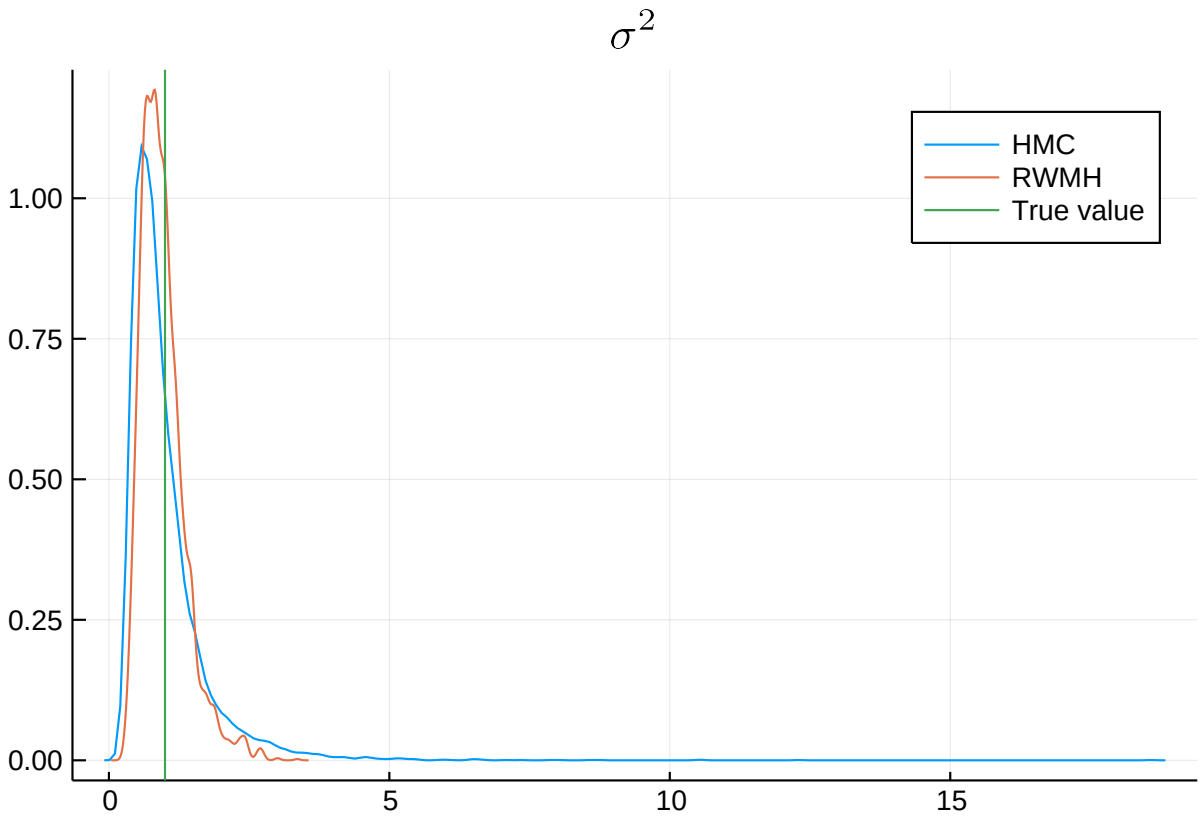
```
StatsPlots.density!(pars_convertidos10k[1:10000,3], label = "RWMH")
title!(L"\theta")
vline!([true_pars[:theta]], label = "True value")
```




```
StatsPlots.density(pars_hmc10k[:,4], label = "HMC")
StatsPlots.density!(pars_convertidos10k[1:10000,4], label = "RWMH")
title!(L"\sigma")
vline!([true_pars[:sig]], label = "True value")
```

