

Estimation of DSGE Models in Julia

Course 5 — Short Macroeconomics Course Using Julia

November 27, 2025

Outline

- 1 Motivation and Overview
- 2 RBC Model in `MacroModelling.jl`
- 3 Turing: quick intro
- 4 Bayesian Estimation: Schorfheide (2000)
- 5 Wrap-Up

Why Estimate DSGE Models?

- Move from *calibration* to *data-consistent* models.
- Quantify parameters (preferences, frictions, policy rules).
- Evaluate model fit and compare specifications.
- Produce forecasts and counterfactual policy experiments.

Where MacroModelling.jl Fits

- One of the few Julia package for developing and solving DSGE models.
- User-friendly: it uses `@model` macro writing down a model.
- Automatic variable and parameter handling; steady state solver.
- Perturbation solutions up to third order (with pruning).
- Built-in tools for IRFs, simulations, moments, and estimation with gradient-based samplers.

Estimation Features in MacroModelling.jl

- Kalman filter for linear state-space representation (first-order solution).
- Automatic differentiation of the log likelihood w.r.t. parameters.
- Interface to gradient-based MCMC (e.g. NUTS, HMC) via Turing.jl.
- Helpers to work with steady states, moments, IRFs under estimated parameters.
- Example models included (e.g. Schorfheide (2000), Smets–Wouters).

- Step 1: how to write and analyse an RBC model in `MacroModelling.jl`.
- Step 2: move to a small monetary DSGE model (Schorfheide (2000)).
- Step 3: connect the model to data (observables, measurement equations).
- Step 4: introduce Turing. Estimate a state space model.
- Step 5: specify priors and use NUTS to approximate the posterior.
- Step 6: interpret posterior estimates and implied dynamics.

- Representative household:
 - Chooses consumption and savings (capital), faces intertemporal Euler equation.
- Firm:
 - Cobb–Douglas production with capital and technology.
- Technology:
 - AR(1) shock to TFP.
- A standard real business cycle environment, ideal as a first model.

RBC: Model Equations (Sketch)

- Euler equation:

$$\frac{1}{c_t} = \beta \mathbb{E}_t \left\{ \frac{1}{c_{t+1}} \left[\alpha e^{z_{t+1}} k_t^{\alpha-1} + (1 - \delta) \right] \right\}$$

- Resource constraint:

$$c_t + k_t = (1 - \delta)k_{t-1} + q_t$$

- Production:

$$q_t = e^{z_t} k_{t-1}^{\alpha}$$

- Technology:

$$z_t = \rho_z z_{t-1} + \sigma_z \varepsilon_t^z$$

Defining the RBC Model

using MacroModelling

```
@model RBC begin
```

```
1 / c[0] = (beta / c[1]) *  
(alpha * exp(z[1]) * k[0]^(alpha - 1)  
+ (1 - delta))
```

```
c[0] + k[0] = (1 - delta) * k[-1] + q[0]
```

```
q[0] = exp(z[0]) * k[-1]^alpha
```

```
z[0] = rho_z * z[-1] + sigma_z * eps_z[x]
```

```
end
```

- Time indices in square brackets: [0] current, [1] next, [-1] lag.
- Shock `eps_z` marked as exogenous via [x].

Parameters and Steady State

```
@parameters RBC begin
sigma_z = 0.01
rho_z   = 0.20
delta   = 0.02
alpha   = 0.50
beta    = 0.95
end
```

- One parameter definition per line.
- Package attempts symbolic steady state, then falls back to numerical solver.
- After this step, we have:
 - non-stochastic steady state (NSSS),
 - first-order solution around NSSS,
 - derivatives ready for moments and estimation.

IRFs and Simulations

```
import StatsPlots

# Impulse response functions
plot_irf(RBC)

# IRFs for alternative parameter value
plot_irf(RBC, parameters = :alpha => 0.3)

# Simulations
plot_simulations(RBC)
```

- First call triggers compilation + steady state + solution.
- Subsequent calls are fast: reuse compiled functions and structure.
- Good playground for intuition before touching data.

Steady State and Moments

```
# Steady state and sensitivities
```

```
ss = get_steady_state(RBC)
```

```
# Standard deviations and sensitivities
```

```
sd = get_standard_deviation(RBC)
```

```
# Correlations
```

```
corr = get_correlation(RBC)
```

- Helpers to inspect NSSS and implied second moments.
- All can be evaluated under alternative parameter values using the `parameters` keyword argument.

Why Turing.jl for macro?

- **Universal PPL:** write generative models directly in Julia; supports HMC/NUTS, SMC, PG, Gibbs.
- **State-space friendly:** time loops, latent states, and observation equations are just Julia code.
- **Ecosystem:** works with MCMCChains, StatsPlots, Distributions, ForwardDiff/ReverseDiff.

```
using Turing, Distributions, MCMCChains, StatsPlots
using CSV, DataFrames, Random, Statistics
```

Data: macro series (e.g. quarterly inflation)

- We'll estimate a local-level AR(1) state-space on a single observable y_t (e.g., demeaned inflation). Replace the CSV/column with your series.

```
# Read data (expects a column named :infl, numeric)
dat = CSV.read("macro_data.csv", DataFrame)
y    = collect(skipmissing(dat.infl)) # Vector{Float64}
T    = length(y)
@info "Loaded $(T) obs"

# Optional: de-mean for stability
y = y .- mean(y)
```

Model: local-level AR(1) state-space

Let $t = 1, \dots, T$ with observed y_t and latent state x_t .

$$\begin{aligned} \text{State (AR(1))}: \quad & x_1 \sim \mathcal{N}(0, 10^2), \\ & x_t \mid x_{t-1}, \phi, \sigma_\eta \sim \mathcal{N}(\phi x_{t-1}, \sigma_\eta^2), \quad t = 2, \dots, T. \end{aligned}$$

$$\text{Observation:} \quad y_t \mid x_t, \sigma_\varepsilon \sim \mathcal{N}(x_t, \sigma_\varepsilon^2), \quad t = 1, \dots, T.$$

Model: Turing code

```
T = length(y)
phi ~ Beta(20, 2)           # AR(1) coefficient in (0,1)
sigma_eta ~ InverseGamma(2, 1) # state noise sd (>0)
sigma_eps ~ InverseGamma(2, 1) # obs noise sd (>0)
x = Vector{Real}(undef, T)  # latent state vector
x[1] ~ Normal(0, 10)        # diffuse prior
for t in 2:T
  x[t] ~ Normal(phi * x[t-1], sigma_eta)
end
for t in 1:T
  y[t] ~ Normal(x[t], sigma_eps)
end
```


Inference: NUTS sampling & basic diagnostics

- Gradient-based HMC/NUTS is efficient for differentiable models.

```
model = local_level_ar1(y)
chains = sample(model, NUTS(), MCMCThreads(), 1000, 4; discard_adapt=500)
#check estimation
summ = describe(chain)           # mean, sd, quantiles
println(summ)
ess_rhat(chain)                  # effective sample size & R
```

One-step-ahead forecast (posterior predictive)

- Use last state draws to forecast y_{T+1} .

```
# posterior draws
```

```
phi_draw = vec(Array(chain[:phi]))
```

```
s_eta_draw = vec(Array(chain[:s_eta]))
```

```
s_eps_draw = vec(Array(chain[:s_eps]))
```

```
xT_draw = vec(Array(chain[Symbol("x[$T]")]))
```

```
N = length(phi_draw)
```

```
y_pred = similar(phi_draw)
```

```
# One-step-ahead posterior predictive for  $y_{\{T+1\}}$ 
```

```
for s in 1:N
```

```
  x_next = rand(Normal(phi_draw[s] * xT_draw[s], s_eta_draw[s]))
```

```
  y_pred[s] = rand(Normal(x_next, s_eps_draw[s]))
```

```
end
```

The Schorfheide (2000) Model

- Small-scale New Keynesian-style monetary DSGE model.
- Used to evaluate policy rules and model fit.
- Implemented in `MacroModelling.jl` as `FS2000`.
- Our tasks:
 - Solve the model given a parameter vector.
 - Map model to observed data (output and inflation).
 - Use Bayesian MCMC (NUTS) to estimate parameters.