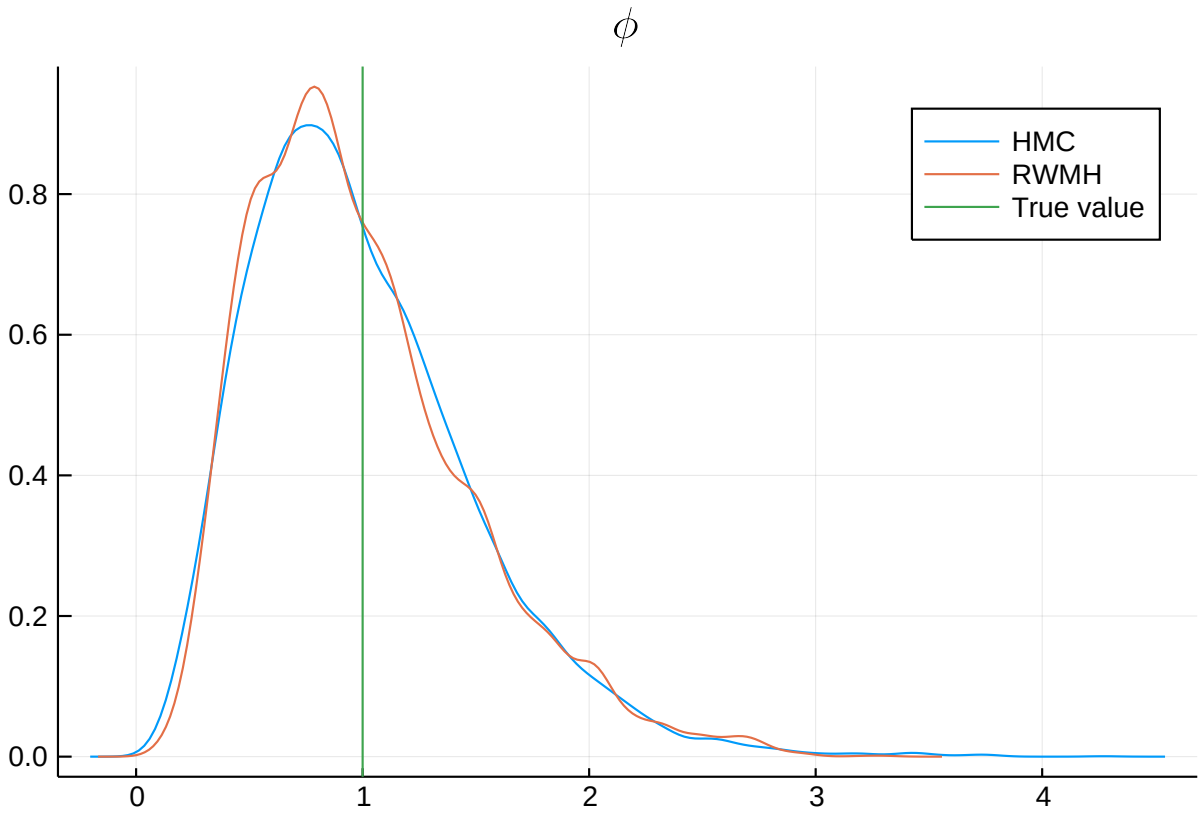



```
StatsPlots.density(pars_hmc10k[:,5], label = "HMC")
StatsPlots.density!(pars_convertidos10k[1:10000,5], label = "RWMH")
title!(L"\sigma^2")
vline!([true_pars[:s2]], label = "True value")
```

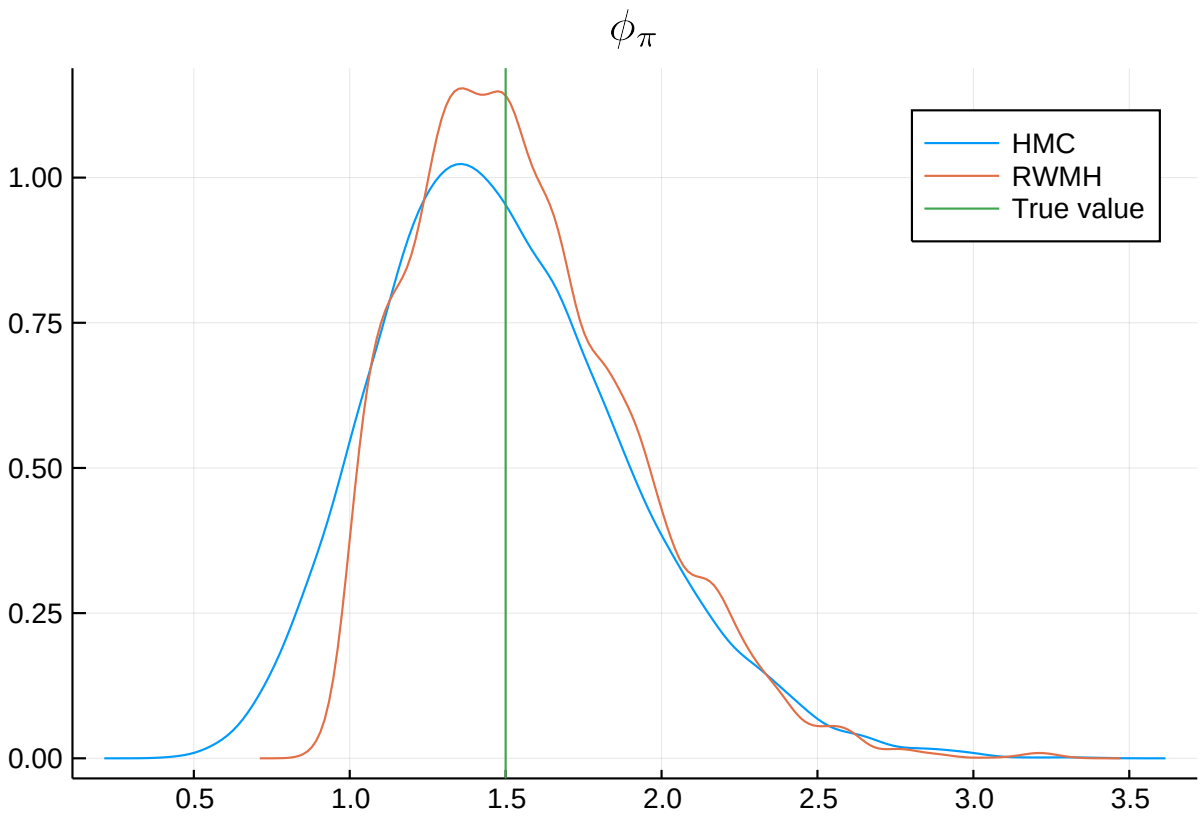



```
StatsPlots.density(pars_hmc10k[:,6], label = "HMC")
```