Model and Parameters in MacroModelling.jl

- Model equations are provided in a template file FS2000.jl.
- We load the package and define parameters similarly to the RBC case.

```
using MacroModelling
```

```
# model definition loaded from FS2000.jl (not shown here)
```

```
@parameters FS2000 begin
```

```
alp    = 0.356
```

```
bet    = 0.993
```

```
gam    = 0.0085
```

```
mst    = 1.0002
```

```
rho    = 0.129
```

```
psi    = 0.65
```

```
del    = 0.01
```

```
z_e_a  = 0.035449
```

Step 1: Load and Prepare Data

- Data in CSV: observed output growth and inflation.
- We:
 - read CSV into a DataFrame,
 - convert to a keyed array,
 - log-transform levels,
 - keep only variables that are observables in the model.

using CSV, DataFrames, AxisKeys

```
dat  = CSV.read("FS2000_data.csv", DataFrame)
data = KeyedArray(
  Array(dat)',
  Variable = Symbol("log_" .* names(dat)),
  Time      = 1:nrow(dat)
)
```

```
data = log.(data)
```

Step 2: Specify Priors

- Prior distributions reflect external information and identification.
- Implemented using `Distributions.jl` with moment-based parameterization.

using Distributions

```
prior_distributions = [  
Beta(0.356, 0.02, mu_sigma = true), # alp  
Beta(0.993, 0.002, mu_sigma = true), # bet  
Normal(0.0085, 0.003),              # gam  
Normal(1.0002, 0.007),              # mst  
Beta(0.129, 0.223, mu_sigma = true), # rho  
Beta(0.65, 0.05, mu_sigma = true),  # psi  
Beta(0.01, 0.005, mu_sigma = true), # del  
InverseGamma(0.035449, Inf, mu_sigma = true), # z_e_a  
InverseGamma(0.008862, Inf, mu_sigma = true) # z_e_m  
]
```

Step 3: Log Likelihood via Kalman Filter

- `MacroModelling.jl` provides a function `get_loglikelihood` that:
 - takes model, data, parameter vector,
 - uses the linear solution and Kalman filter,
 - returns log likelihood.
- This is the bridge between the structural model and the data.

```
ll = get_loglikelihood(FS2000, data, parameters_vector)
```

Step 4: Bayesian Model in Turing.jl

- Use Turing's @model macro.
- parameters is drawn from the prior vector.
- Likelihood added via Kalman-filter log likelihood.

```
import Turing
import DynamicPPL
```

```
Turing.@model function FS2000_loglikelihood_model(data, model)
parameters ~ Turing.arraydist(prior_distributions)
```

```
if DynamicPPL.leafcontext(__context__) !== DynamicPPL.PriorContext()
Turing.@addlogprob! get_loglikelihood(model, data, parameters)
end
end
```


Step 5: Sampling with NUTS

- NUTS (No-U-Turn Sampler) uses gradient information to explore the posterior efficiently.
- Automatic differentiation backend provided via ADTypes (e.g. AutoZygote).

```
using ADTypes
import Turing: NUTS, sample

fs_model    = FS2000_loglikelihood_model(data, FS2000)
n_samples   = 2000

chain_NUTS = sample(
  fs_model,
  NUTS(adtype = AutoZygote()),
  n_samples;
  progress = true
)
```

Inspecting the Posterior

- Standard tools:
 - trace plots,
 - marginal posterior densities,
 - summary statistics (mean, credible intervals),
 - convergence diagnostics.

using StatsPlots, MCMCChains

```
describe(chain_NUTS)
```

```
plot(chain_NUTS)           # trace and density plots
```

```
posterior_means = mean(chain_NUTS)
```

From Posterior to Dynamics

- Evaluate IRFs and simulations at:
 - posterior mean,
 - or multiple parameter draws.
- Compare model-implied moments with data.

Example: IRFs at posterior mean

```
pm = posterior_means
```

```
pars_tuple = (
```

```
:alp    => pm[:parameters_1],
```

```
:bet    => pm[:parameters_2],
```

```
:gam    => pm[:parameters_3],
```

```
:mst    => pm[:parameters_4],
```

```
:rho    => pm[:parameters_5],
```

```
:psi    => pm[:parameters_6],
```

```
:del    => pm[:parameters_7],
```

```
:z_e_a  => pm[:parameters_8],
```

```
:z_e_m  => pm[:parameters_9])
```

- Warmup / burn-in and effective sample size matter.
- Check identification: weakly informed priors can make life hard.
- Start with small models (RBC, FS2000) before jumping to large NK systems.
- Keep separate environments for:
 - model definition and solution,
 - data handling and estimation scripts.

Key Takeaways

- `MacroModelling.jl` provides a full pipeline:
 - model specification,
 - solution and simulation,
 - Bayesian estimation using gradient-based MCMC.
- RBC serves as a simple playground for solution and moments.
- Schorfheide (2000) FS2000 shows how to bring a monetary DSGE to the data.
- Next steps: richer models (Smets–Wouters, heterogeneous agents, occasionally binding constraints).