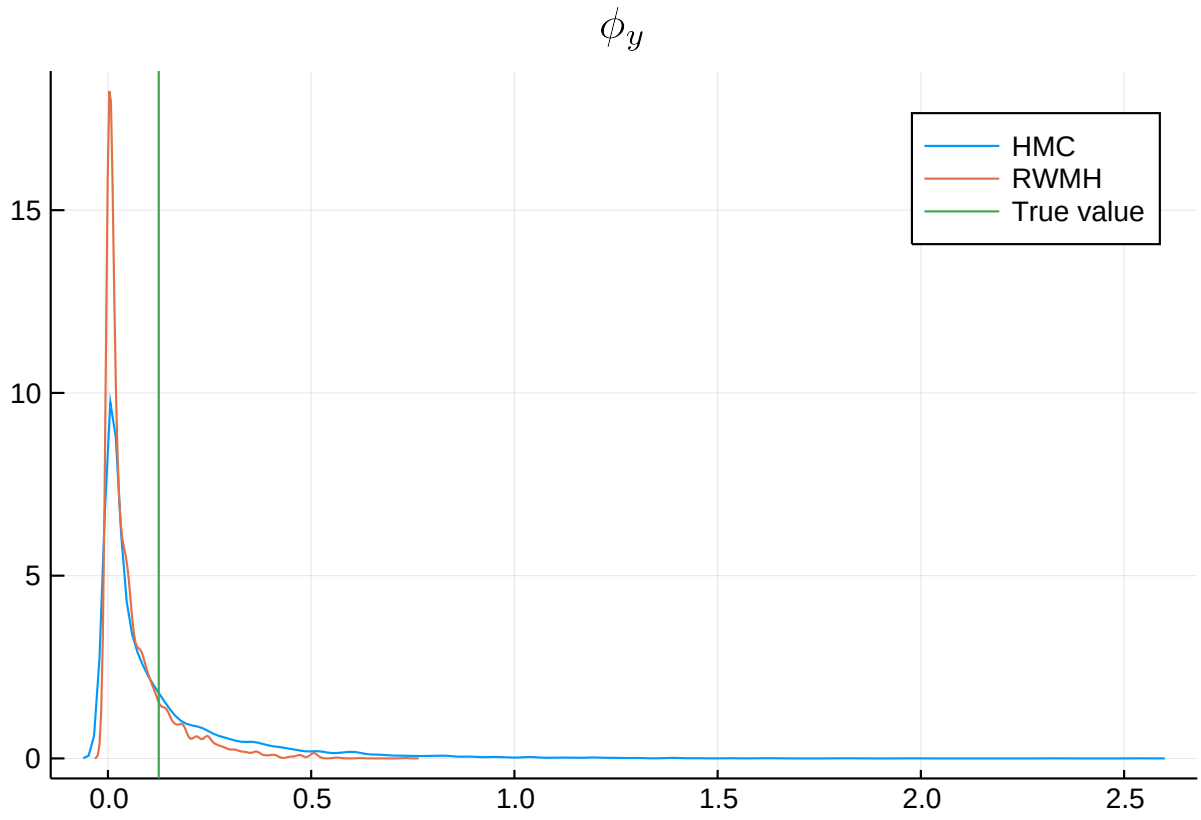
```
StatsPlots.density!(pars_convertidos10k[1:10000,6], label = "RWMH")
title!(L"\phi")
vline!([true_pars[:phi]], label = "True value")
```




```
StatsPlots.density(pars_hmc10k[:,7], label = "HMC")
StatsPlots.density!(pars_convertidos10k[1:10000,7], label = "RWMH")
title!(L"\phi_{\pi}")
vline!([true_pars[:phi_pi]], label = "True value")
```

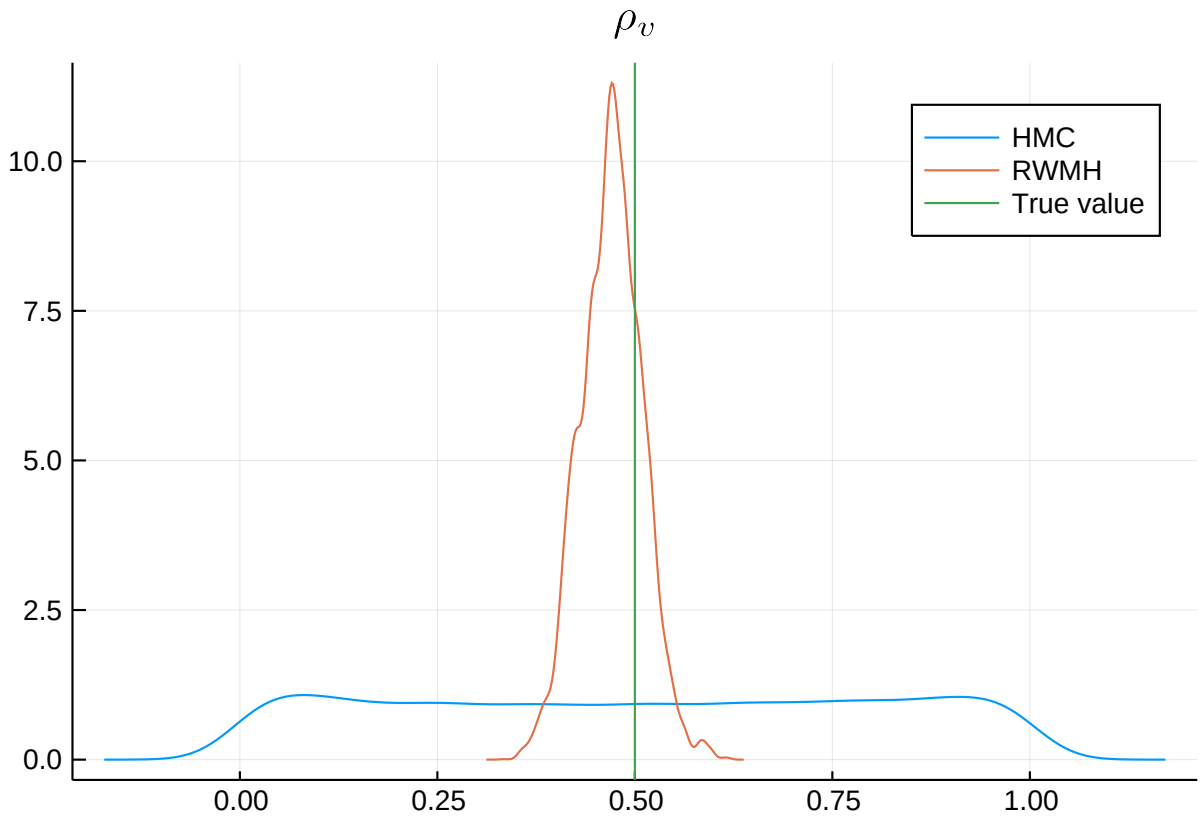



```
StatsPlots.density(pars_hmc10k[:,8], label = "HMC")
StatsPlots.density!(pars_convertidos10k[1:10000,8], label = "RWMH")
title!(L"\phi_y")
vline!([true_pars[:phi_y]], label = "True value")
```




```
StatsPlots.density(pars_hmc10k[:,9], label = "HMC")
```

```
StatsPlots.density!(pars_convertidos10k[1:10000,9], label = "RWMH")
title!(L"\rho_v")
vline!([true_pars[:rho_v]], label = "True value")
```



Random Walk Metropolis Hasting distributions are never better behaved than the ones obtained from HMC - with the notable exception of the distribution of ρ_vHowever, the ρ_v is not well behaved even with a hundred thousand iterations, so this is not due to the number of iterations.

References

Betancourt, Michael. 2017. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*.

Canova, Fabio. 2011. *Methods for applied macroeconomic research*.

Galí, Jordi. 2009. *Monetary policy, inflation, and the business cycle: An introduction to the new Keynesian framework*.

Gelman, Andrew, Carlin, John B B, Stern, Hal S S, & Rubin, Donald B B. 2014. *Bayesian Data Analysis, Third Edition (Texts in Statistical Science)*.

Keesman, Karel J. 2011. System identification: An introduction. *In: Advanced Textbooks in Control and Signal Processing*.

Miao, Jianjun. 2014. *Economic dynamics in discrete time*. MIT press.

Sims, Christopher A. 2002. Solving Linear Rational Expectations Models. *Computational Economics*.